UNIVERSIDAD DE MURCIA

ESCUELA INTERNACIONAL DE DOCTORADO

TESIS DOCTORAL

Improving the Performance, Portability, and Productivity of Hardware Accelerators

Mejorando el Rendimiento, la Portabilidad y Usabilidad de los Aceleradores Hardware

D. Pablo Antonio Martínez Sánchez

2023

UNIVERSIDAD DE MURCIA
ESCUELA INTERNACIONAL DE DOCTORADO

TESIS DOCTORAL

Improving the Performance, Portability, and Productivity of Hardware Accelerators

Mejorando el Rendimiento, la Portabilidad y Usabilidad de los Aceleradores Hardware

Autor:
Pablo Antonio Martínez Sánchez

Directores:
José Manuel García Carrasco
Gregorio Bernabé García

UNIVERSIDAD DE MURCIA

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA EN MODALIDAD DE COMPENDIO O ARTÍCULOS PARA OBTENER EL TÍTULO DE DOCTOR

Aprobado por la Comisión General de Doctorado el 19-10-2022

D./Dña. Pablo Antonio Martínez Sánchez

doctorando del Programa de Doctorado en

Informática

de la Escuela Internacional de Doctorado de la Universidad Murcia, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Improving the Performance, Portability, and Productivity of Hardware Accelerators / Mejorando el Rendimiento, la Portabilidad y Usabilidad de los Aceleradores Hardware

y dirigida por,

D./Dña. José Manuel García Carrasco

D./Dña. Gregorio Bernabé García

D./Dña.

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita, cuando se han utilizado sus resultados o publicaciones.

Además, al haber sido autorizada como compendio de publicaciones o, tal y como prevé el artículo 29.8 del reglamento, cuenta con:

- *La aceptación por escrito de los coautores de las publicaciones de que el doctorando las presente como parte de la tesis.*
- *En su caso, la renuncia por escrito de los coautores no doctores de dichos trabajos a presentarlos como parte de otras tesis doctorales en la Universidad de Murcia o en cualquier otra universidad.*

Del mismo modo, asumo ante la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada, en caso de plagio, de conformidad con el ordenamiento jurídico vigente.

En Murcia, a 10 de Mayo de 2023

Fdo.: Pablo Antonio Martínez Sánchez

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

Información básica sobre protección de sus datos personales aportados	
Responsable:	Universidad de Murcia. Avenida teniente Flomesta, 5. Edificio de la Convalecencia. 30003; Murcia. Delegado de Protección de Datos: dpd@um.es
Legitimación:	La Universidad de Murcia se encuentra legitimada para el tratamiento de sus datos por ser necesario para el cumplimiento de una obligación legal aplicable al responsable del tratamiento. art. 6.1.c) del Reglamento General de Protección de Datos
Finalidad:	Gestionar su declaración de autoría y originalidad
Destinatarios:	No se prevén comunicaciones de datos
Derechos:	Los interesados pueden ejercer sus derechos de acceso, rectificación, cancelación, oposición, limitación del tratamiento, olvido y portabilidad a través del procedimiento establecido a tal efecto en el Registro Electrónico o mediante la presentación de la correspondiente solicitud en las Oficinas de Asistencia en Materia de Registro de la Universidad de Murcia

A mis padres, mi hermano Daniel y Ana

Índice

Agradecimientos

En primer lugar quiero agradecer a mis padres. Si he llegado hasta aquí creo que es en gran parte gracias a ellos, que me dieron la educación y valores que hoy defiendo, y también la oportunidad de estudiar en la universidad. Haber conseguido todo lo que ha venido después ha sido posible gracias a ellos.

De la informática he disfrutado casi desde que empecé mis estudios en esta universidad, y por suerte esa pasión sigue viva hasta hoy. Con el tiempo, sin embargo, este gusto por el campo de la informática ha ido evolucionando. Y en estos últimos años, una de las cosas que mantenía esa motivación y entusiasmo tan latente han sido las extensas conversaciones con mi hermano Daniel. Hace mucho que sentí en él también esa misma pasión por la informática que yo tenía con su edad. Espero que al leer estos agradecimientos y esta tesis se sienta tan inspirado y motivado como yo me he sentido hablando con él. Y que esto le permita darse cuenta de lo que se puede conseguir con esa pasión por aprender y saber más. Yo diría que con pasión, intelecto y paciencia, en ese orden, cualquier cosa.

En siguiente lugar quiero agradecer a mis directores de tesis, José Manuel y Gregorio. Gracias a Gregorio por su ayuda en todos los problemas que han ido surgiendo y sobre todo por su motivación constante, fundamentalmente en los primeros años en los que empecé a dar clase. Para mi fue una experiencia muy bonita y que siempre quedará en mi memoria. A José Manuel quiero agradecerle especialmente todos estos años que hemos estado trabajando juntos. Quienes me conocen saben que suelo ser una persona muy exigente conmigo mismo. Le agradezco mucho su exigencia durante todo este tiempo, aun incluso en los momentos más complicados, como en los peores meses de la pandemia. Para mí esa exigencia significaba también confianza y certeza de que a veces podía hacerlo mejor. A la larga eso me ha ayudado mucho. Con él he aprendido mucho de arquitectura de computadores, pero sin duda lo que más me ha marcado ha sido todo lo demás. La capacidad de análisis, el pensamiento crítico, el rigor científico. También la parte humana y personal, que a menudo tiende a infrava-

lorarse y olvidarse, y que en realidad es fundamental. No puedo poner ni una pega a cómo ha dirigido la tesis. Creo honestamente que nadie podría haberlo hecho mejor.

Ana ha sido otro pilar fundamental todo este tiempo. Desde un punto de vista motivacional, pero también técnico. No olvidaré la de horas que hemos pasado juntos revisando los artículos, y hasta incluso esta misma tesis, para revisar y corregir el texto en inglés. Si esta tesis está escrita en un inglés decente, es en buena parte gracias a lo que me has enseñado durante estos años.

Y qué habría sido de mí y de este trabajo sin las incontables partidas de juegos de mesa con Jonatan, Siro, Javi, Miguel Ángel y Gonzalo. En el equilibrio está la virtud y después de una dura semana de trabajo, esas tardes con ellos me hacían recuperar toda esa energía. Quiero agradecer especialmente a mi gran amigo Gonzalo, mi compañero de carrera desde el principio y que por suerte puedo seguir contando con él en esta etapa de mi vida. No quiero olvidarme de mi amigo José Luis, con el que pude compartir mi año de máster y al que tengo un gran aprecio.

My stay in Edinburgh was, without a doubt, the best moment in my Ph.D. It has been an honor to work with Michael O'Boyle during my stay. I want to thank him for his hospitality and for the time he spent with me in not always so serious conversations. I want to take this opportunity to say that I miss all my friends from Edinburgh. Celeste, Jackson, Jordi, Li, I miss being there - I miss those days. The memories of Princes Street, Edinburgh Castle, and the streets I crossed to reach the School of Informatics are burnt inside my heart. They are now a dream that a long time ago I had the chance to live. The time I spent with all of you in Edinburgh is something I will never forget.

I also want to thank Biagio and Sandro, from the University of Siena, for the time we shared in Italy and all the work we have done together. The beginning of my research career was not easy for me, but you always have been a great support. It has been a pleasure to work with you.

Last but not least, I wish to thank all my friends and Ph.D. colleagues. Special thanks to my friends Ashkan, Vahid, Sawan, Víctor, Paco, Agustín, Eduardo, Juan Manuel and Sebastián. Thanks for all the lunches we had during this time, all the deep conversations about life, and your company.

Estoy feliz y satisfecho de cerrar este capítulo en mi vida. Al mismo tiempo, no tengo duda de que echaré de menos esto. La gente con la que he compartido esta experiencia, lo que he aprendido por el camino y todo lo que he vivido. Gracias a todos.

Resumen

Introducción y Motivación

Mejorar el rendimiento de los computadores es uno de los objetivos centrales de la arquitectura de computadores. Desde el diseño del primer microprocesador, los ordenadores han evolucionado de maneras innumerables e inimaginables. Conceptos básicos como la jerarquía de memoria, el *pipelining* y la predicción de saltos pronto aparecieron en los primeros microprocesadores. Después, la explotación del paralelismo a nivel de instrucción (ILP) fue el método principal para ganar rendimiento. La explotación del ILP empezó alrededor de los años 80 y terminó aproximadamente en el año 2000. Para entonces, una nueva barrera apareció en la lucha para mejorar el rendimiento de los computadores: la potencia de diseño térmico (TDP). Aumentar las capacidades del *hardware* y su frecuencia implica, en el fondo, incrementar la potencia. Por desgracia, la potencia que entra en un microprocesador debe ser extraída en forma de calor. Este factor dictó cómo se seguiría innovando en el campo de la arquitectura. Para principios de siglo, la atención pasó a los procesadores multinúcleo, lo cual abrió nuevas oportunidades para la investigación y mejoras de las CPU: el paralelismo a nivel de hilo (TLP). Es un momento muy relevante en la historia de la arquitectura de computadores porque, por primera vez, las mejoras en la arquitectura de los computadores llevaban ligadas un incremento de la complejidad en la programación. Al contrario que otras mejoras en los microprocesadores, los multinúcleo no se pueden aprovechar automáticamente. Los arquitectos de computadores dejaron parte del trabajo a los desarrolladores, que tuvieron que incluir paralelismo a nivel de hilo en sus programas. Sin embargo, la fiebre por el paralelismo a nivel de hilo no podía durar para siempre, puesto que tener una gran cantidad de núcleos es inviable, principalmente debido a la sincronización y coherencia de cache. El paralelismo a nivel de hilo ha sido crucial para el rendimiento de los microprocesadores en los últimos años, pero hoy en día

su impacto es menos perceptible. Aunque las mejoras en la arquitectura de las CPU se están ralentizando rápidamente, las necesidades de potencia de cómputo no hacen más que incrementar día tras día. Por ejemplo, uno de las últimas propuestas de DeepMind, AlphaCode, es un modelo de 41 billones de parámetros que necesitó más de 2000 *PFLOP/s-días* para ser entrenado, lo que viene a ser unos 157 megavatios/hora, que se estima que son 16 veces el consumo energético de un un hogar americano al cabo de un año. Entonces, ¿cuál es el nuevo paradigma en el que los arquitectos de computadores deberían trabajar en las próximas décadas? La realidad que debemos afrontar es que mejorar el rendimiento de los microprocesadores es cada vez más complejo. Por lo tanto, el presente y futuro de la arquitectura de computadores está en nuevas arquitecturas, especializadas y más eficientes. En vez de usar una misma arquitectura para todas las tareas, la arquitectura de computadores está evolucionando hacia el uso de arquitecturas especializadas para cada carga de trabajo. Pero, de nuevo, las mejoras en la arquitectura no pueden desarrollarse sin alterar el flujo de trabajo del desarrollo de *software*. Al igual que en la irrupción de los multinúcleo, esta nueva era, la era de los aceleradores *hardware*, no necesita solo depender del *hardware*, sino también del *software*. Realmente, los aceleradores proporcionan una mejor eficiencia energética y potencia de cómputo que las CPU. Pero también está claro que el *software* debe evolucionar para soportar esta inmensa cantidad de diversidad de *hardware*. Esta nueva era, que a veces se dice que va a abrir una nueva edad de oro para la arquitectura de computadores, viene con grandes promesas, pero también con retos extraordinarios.

Uno de los primeros retos consiste en encontrar el nuevo paradigma que permita explotar los aceleradores para mejorar el rendimiento de un computador. Después de los mencionados ILP, TLP y del posterior paralelismo a nivel de datos (DLP), un nuevo paradigma está emergiendo en los últimos años. El paralelismo a nivel de acelerador (ALP) se postula como el siguiente gran paradigma, candidato a dominar los próximos años en los que los aceleradores ya están siendo los actores principales de las innovaciones dentro del ámbito de la arquitectura. El ALP se define como el uso concurrente de múltiples aceleradores. De forma similar a como el ILP utiliza múltiples unidades funcionales, el ALP trata de explotar el paralelismo a nivel de acelerador. De hecho, el ALP ya se está manifestando dentro de los SoC, donde la CPU se encarga de orquestar una extensa lista de aceleradores, desde la GPU hasta otros aceleradores más específicos como DSPs, aceleradores para aprendizaje profundo, etc.

Después del fin de la ley de Moore y de la escala de Dennard, los arquitectos de computadores han de buscar nuevas formas de mejorar el rendimiento de los computadores. Se estima que un procesador en orden ejecutando una instrucción aritmética tan solo utiliza el 6 % de la energía total para ejecutar la

instrucción en sí, mientras que el resto de energía se desperdicia en acceder a las cachés, en llevar a cabo la lógica de control, etc. El caso de un procesador moderno fuera de orden es mucho peor, en el que se estima que el 99.9 % de su energía se desperdicia. Por lo tanto, la respuesta a cómo seguir mejorando el rendimiento de los computadores está lejos de seguir mejorando el rendimiento de las CPU. La computación ha llegado a un momento en el que, si queremos conseguir este objetivo, tienen que aparecer nuevas arquitecturas. Estas son las arquitecturas especializadas, también conocidas como aceleradores. La diferencia respecto a las CPU está en que los aceleradores se especializan en resolver un problema concreto. Esto ha provocado una gran explosión de aceleradores en los últimos años. Primero apareció la GPU, el acelerador más conocido, especializado en aplicaciones como aprendizaje automático, compresión de video, videojuegos y realidad aumentada. Otro ejemplo es las FPGA, que proporcionan un *hardware* reconfigurable, de forma que permiten resolver problemas directamente en *hardware*. Al igual que las GPU, también son aplicables a más de un dominio, desde aplicaciones médicas hasta aplicaciones industriales. En cuanto a aceleradores específicos de un dominio, el aprendizaje automático es claramente el campo más estudiado. La TPU es un ejemplo típico de acelerador para aprendizaje profundo, tanto para el entrenamiento como para inferencia.

Al igual que las CPU, los aceleradores se pueden programar de formas muy distintas. Sin embargo, en el caso de los aceleradores, la variedad en cuanto a lenguajes de programación y tecnologías es significativamente mayor. Mientras que los lenguajes de propósito general como C/C++ se usan típicamente para programar CPU, no hay ningún lenguaje estandarizado para aceleradores. Esto provoca que para usar un acelerador el programador necesite desarrollar un nuevo programa. Así, cuando el número de aceleradores aumenta linealmente, la complejidad en el código aumenta de forma exponencial, debido a lo complejo que resulta mantener códigos diferentes para cada dispositivo. Además de los lenguajes específicos para cada acelerador, también existen lenguajes que permiten escribir un código único y que este se ejecute en múltiples dispositivos. En esta tesis nos refereimos a ellos como «lenguajes de código único». Algunos ejemplos son OpenCL, oneAPI, PHAST o Kokkos. Esta nueva generación de lenguajes de programación abre grandes posibilidades dentro del mundo del desarrollo de software. Sin embargo, también conllevan muchos problemas y retos. Diversos autores creen que existen tres problemas fundamentales dentro de la programación en entornos heterogéneos:

- **Productividad**: el desarrollo de *software* tiene un coste. Normalmente se mide en horas de desarrollo, así que para comparar diferentes lenguajes de programación se puede considerar el numero de horas necesarias para

desarrollar un programa. Cada acelerador necesita un lenguaje distinto, así que la complejidad del desarrollo crece de forma exponencial en entornos heterogéneos. Una alternativa a usar lenguajes específicos para cada acelerador es usar lenguajes de código único. Estos lenguajes (como DPC++) permiten a los desarrolladores escribir un programa una vez y ejecutarlo en múltiples aceleradores. Sin embargo, estos lenguajes pueden ser más complejos que los lenguajes de propósito general (como C++), lo que significa que se necesita más tiempo para terminar un programa. Por otro lado, ya hay una gran cantidad de código escrito, así que para utilizar aceleradores nuevos con este tipo de lenguajes, los desarrolladores tienen que reescribir código. En los últimos años han aparecido nuevas formas de reemplazar código ya escrito con llamadas a librerías optimizadas. En el fondo, esto puede ser útil para compilar código antiguo y reemplazarlo con librerías que utilizan las API de los aceleradores, lo que permite así su uso sin reescribir código. Aunque estas aproximaciones pueden funcionar en ciertos casos y escenarios limitados, son frágiles, porque son incapaces de compilar código complejo y no son extensibles a otros dominios.

- **Portabilidad**: la portabilidad de un lenguaje de programación es proporcional al número de arquitecturas distintas que soporta. Por ejemplo, C++ es uno de los lenguajes de propósito general más populares. Esto es, en parte, porque proporciona un rendimiento excelente. Sin embargo, su portabilidad es muy limitada porque el código solo es compilable a CPU. Los lenguajes de código único son mucho más portables, ya que el código escrito en esos lenguajes se puede ejecutar en diferentes aceleradores (además de la CPU).

- **Rendimiento**: conseguir un buen rendimiento depende de dos factores: el *hardware* y el *software*. Esta vez, con la irrupción de los aceleradores, no iba a ser una excepción. De hecho, el problema del rendimiento es agravado por el hecho de que, como hemos discutido, los aceleradores son muy diversos. Si un programa escrito en un lenguaje de programación de código único funciona bien para un acelerador no hay garantía que de que funcione también bien en otro acelerador. Este problema se conoce comúnmente como *performance portability*. Es un término cuya definición se ha discutido desde su creación, aunque algunos estudios recientes dan una definición más precisa del mismo. En general, una aplicación es *performance portable* cuando consigue un buen rendimiento en *todas* las plataformas hardware que la aplicación soporta. Es importante mencionar que, puesto que los aceleradores están muy especializados, no todos los aceleradores

son adecuados para una tarea (por ejemplo, una TPU no puede realizar tareas criptográficas). Por lo tanto, la *performance portability* solo debería considerarse entre los aceleradores adecuados para una tarea (por ejemplo, la CPU, GPU y *tensor cores* para una multiplicación de matrices).

Por lo tanto, la comunidad está buscando formas de programar sistemas heterogéneos que consigan un buen rendimiento (*performance*), portabilidad (*portability*) y productividad (*productivity*) (P^3). En el problema de las tres P (P^3) es imposible conseguir lo mejor de los tres mundos al mismo tiempo. Algunos lenguajes y librerías, como las que ya hemos mencionado, se han creado para tratar de resolver el problema de P^3 en los últimos años.

Lenguajes de Programación de Alto y Bajo Nivel en la Era Heterogénea

Comenzamos explorando la reprogramación de Caffe, un *framework* de aprendizaje profundo, utilizando la librería PHAST, que permite escribir código único que puede ejecutarse en CPU y GPU. Utilizando este lenguaje de código único, y basándonos en una implementación ya existente, estudiamos los problemas de rendimiento que sufre respecto a la versión original de Caffe, implementada en C++ (para CPU) y CUDA (para GPU de NVIDIA). Concretamente, encontramos oportunidades de mejora de rendimiento en las capas de softmax y de convolución, así como en el *solver*. De esta forma estudiamos si es posible alcanzar la *performance portability* en un de caso de estudio real, como lo es el *framework* Caffe. Experimentalmente, encontramos que la *performance portability* es posible incluso con una base de código tan grande como la de Caffe. Nuestra versión implementada en PHAST obtiene una aceleración respecto a la versión original usando el dataset de MNIST de -15 % y -2 % en CPU y GPU, respectivamente, mientras que usando el dataset CIFAR-10 obtenemos una aceleración de un 51 % y un 49 % en CPU y GPU, respectivamente. En cuanto a la *performance portability*, obtenemos un 91.24 % y un 100 % en MNIST y CIFAR-10, respectivamente. De esta forma conseguimos un código competente respecto al Caffe original, mientras que nuestra versión consta de un único código, al contrario de la versión original, que está implementada en C++ y CUDA, dificultando su desarrollo y mantenimiento.

A pesar de estas ventajas, aprendimos que el desarrollo usando PHAST se vuelve tedioso y complejo cuando se busca obtener una buena *performance portability*. Por ello, estudiamos a continuación el uso de oneAPI, uno de las apuestas más novedosas y fuertes por la computación heterogénea, al mismo caso

de uso, Caffe. En este estudio empezamos con una implementación desde cero y nos centramos en las capas de softmax y convolución. Dentro de oneAPI podemos distinguir dos formas de funcionar: programar usando su lenguaje de código único, DPC++, o bien utilizar su conjunto de librerías optimizadas como oneDNN, oneTBB, oneMKL, etc. En primer lugar implementamos la capa de softmax utilizando DPC++ y posteriormente la de convolución usando oneDNN. El rendimiento obtenido con DPC++ es bueno (solo en CPU, la única plataforma disponible en nuestro servidor que soporta DPC++ en el momento de realizar los experimentos) y el de oneDNN, excelente. Además de un rendimiento superior al obtenido con PHAST, oneAPI tiene potencial para una mayor portabilidad que PHAST, ya que soporta CPU, GPU de NVIDIA y de Intel e incluso FPGA.

Sin embargo, la programación usando DPC++ y oneDNN, al igual que en el caso de PHAST, sigue siendo compleja. En consecuencia, proponemos un nuevo lenguaje de dominio específico (DSL), llamado «*Heterogeneous Deep Neural Network*» (HDNN), que está enfocado al dominio de las redes neuronales profundas. Este lenguaje está basado en MLIR, un nuevo proyecto que nace en Google con el fin de adaptar ideas del conocido LLVM a otros ámbitos y mejorar ciertos aspectos del mismo. La gran diferencia de MLIR respecto a LLVM es que MLIR permite hacer un *lowering* de la representación intermedia en varios niveles o pasadas. Pensamos que este aspecto es tremendamente interesante de cara al desarrollo de un lenguaje de código único que pretende usarse en lenguajes heterogéneos, ya que permite traducir un código de muy alto nivel a otros de nivel de abstracción más bajo en diferentes pasos. Esta traducción por niveles es muy interesante en un contexto de arquitecturas heterogéneas porque permite especializar el código según la arquitectura a la que se vaya a compilar el código. HDNN soporta CPU, GPU de NVIDIA y TPU. Además, incluye una idea novedosa: en vez de compilar el código a cada dispositivo de forma manual, HDNN utiliza por debajo librerías optimizadas de aprendizaje profundo (oneDNN, cuDNN, etc) con el fin de conseguir el mejor rendimiento posible. Nuestros experimentos demuestran que HDNN proporciona una productividad similar a otros DSL, lo que permite una programación mucho más sencilla que otros lenguajes de código único, como los estudiados PHAST y oneAPI. Además, su portabilidad es muy buena gracias a su soporte para CPU, GPU y TPU. Por último, su rendimiento es similar o superior al de otras propuestas, gracias al uso de librerías optimizadas. Creemos que este tipo de propuestas tienen el potencial de solventar el problema de las tres P's (P^3) proporcionando al mismo tiempo portabilidad, rendimiento y productividad.

Compilación de Código Existente a Aceleradores

Reemplazar código escrito en un lenguaje de propósito general (como C++) por una llamada a una API optimizada (que hace lo mismo, pero con un rendimiento superior) es una técnica que ha ganado mucha popularidad en los últimos años gracias a la llegada de la computación heterogénea, debido a que estas APIs a menudo descargan la carga de trabajo en aceleradores *hardware*. Algunos ejemplos de estas técnicas son IDL que define un lenguaje (*Idiom Description Language*) que permite describir patrones de código. Por ejemplo, para detectar una multiplicación de matrices, IDL permite definir su patrón, que posteriormente se detecta y reemplaza por la llamada a la API correspondiente (MKL, openBLAS, etc). Otro ejemplo es KernelFaRer, que busca directamente patrones de multiplicaciones de matrices y las reemplaza con llamadas a MKL. Estas aproximaciones son muy interesantes pero a la vez limitadas, ya que la búsqueda de patrones es una técnica muy frágil, porque pequeños cambios en el código hacen inviable su detección. Un ejemplo más avanzado es FACC, una aproximación aplicada a transformada rápida de Fourier (FFT) que utiliza redes neuronales para detectar las secciones de código a reemplazar, en vez de buscar patrones. Sin embargo, FACC está muy limitado a FFT y no escala a otro tipo de códigos debido a ciertos problemas que se estudian en la tesis.

En este contexto, proponemos «*Algebra and Tensor Compiler*» (ATC), un compilador capaz de detectar y reemplazar programas de álgebra lineal y de cálculo tensorial. ATC funciona en dos etapas. Primero detecta secciones de código que son candidatas a ser reemplazadas por la llamada a una API. Después, analiza todos los candidatos, determinando cual de esos candidatos es equivalente a la sección de código reemplazable y encontrando la correspondencia entre las variables del programa que se está compilando y los argumentos de la API. Esta es la misma aproximación utilizada en FACC, que tiene tres limitaciones:

- FACC necesita que el programador analice el programa de entrada y proporcione un fichero JSON al compilador que contenga información de entrada/salida, como los arrays de entrada, su tamaño y si son arrays de entrada o de salida. ATC incorpora un compilador en tiempo de ejecución (JIT) que permite recolectar esta información automáticamente.
- A la hora de encontrar la correspondencia entre las variables del programa y los argumentos de la API, el espacio de soluciones crece de forma exponencial con el número de argumentos de la API y las variables del programa. Las APIs de FFT no suelen tener más de 3 argumentos, por lo que el espacio de soluciones no es excesivamente grande. Sin embargo, las

de GEMM o convolución pueden llegar a tener hasta 12. En esta investigación proponemos diferentes técnicas que consiguen reducir el espacio de soluciones para hacerlo manejable.

- FACC siempre descarga el trabajo a la API. Sin embargo, dependiendo de la sobrecarga de usar el acelerador, el coste de las copias de memoria, etc, la API no siempre es más rápida que ejecutar el código en la CPU. Por ello, en ATC proponemos el uso de una máquinas de vectores de soporte (SVM) para predecir cuando descargar a la API es beneficioso o no.

Al contrario que otras propuestas, ATC utiliza equivalencia a nivel de entrada/salida para determinar cuándo dos secciones de código son equivalentes. Esto le permite compilar programas de multiplicación de matrices con la estructura típica de los tres bucles anidados, pero también otras estructuras más complejas, como el algoritmo de Strassen, algoritmos paralelos o vectorizados.

Evaluamos ATC con programas de multiplicación de matrices y convolución y lo comparamos con las alternativas más actuales en el campo. ATC soporta la descarga en *tensor cores* para la multiplicación de matrices y en TPU en el caso de los programas de convolución. Diseñamos un conjunto de pruebas de 50 programas de multiplicación de matrices y 15 de convolución. En ambos casos utilizamos programas de GitHub y buscamos programas que estén implementados de formas distintas (distintos algoritmos, estrategias, etc). Aunque FACC solo funciona para FFT, extendemos su funcionalidad para soportar GEMM, aunque no incorporamos ninguna de las mejoras presentadas en ATC. Llamamos a esta nueva implementación FACC*. En programas de multiplicación de matrices, ATC es capaz de compilar 42 programas (84 %), mientras que LLVM-Polly, KernelFaRer, FACC* e IDL consiguen compilar 13, 11, 10 y 6 programas, respectivamente (lo que corresponde a un 26 %, 22 %, 20 % y 12 % de los programas, respectivamente). En los programas de convolución ATC consigue compilar un total de 10 programas (un 66 %). En cuanto a rendimiento, ATC consigue una aceleración media ponderado de 344x en los programas de multiplicación de matrices, mientras que el resto de propuestas no pasan de un 10x. Por último, ATC genera un número de combinaciones posibles significativamente inferior a FACC*, que genera en la mayoría de casos más de un millón de combinaciones para un programa, lo que hace inviable su compilación.

Explotación del Paralelismo a Nivel de Acelerador

Los aceleradores son cada vez más comunes, mejorando las capacidades de cómputo de aplicaciones tanto científicas e industriales (como el aprendizaje

automático) como la de aplicaciones más simples, al alcance de cualquier usuario (como la compresión de vídeo). Esto último es posible gracias a los SoC, unos circuitos que integran una unidad central de procesamiento junto a múltiples dispositivos como GPU y aceleradores. Disponer de varios aceleradores dentro de un mismo chip permite ejecutar ciertas cargas de trabajo en aceleradores específicos, mejorando el rendimiento del computador. Otra opción para aprovechar las capacidades de los SoC sería utilizar varios aceleradores para una tarea común, siempre y cuando dicha tarea pueda ejecutarse en varios aceleradores. Ambas opciones pueden realizarse de forma paralela, por lo tanto aprovechando lo que se conoce como el paralelismo a nivel de acelerador (ALP). En el ejemplo de utilizar varios aceleradores para ejecutar un mismo programa, esto implicaría ejecutar dicho programa (por ejemplo, una multiplicación de matrices) en más de un acelerador de forma concurrente. Sin embargo, aunque el *hardware* lo permite, no existe actualmente ninguna propuesta desde el punto de vista *software* para explotar el ALP en entornos con múltiples aceleradores.

En este marco, presentamos «*Predict, Optimize, Adapt and Schedule*» (POAS), un *framework* que permite planificar aplicaciones de forma automática para explotar el ALP. El *framework* es adaptable a cualquier tipo de aplicación y funciona en cuatro fases bien diferenciadas. En la primera, la fase *predict*, se diseña un predictor capaz de estimar bien el tiempo de ejecución o bien el consumo energético de los aceleradores que participen en la ejecución del programa. En la fase *optimize* se formula el comportamiento de la aplicación y se optimiza. Esta formulación se lleva a cabo como un problema de satisfacción de restricciones (CSP). La optimización puede llevarse a cabo con el fin de minimizar el tiempo de ejecución o bien para minimizar el consumo energético. A continuación, la fase *adapt*, que es la única fase opcional, se encarga de adaptar los valores de salida de la fase anterior. A veces, es necesario adaptar estos valores para que el planificador (*scheduler*) pueda llevar a cabo la planificación (por ejemplo, si el planificador necesita trabajar con tamaños de matrices pero la optimización se ha hecho trabajando con otros datos). Por último, la fase *schedule* utiliza la información proveniente de las fases anteriores para planificar la aplicación en los diferentes dispositivos.

Aplicamos POAS a dos casos de estudio: multiplicación de matrices y convolución. Evaluamos POAS en dos máquinas distintas que disponen de CPU, GPU y *tensor cores* (XPU), un *hardware* específico para multiplicación de matrices. Para evaluar POAS diseñamos un conjunto de pruebas con matrices de diferentes tamaños y formas. En cuanto a la convolución, diseñamos un conjunto de pruebas similar a las entradas típicas de una capa de convolución, con tamaños de imagen de aproximadamente 256x256 y distintos tamaños de filtro. Para evaluar el desempeño de POAS, calculamos la raíz del error cuadrático medio (RMSE)

de la predicción que POAS hace sobre el tiempo de ejecución y de copia en cada dispositivo respecto al tiempo real que luego tarda tanto el cómputo como la copia. A continuación, comparamos la distribución obtenida con POAS respecto a la distribución óptima, que calculamos ajustando manualmente el reparto entre los diferentes dispositivos hasta encontrar aquel reparto cuyo tiempo de ejecución sea mínimo. Los resultados de RMSE son especialmente bajos en CPU para ambas máquinas (RMSE <3) y muy competitivos en GPU y XPU (RMSE <6 en ambos casos). Respecto al reparto óptimo, POAS es capaz de encontrarlo en 4 de las 12 pruebas que se hacen para multiplicación de matrices, quedando en el resto de casos muy cerca del óptimo, obteniendo una planficación con un error por debajo del 4 % respecto a la óptima. En convolución, POAS genera planificaciones muy cercanas a la óptima, desde un 0.1 % hasta un máximo de 2.8 % de error respecto a la óptima. En vista de los resultados, POAS es capaz de proporcionar ALP en entornos heterogéneos y con una sobrecarga despreciable, lo que lo hace un candidato excelente para alcanzar ALP en los sistemas computacionales del futuro.

Conclusiones y Vías Futuras

Cada vez es más evidente que el mundo de la computación se inclina por las arquitecturas heterogéneas y abandona la idea de una unidad central que lleve a cabo todas las tareas. Desde un punto de vista *hardware*, creemos que esta tendencia tiene todo el sentido, pero desde un punto de vista *software* queda demostrado que aún hay mucho camino por recorrer para poder explotar todas las capacidades que el *hardware* es capaz de ofrecer. En esta tesis hemos propuesto soluciones a varios problemas que consideramos relevantes dentro del incipiente mundo de la computación heterogénea, como un estudio sobre la nueva generación de lenguajes junto con una nueva propuesta de lenguaje para el campo del aprendizaje profundo, un novedoso compilador capaz de reemplazar código ya escrito por llamadas a una API optimizada que a su vez descanse en un acelerador, o un *framework* capaz de explotar ALP automáticamente en entornos heterogéneos. Creemos que todas estas propuestas pueden ser de utilidad para reducir la complejidad del desarrollo del *software* en los próximos años, así como un mayor aprovechamiento del *hardware* sin la necesidad de realizar trabajo manual adicional.

Por otro lado, el trabajo desarrollado en esta tesis abre nuevas vías de trabajo muy interesantes:

- Diseñar una intefaz de alto nivel para HDNN. A pesar de los aspectos

positivos de HDNN, los desarrolladores tendrían que trabajar a nivel de MLIR e interaccionar con el dialecto HDNN directamente. Este no es el escenario ideal ya que programar a nivel de MLIR está en un nivel de abstracción muy bajo. Por ello nos gustaría diseñar un lenguaje de alto nivel para HDNN similar a lenguajes fáciles de usar como Python. Esto facitlitaría la extensiblidad y mejoraría la popularidad de HDNN, haciendo más fácil su uso para la audiencia general.

- Extender el compilador ATC a otros dominios y lenguajes de programación. Hemos demostrado que la metodología usada por ATC tiene un gran potencial para reemplazar código de CPU con llamadas a APIs automáticamente. Sin embargo, ATC solo se ha aplicado a multiplicación de matrices y convolución. Extender el compilador a otros dominios demostraría su gran capacidad y extendiría su uso en entornos no solo puramente académicos si no también industriales.

- Diseñar un método de aprendizaje automático capaz de encontrar la correspondencia entre las variables de un programa y los argumentos de una API. En ATC, utilizamos un método de aprendizaje automático para encontrar las porciones de código candidatas a ser acelerables y a continuación un método determinista que encuentre la correspondencia entre variables del programa y argumentos de la API. Una mejora interesante sería utilizar aprendizaje automático en ambas fases, ya sea de forma separada (una red neuronal que haga una tarea y otra que haga la otra tarea) o bien de forma conjunta (una red neuronal que haga ambas cosas de golpe).

- Utilizar modelos de lenguaje grandes (LLMs) para detectar código acelerable dentro de un programa. Actualmente, utilizamos una red basada en PrograML entrenada con el dataset OJClone para detectar cuando una porción de código corresponde a un cierto tipo de programa (por ejemplo, si es una multiplicación de matrices o no). En base a nuestra experiencia, este método funciona bien si se usa conjuntamente con un test de equivalencia a nivel de entrada/salida, ya que esta red actúa como un «filtro» para evitar tener que probar todo el código para comprobar si corresponde a un cierto tipo de programa o no. Debido a los avances recientes en LLMs (como GPT-3), creemos que este tipo de técnicas pueden ser de una gran utilidad para este campo.

- Diseñar un nuevo planificador para ejecutar múltiples aplicaciones en ALP. POAS es un *framework* que permite ejecutar una aplicación el ALP, por lo

tanto ejecutándose en varios aceleradores concurrentemente. Una vuelta de tuerca a esta idea es ejecutar varias aplicaciones en ALP. Esto abre un reto fascinante: el descubrimiento. En otras palabras, descubrir aplicaciones y *hardware* y comprender qué aceleradores son útiles para qué aplicaciones. Pensamos que esta nueva aproximación podría llegara a integrarse dentro del sistema operativo para orquestar la ejecución de programas, de forma similar al planificador de procesos, pero para aceleradores.

- Extender POAS con políticas de planificación más sofisticadas, puesto que las que incorpora actualmente no siempre funcionarán bien en todos los escenarios. Otro aspecto a estudiar es cómo planificar eficientemente las comunicaciones entre la CPU y los aceleradores, ya que esta tiene un impacto notable en el rendimiento, especialmente en entornos con un bus compartido.

Contents

Abstract

With the end of Moore's Law and Dennard's scaling, attention is shifting to new ways of enhancing computer performance. Improving microprocessor performance is becoming increasingly complex, whereas computational power demands still grow tremendously fast. In recent years, we are witnessing a paradigm change: rather than using one single chip, the CPU, for computing everything, computers are evolving into more heterogeneous organizations. In this new configuration, multiple specialized chips compute specific workloads while the CPU orchestrates them, and is only used for actual computing when no other chip can be used. These specialized chips are usually called accelerators. Since they are highly specialized, architecture enhancements have tremendous room for improvement, unlike CPUs. Accelerators are way more efficient than CPUs in terms of performance, energy consumption, or both.

Like multicores, accelerators come with great benefits to computer performance, but also notable challenges to the programming workflow. In environments with multiple accelerators, writing code for each of them is very inefficient since each accelerator is programmed with different languages. Performance is also concerning because programming languages often struggle to exploit hardware to take advantage of its full potential. Lastly, portability is also complicated because when a program is designed for an specific accelerator, it cannot be executed in a different one. Achieving programming languages that provide productivity, performance and portability is known as the P^3 problem. To tackle it, in this thesis, we have studied how two different single-source programming languages perform in real-world scenarios. After studying their performance in each of the three P^3 categories, we found that they struggle to achieve good performance, portability, and productivity at the same time. Therefore, we have proposed a new domain-specific language specialized in deep neural networks that supports multiple heterogeneous architectures and reaches superior results in all P^3 aspects.

Even though we can develop programs with decent portability, productivity

and performance in heterogeneous environments, there is much code already written. Therefore, if we wish to target new hardware, we would need to rewrite this code with new languages in order to use new accelerators. In this thesis, we propose a compiler that automatically matches and replaces existing code with API calls. Since the target API can be reconfigured easily, our compiler can target an optimized CPU library, which is more efficient than executing the handwritten code or an API that relies on a hardware accelerator. Our proposal is designed for C/C++ and recognizes linear algebra and tensor codes. The main strength of this proposal is its ability to recognize simple code (e.g., the three-loop structure of matrix multiplication) as well as complex code constructs (like the Strassen algorithm, hand-optimized vectorized code, etc.).

Furthermore, a notable trend in SoC design, which is becoming increasingly common, is including a sea of disparate accelerators inside the chip. Even though the hardware is already offering performance improvements never seen before, the software is still struggling to take advantage of it. For example, there is no clear way of managing multiple accelerators to accelerate a given workload or how to assign accelerators to the right tasks automatically. Using multiple accelerators concurrently, like how ILP exploits multiple functional units, is called Accelerator-Level Parallelism (ALP). In this thesis, we show a new proposal for exploiting ALP in heterogeneous environments. We present a framework capable of orchestrating multiple accelerators to run a single task jointly, significantly improving performance. We apply our framework to matrix multiplication and convolution use cases, demonstrating that it automatically schedules tasks between accelerators with a low prediction error and a work distribution very close to the optimal.

We expect that the proposal described in this thesis will help to improve the usability and the performance of heterogeneous computing, which will relentlessly establish the standard for future-generation computing systems.

List of Figures

List of Tables

List of Acronyms

AI: Artificial Intelligence (Inteligencia artificial).

ALP: Accelerator-Level Parallelism (Paralelismo a nivel de acelerador).

ALU: Arithmetic Logic Unit (Unidad aritmético-lógica).

AOCL: AMD Optimized CPU Libraries (Librerías optimizadas para CPU AMD).

API: Application Programming Interface (Interfaz de programación de aplicaciones).

ASIC: Application-Specific Integrated Circuit (Circuito integrado de aplicación específica).

BLAS: Basic Linear Algebra Subprograms (Subprogramas básicos de álgebra lineal).

CNN: Convolutional Neural Network (Red neuronal convolucional).

DNN: Deep Neural Network (Red neuronal profunda).

DSL: Domain-Specific Language (Lenguaje de dominio específico).

FFT: Fast Fourier Transform (Transformada rápida de Fourier).

FPGA: Field-Programmable Gate Array (Matriz de puertas lógicas programable en campo).

GEMM: General Matrix Multiply (Multiplicación general de matrices).

HMMA: Half Matrix-Multiply Accumulate (Multiplicación acumulado de matriz en media precisión).

HPC: High Performance Computing (Computación de alto rendimiento).

ILP: Instruction Level Parallelism (Paralelismo a nivel de instrucción).

IMMA: Integer Matrix Multiply Accumulate (Multiplicación y acumulado de matriz entera).

IO: Input/Output (Entrada/Salida).

IR: Intermediate Representation (Representación intermedia).

JIT: Just-In-Time (Justo a tiempo).

LLC: Last Level Cache (Caché de último nivel).

LLM: Large Language Model (Modelo de lenguaje grande).

MILP: Mixed Integer Linear Programming (Programación lineal en enteros mixta).

MLIR: Multi-Level Intermediate Representation (Representación intermedia multinivel).

NPU: Neural Processing Unit (Unidad de procesamiento neuronal).

RMSE: Root-Mean-Square error (Raíz del error cuadrático medio).

SIMD: Single Instruction, Multiple Data (Una instrucción, múltiples datos).

SIMT: Single Instruction, Multiple Threads (Una instrucción, múltiples hilos).

SMV: Support Vector Machine (Máquina de vectores de soporte).

TDP: Thermal Design Power (Potencia de diseño térmico).

TLP: Thread Level Parallelism (Paralelismo a nivel de hilo).

TPU: Tensor Processing Unit (Unidad de procesamiento tensorial).

USM: Unified Shared Memory (Memoria compartida unificada).

Chapter 1

Introduction

Improving computers' performance is one of the central goals of computer architecture. Since the design of the first microprocessors, computers have evolved in innumerable and unimaginable ways. Basic architectural concepts like memory hierarchies, pipelining and branch prediction soon appeared in early microprocessors [77]. After that, the exploitation of instruction level parallelism (ILP) was the primary architectural method for gaining performance. ILP exploitation started around 1980 and ended by the year 2000. By then, a new barrier appeared in the race of improving computer performance: thermal dissipation power (TDP). Increasing the hardware capabilities and the frequency ultimately implies increasing the power. Unfortunately, the power that goes into a processor must be removed as heat. This limiting factor dictated how architectural innovations should be made. By the beginning of the century, attention shifted to multicores, which opened new opportunities for CPU research and improvements: the thread-level parallelism (TLP). It is indeed a very relevant point in computer architecture history because, for the first time, improvements in computer architecture led to increased programming complexity. Unlike other enhancements in microprocessors, multicores are not something that programs can exploit automatically. Computer architects left part of the duty to programmers, which had to include thread-level parallelism in their programs. However, TLP and multicores fever couldn't last forever, as very large amounts of cores have many problems like synchronization and cache coherence, power consumption or scalability. TLP has been crucial for microprocessors' performance in the last years, but nowadays its impact is less noticeable. Despite architectural improvements in CPUs rapidly slowing down, computational power needs are

only increasing every day. AI is constantly improving with impressive results, but this unprecedented explosion of AI also needs computational power and efficiency never seen before. For example, one of the last models by DeepMind, AlphaCode, is a 41B model that required more than 2000 petaflop/s-days to be trained, which in turn is around 175 megawatt-hours, which is estimated to be 16 times the average American household's yearly energy consumption [112]. Then, what is the new paradigm in which computer scientists should work for the next decades? Ahead of us is the fact that improving microprocessors' performance is increasingly complex. Therefore, the present and future of computer architecture are in new, specialized and more efficient architectures. Rather than using one architecture for everything, computer architecture is evolving to use specialized architectures for each workload. But, once again, architectural improvements cannot be developed without disturbing the software development workflow. Like in the irruption of multicores, this new era, the era of hardware accelerators, must not only rely on hardware but also software. Accelerators deliver higher energy efficiencies and computing power than CPUs [44], that is for sure. But it is also clear that software needs to evolve in order to support this new plethora of hardware diversity. This new era, which is called to open a new golden age for computer architecture [78], comes with great promises but also astonishing challenges.

1.1 Accelerator-Level Parallelism

As we have discussed, within each stage of processor architecture evolution, a new paradigm has emerged. Superscalar, pipelined processors allowed architects to run multiple instructions concurrently. With it, Instruction Level Parallelism (ILP) arrived. Later, with multicores, Thread-Level Parallelism (TLP) emerged as a smart way of exploiting parallelism inside chips at another level. Data Level Parallelism (DLP) was the next innovation, which takes advantage of big amounts of data that has to be processed in equal ways. Figure 1.1 shows the evolution of these paradigms over the years.

After the shift from general-purpose CPUs to specialized hardware accelerators, a new paradigm is arriving. Accelerator-Level Parallelism (ALP) is often defined as the concurrent use of various accelerators [80]. Similarly to how ILP utilizes multiple functional units, ALP seeks to exploit workload parallelism at the accelerator level. A very clear manifestation of ALP is already happening in SoCs. Inside an SoC, the CPU is responsible for orchestrating the extensive list of accelerators: GPUs, DSPs, deep learning accelerators, etc. Actually, there are many applications where SoCs are already exploiting ALP, like live video

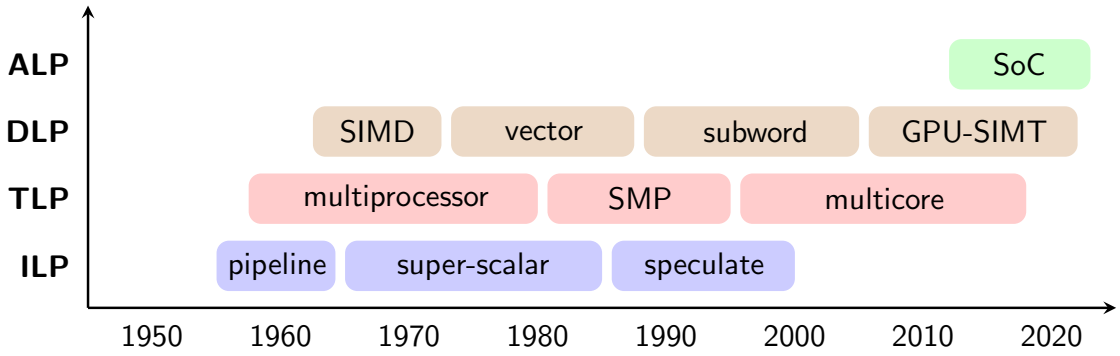


Figure 1.1: Evolution of different computer architecture techniques.

recording. However, for adequately exploiting ALP in heterogeneous environments, new methods should be designed for fully exploiting all the hardware capabilities inside heterogeneous chips.

1.2 Domain-Specific Accelerators

Moore's law, Dennard Scaling and dark silicon. Moore's original prediction in 1965 estimated a doubling in transistor density yearly [127]. In 1975, after reviewing it, he projected a doubling every two years [128]. This estimation is commonly known as Moore's Law. The transistor density growth allowed computer architects to improve performance with relative ease. The more transistors that can fit into a chip, the more cores can contain, more hardware dedicated to caches, etc. However, Moore's Law was not infinite and, as Figure 1.2 shows, it started to fade around 2010. The more time passes, the wider the gap between actual transistor count and Moore's Law.

Accompanying Moore's Law, Robert Dennard also made a projection called "Dennard scaling" [49], stating that as transistor density increased, power consumption per transistor would drop, so the power per mm2 of silicon would be near constant. Since the computational capability of an mm2 of silicon was increasing with each new generation of technology, computers would become more energy efficient. Like Moore's Law, this prediction lasted for some time but also ended years ago, in 2012.

The failure of both Moore's Law and Dennard Scaling motivated the arrival of the "dark silicon" [58] era. Adding transistors and boosting frequency without control is not possible because the CPU must dissipate all this power as heat. In other words, the TDP barrier is a strong limit that architects must comply with. Dark silicon can be defined as a technique based on powering off parts

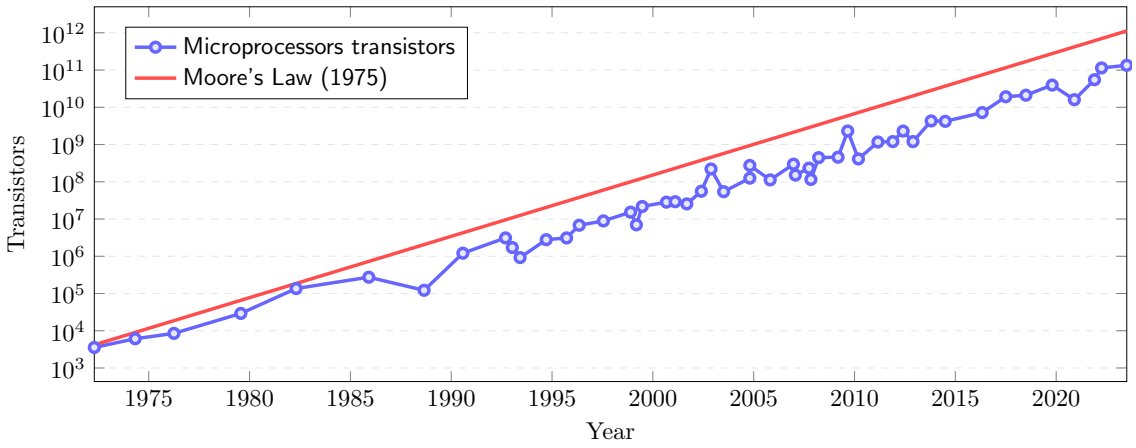


Figure 1.2: Transistors per chip vs. Moore's Law.

of the chip (hence the name) to satisfy TDP limits. If there is part of the chip that is not used for certain tasks, it can be turned off to save power. Over the last few years, the percentage of dark silicon has been increasing.

The end of Moore's law and Denard Scaling is nothing more than a new hurdle in the way of improving microprocessors' performance. How can computer architects keep improving computers' performance, as they have been doing since the conception of the first computer?

The energy efficiency of microprocessors and accelerators. An in-order processor executing an arithmetic instruction is estimated to spend only 6% of the total energy for the instruction itself [42]. The rest of the energy is wasted in supplying the data and instructions from caches, as well as control logic. The case of a modern out-of-order processor is much worse, which is estimated to spend over 99.9% of its energy on overhead [44]. On top of the aforementioned overhead costs, out-of-order processors suffer energy inefficiencies from branch prediction, speculation, register renaming, etc.

The answer to the question about how to keep improving computers' performance seems to be far from improving CPUs' performance. Indeed, many improvements to CPU architecture can still developed. Yet, they are needed very substantial enhancements to hold the tremendous pace that computer improvement has exhibited over the years. This tremendous performance gain has allowed the appearance of several milestones in computer science, like AI. The main source of inefficiencies in microprocessors is their generality. Because CPUs must be able to execute all possible programs and workloads, it prevents them from improving specific aspects for certain domains. Thus, it is very dif-

ficult to achieve great enhancements in microprocessors. Computer history has reached the moment in which new architectures have to appear: domain-specific architectures, also known as accelerators.

Rather than using one device for all the computations (CPU philosophy), accelerators are hardware devices specialized for a given domain. As we mentioned, CPUs can be used for any domain because they are intrinsically generic. This is indeed their source of inefficiency. On the contrary, special-purpose accelerators can eliminate most processors' overhead. For example, they don't typically have to fetch instructions and hence don't spend energy in instructions fetching and decoding. There is no speculation, and hence no work lost due to miss-speculation. Most data is supplied directly from dedicated registers, so no energy is required to read from cache.

To compare the energy efficiency of CPUs and specialized architectures, let's consider the NVIDIA Volta GPU. In the Volta microarchitecture, the matrix multiply-accumulate instruction (HMMA) multiplies two $4x4$ half-precision matrices, accumulating the results into a third matrix. This instruction is implemented in a new type of core inside the GPU, called tensor cores, specialized in tensor computations. It is estimated that the HMMA in Volta utilizes 77% of the energy for the arithmetic, while the rest 23% is wasted [44]. Compared to the in-order processor example, the HMMA instruction is around 12.8x more energy efficient. Even bigger is the gap if out-of-order processors are compared to more specialized accelerators like the TPU [95].

A sea of accelerators. Motivated by all these facts, massive growth in accelerator applications for different domains has risen in recent years. To understand better how accelerators are used, Figure 1.3 classifies the sea of different accelerator types. The first, and most common accelerator is the Graphics Processing Unit (GPU). The GPU is the most popular architecture, being more specialized than the CPU, but still more generic than other architectures. Their field of application is very wide, from applications like machine learning, video encoding, gaming and augmented reality. In the last years, GPUs are incorporating dedicated hardware for specific applications [44] (e.g., tensor cores) and they might even specialize further in the near future [63]. Field-programmable gate arrays (FPGAs) play another crucial role in specialization, as they allow solving specific problems directly in hardware. Their hardware is reprogrammable, which means that can be reconfigured for different tasks. Coarse-Grained Reconfigurable Arrays (CGRAs) also have reconfigurable hardware, although they have much shorter reconfiguration times. Systolic accelerators and ASICs are the last levels of hardware specialization. However, since accelerators are targeted to specific tasks, the majority of them are useful only for a particular domain. The

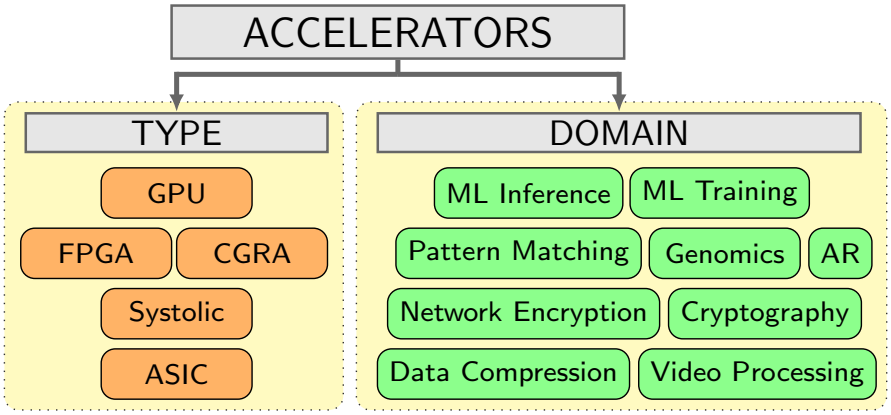


Figure 1.3: Summary of accelerator types divided by application domain and accelerator type.

TPU [95] is a good example of a systolic accelerator specialized for the machine learning domain. In systolic accelerators, computations are performed in 1D or 2D arrays of ALUs. Data enters the array in the first level, then ALUs perform a given computation over the data and pass its output to the ALUs in the following level. Finally, ASICs are accelerators specialized for a specific application, thus offering the most efficient solution. The Darwin accelerator [185], an accelerator for genomics is a good example of ASIC. In the end, many of these accelerators are often included in System on a Chip (SoC) that nowadays power not only HPC superchips [57, 143] but also our smartphones [83], laptops [98] and workstations [168].

1.3 Productivity, Portability and Performance

Like CPUs, accelerators can be programmed in very different ways. However, in the case of accelerators, the variety of programming languages, technologies and frameworks is significantly wider. While general-purpose languages like C/C++ are typically used for programming CPUs, accelerators have no standard language. In fact, each of them requires radically different languages. Even accelerators that belong to the same category, like GPUs, require different, often proprietary languages or frameworks (CUDA [139] for NVIDIA GPUs and ROCm [10] for AMD GPUs). When the accelerator type is different (e.g., a deep learning accelerator and a Fast-Fourier Transform accelerator), the difference in use is huge. The consequence of this diversity is that the volume of code for accelerators is very big and one code cannot be reused for another

accelerator. In a heterogeneous environment where multiple accelerators exist, using one language for each accelerator increases software development complexity rapidly. Therefore, developers seek languages that allow them to write code once and target multiple devices. In this thesis, we refer to those languages as "single-source languages". OpenCL [177] is one of the best known examples of this, although newer and more modern solutions have appeared over the last years, like DPC++ [87], PHAST [153] or Kokkos [55]. This new generation of programming languages and frameworks opens great possibilities in software development. However, they also open many new issues and challenges. Many authors [195] devise three crucial aspects within heterogeneous programming. We share this vision of these three potential challenges:

- **Productivity**: Software development has a cost. It is commonly measured in hours of development, so to compare different programming languages, the number of hours needed to develop a program can be considered. Each accelerator need different languages to be programmed, so software development complexity grows exponentially in heterogeneous environments. An alternative to using specific languages for each accelerator is to use single-source languages. In practice, single-source languages (e.g., DPC++) can be used to reduce software complexity. These languages allow programmers to write code once and target multiple accelerators. However, single-source languages can be more complex than general-purpose languages (e.g., C++), meaning that more time is required to finish a program. On the other hand, developing a new program from scratch is only useful when new software is developed. There is already a huge amount of code base that is already written, so to use new accelerators with single-source languages, developers need to rewrite already existing code. In recent years, new ways of replacing handwritten code with calls to optimized libraries [68, 46] have appeared. In the end, this can be useful for compiling old code with libraries that target the accelerator API, effectively using the accelerator without rewriting code [196]. Although these approaches can work for some niches and limited scenarios, they are brittle, because they are unable to compile complex code and are not extendible to further domains.

- **Portability**: The portability of a programming language is proportional to the number of different architectures that the language support. For example, C++ is one of the most popular general-purpose languages. It is, in part, because it provides excellent performance. However, its portability is very limited since it is only applicable to CPUs. Single-source languages

are way more portable, as the code written in those languages can be executed in different accelerators (besides the CPU).

- **Performance**: That accelerators are significantly more efficient than CPUs is a fact. However, performance has always been a matter of two: hardware and software. This time, with the irruption of accelerators, it is not an exception. In fact, the performance problem is exacerbated by the fact that, as we have discussed, accelerators are highly diverse. If a program written in a single-source programming language works well under a certain accelerator, there is no guarantee that it will work well too on another accelerator. This issue is commonly known as performance portability [132].

  This is a term that has been extensively discussed since its creation, despite the ambiguity associated with the term. More recent research gives us more precise definitions of performance portability [160, 195]. In general, an application is performant portable when it achieves good performance on *all* the hardware platform that the application supports. It is worth noting that because accelerators are highly specialized, not all accelerators are suitable for a task, e.g., a TPU cannot perform cryptography tasks. Therefore, performance portability should only be considered among the suitable accelerators for a given task, e.g., CPU, GPU and tensor cores for a matrix multiplication computation.

Therefore, the community seeks ways to program heterogeneous systems that offer performance, portability and productivity (P^3) [195]. This problem is usually a tradeoff [160] because it is impossible to get the best of the three worlds at the same time. Some frameworks and libraries, like the ones we have already mentioned, have appeared trying to solve the P^3 problem in recent years.

1.4 Objectives and Goals

With the irruption of heterogeneous computing, there are many challenges to overcome in order to unlock the full potential of accelerators. Programming accelerators is difficult, not only to achieve good performance but also to maintain software development productivity. We identify opportunities for these issues in single-source languages, but also in other compilation techniques that we will discuss later. In this thesis, we also study the new paradigm called Accelerator-Level Parallelism, and how it can be exploited to enhance the present and future computers' performance. More precisely, we identified the following research opportunities:

- *Productivity* in software development with accelerators is very low due to the need of developing multiple versions of a program for different devices. Single-source languages can be a good solution to improve it.
- Good *performance* in accelerators is difficult to achieve. Using Domain-Specific Languages with optimized APIs under the hood, instead of compiling directly code to the accelerator, can boost accelerator performance drastically.
- Rewriting code for using accelerators is expensive and unrealistic for large code bases. *Portability* can be further enhanced with novel compilation techniques based on program synthesis, which can be used to make this rewriting automatic, saving countless software development hours and other resources.
- Having multiple accelerators inside a chip (SoC) has much potential, but current schemes are incapable of exploiting it. Smart scheduling can exploit Accelerator-Level Parallelism in accelerator-rich environments, enhancing the energy efficiency and/or *performance* of future computing systems.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives a generic background on different topics relevant to the thesis, like compiler technologies, single-source programming languages, program synthesis and scheduling, among others. The remaining three Chapters (3, 4 and 5), present the three research lines that compose the thesis. All Chapters are organized similarly. First, a brief introduction of the research line is given, followed by a background. The following chapter details our contributions. After that, the evaluation is presented, where we study our proposal and/or compare our work to state-of-the-art approaches. It follows an analysis of related work and lastly, our conclusions. All Chapters are mostly self-contained to ease their reading independently. Lastly, Chapter 6 concludes the thesis and gives some hints for future research.

Chapter 2

Background

In this chapter, we describe the most relevant concepts needed to understand the contributions described in this thesis. First, we detail the existent compiler technologies, necessary to understand Chapters 3 and 4. We later explore the single-source language proposals, study the Caffe framework and explain the performance portability problem. Those concepts are needed to understand Chapter 3. We also explore the field of accelerators, and more precisely, tensor cores, which are needed to better understand all Chapters, especially Chapters 4 and 5. The program synthesis technique is detailed later, which is essential to understand Chapter 4. Lastly, we detail scheduling and co-execution techniques, the fundamental basics to comprehend Chapter 5.

2.1 Compiler Technologies

LLVM. It is a collection of modular compiler and toolchain technologies [105]. One of the most important features of LLVM is intermediate representation (IR). Instead of compiling high-level code to machine code directly, LLVM compiles code into IR. Intermediate representation is a program representation that sits between the source code and the machine code. LLVM features optimization passes at the IR level, which allows reasoning about code without details of its implementation. After the optimization phase, the IR is compiled into machine code. The LLVM IR is based on Static Single Assignment (SSA), which provides many important features in compilation workflows. Nowadays, LLVM is one of the de facto standards for building compilers. It has been extensively used in industry and academia. Some examples of successful projects in academia are

HPVM [104, 56], Glow [169], Polly [73] or IDL [68]. In industry, oneAPI [87], OpenCL [177], XLA [109] and AOCC [45] are some examples.

MLIR. In addition to the aforementioned features, LLVM is composed of different sub-projects (like the mentioned Polly [73]). Multi-Level Intermediate Representation (MLIR) [106, 107] is one of the latest LLVM sub-projects which aims to build reusable and extensible compiler infrastructures. This project was developed by Google with a very concise goal in mind: to improve the development of machine learning frameworks [107]. Google is now using MLIR in TensorFlow [1], but it is also used in many other scientific projects [102, 75, 123]. Like LLVM, MLIR is also based on the concept of IR, but introduces a novel idea: the multi-level IR. Instead of translating source code to LLVM IR directly, MLIR offers a framework to do a *progressive lowering* of the IR. In MLIR, the IR starts from a high-level representation of the original code that gets *lowered* into lower-level IR at each compiler pass. This process is known as *progressive lowering*. Those transformations are technically referred to as *transformation passes*. Each of the *transformation passes* modifies the IR with different goals. Thanks to these step-by-step transformations, the high-level semantics of the high-level code are preserved during IR transformations. MLIR introduces a concept called *Op* (or operations), which represents an individual operation, which can be an instruction, a function or a module. The key idea of operations is that MLIR does not limit the number of MLIR operations but rather encourages developers to extend MLIR with newer operations. To ease the organization and extensibility of operations, MLIR introduces *dialects*. In essence, an MLIR dialect groups different operations and attributes under a common namespace.

HPVM. Heterogeneous Parallel Virtual Machine (HPVM) is a state-of-the-art proposal compiler for heterogeneous computing based on LLVM [104, 56]. It extends the idea of the IR used in LLVM to a hardware-agnostic, parallel IR. HeteroC++ is the straightforward approach for programming in HPVM, but other frameworks like PyTorch and Keras can also be employed. In essence, HeteroC++ is a C++ extension that allows programming in HPVM. At the IR level, HPVM uses a hierarchical dataflow graph representation, which allows different optimization passes within the IR. In conjunction with a runtime system (HPVM-RT), HPVM supports compilation for diverse hardware backends like CPUs, NVIDIA GPUs, FPGAs and even accelerators for FFTs and machine learning. Under the hood, HPVM uses different optimized libraries for high-performance in end devices. For example, in NVIDIA GPUs, HPVM uses cuDNN and the ATen library. Using device-specific optimized libraries provides high-performance whitout compromising productivity or portability.

2.2 Single-Source Languages

2.2.1 The PHAST Library

The PHAST library [154, 155, 153] is a C++ high-level library for easily programming both multi-core CPUs and NVIDIA GPUs. PHAST code can be written once and targeted to different devices via a single macro at compile time, which generates either CPU or NVIDIA GPU executables. The library allows programmers to do so through an STL-like interface that provides containers, iterators, algorithms, and functors.

Containers, iterators, functors and algorithms. PHAST containers are used to encapsulate data. They are collections that store contiguous data on the device and provide a multi-dimensional layout: 1D for vectors, 2D for matrices, and 3D for cubes. These containers can be visited via many kinds of iterators that provide access to the data. In PHAST, iterators are used to retrieve single elements from containers, as well as collections of elements with different shapes. PHAST C++ functors give the ability to customize any computation, which then can be executed inside the PHAST environment. The PHAST library gives the programmer a high-level interface without preventing the code from being expressive and concise. Besides that, PHAST algorithms let the programmer achieve automatic parallelization (on CPUs and GPUs) and vectorization, thus exploiting the target hardware's parallelism. To achieve that, the programmer must use PHAST algorithms. For example, a PHAST algorithm to compute the dot product (`phast::dot_product`) is parallelized internally by the library. A typical PHAST program includes algorithms like `for_each` that take advantage of functors to apply a custom computation to a set of the elements. This construction is inherently parallel, allowing the computation of the elements without any data race.

The PHAST library was compared to other low-level and high-level approaches, from both performance and productivity points of view. It was demonstrated to be a more productive approach in terms of three different complexity metrics (Source Lines Of Code, Halstead's Mental Discriminations, and McCabe's Cyclomatic Complexity) and also to provide competitive performance [153]. Furthermore, PHAST supports both the data-parallelism and task-parallelism approach [154, 155]. By default, when a PHAST algorithm is launched, the library applies heuristics to determine the best values for a set of *parallelization parameters*, which act in the backend as degrees of freedom. For instance, the number of threads and the affinity policy on the CPU side, or the block-sizes, and the Streaming Multiprocessors' scheduling strategy on the

GPU. This choice can be left to heuristics or overridden using library calls, such as `phast::custom::multi_core::set_n_thread` on the CPU to set the number of threads.

2.2.2 oneAPI

oneAPI is the Intel's implementation of the SYCL [99] standard. SYCL defines a standard that must be followed by specific implementations. Other implementations of the SYCL standard are ComputeCpp [172] and triSYCL [199]. Strictly speaking, oneAPI is the composition of the Intel's SYCL implementation, called Data Parallel C++ (DPC++) and other Intel tools, like oneDNN (for neural networks), oneMKL, or oneTBB. This integration between the programming model and other Intel technologies enhances integration with other tools and frameworks and performance. In addition to the SYCL standard features, DPC++ includes some relevant contributions like the USM (Unified Shared Memory) memory model or the concept of sub-items. It makes sense to add USM in oneAPI because one of the things that oneAPI offers is the possibility of targeting Intel integrated GPUs, which use the same address space as the CPU itself. Sub-items are an interesting contribution too because they allow automatic vectorization on Intel CPUs.

Now, to briefly outline the current state of oneAPI, we divide the analysis in two parts; the state of the DPC++ compiler (`dpcpp`) and the state of oneDNN (optimized oneAPI library for deep learning):

a) oneAPI can be downloaded from Intel's webpage [1]. The oneAPI toolkit provides libraries and tools to deploy a single-source application to many heterogeneous architectures. The key component that allows this integration is the `dpcpp` compiler (Data Parallel C++ compiler). At the time of writing, oneAPI's latest official release (`2021.1-beta09`) provides a `dpcpp` compiler that supports Intel CPUs, some Intel GPUs (integrated graphics) and Intel FPGAs, but not NVIDIA GPUs. However, the source code of the `dpcpp` compiler does have support for NVIDIA GPUs [2]. Because this enhancement was introduced recently [70], the NVIDIA backend is expected to be less mature than the rest of the oneAPI standard. Currently, it is disabled in the official builds, but compiling `dpcpp` from source allows to build it with support for NVIDIA GPUs (AMD GPUs support is available too).

[1] Available at `https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html`

[2] Available at `https://github.com/intel/llvm`

b) In oneDNN we find a similar scenario. Even with a `dpcpp` with support for NVIDIA, the official build of oneDNN (version 1.7) does not support NVIDIA GPUs. This happens because it does not interoperate with `dpcpp`, so it is unable to use the `dpcpp` support for heterogeneous architectures. However, oneDNN v2.0 Beta can interoperate with `dpcpp`, but because it has no official release yet, to enable CUDA we must compile it from source. To enable CUDA support, several CMake flags such as `-DDNNL_GPU_VENDOR=NVIDIA`, have to be set to specify the path of CUDA and cuDNN installations.

To sum up, both the DPC++ compiler and oneDNN offer support for NVIDIA GPUs, but none of them has enabled such a functionality in the official builds. To enable CUDA support, compiling from source is needed.

2.3 A DNN Framework: Caffe

Caffe is an open-source deep learning framework initially developed by Berkeley AI Research. While it supports other network models, it is specialized in convolutional neural networks [88]. Nowadays, Caffe has been superseded by more modern solutions, like PyTorch [152] and TensorFlow [1]. However, it is a simple framework compared to others, making it a great choice for research. In this Section we briefly outline the structure of Caffe. The softmax and convolution layers have been deeply studied throughout this thesis. Refer to Appendix A for an detailed explanation of those layers in Caffe.

The framework is organized into modules, where each layer is implemented in a separate file. The majority of these layers are coded twice, in two separate files: one for CPUs (in C++) and another for NVIDIA GPUs (in CUDA). This not only complicates the development but also makes code maintenance difficult. To provide high performance, Caffe implements the majority of layers of neural networks using BLAS [22]. Under the hood, Caffe can leverage three BLAS libraries on the CPU: ATLAS, openBLAS, and MKL. On the GPU, cuBLAS is always used. The specific library to use on the CPU can be chosen at compile time. Moreover, all the layers implement two methods, which correspond to feedforward and backpropagation stages.

Feedforward. In the feedforward phase, all computation are performed as a sequence of operations on the outputs of a previous layer. The final set of operations generates the output of the network. The feedforward phase is used in both inference and training. Specifically, an inference iteration only needs the feedforward phase, while a training iteration is performed with a feedforward phase followed by backpropagation.

Backpropagation. To allow networks to learn from the data, the weights have to be updated, which often is performed using a hill-climbing optimization process called gradient descent. An efficient way to compute the partial derivatives of the gradient is through backpropagation. In backpropagation, values are passed backwards from the end of the network to the beginning. At each layer, backpropagation computes how the loss is affected by each weight. Actually, backpropagation is quite similar to feedforward. Thus, efficient techniques for performing feedforward are usually efficient for backpropagation too [183].

In Caffe, there are two structures to store data, called `bottom` and `top`. All layers take these two data structures as arguments. In the feedforward stage, `top` contains the input for a given layer, while `bottom` is the structure where the layer's output is stored. In the backpropagation stage, the role of the data structures is the other way around.

2.4 Performance Portability

Performance portability has been regarded as a major concern in recent research, despite the ambiguity associated to the term [132]. Fortunately, recent studies [159, 160] give a more precise definition of the term. They define a performance portable application as "an application that can solve a given problem achieving good performance on *all* of the supported hardware platforms". In [159], authors propose a metric for quantitatively measuring performance portability, based on either the architectural or the application efficiency. On one hand, architectural efficiency represents the performance achieved as a fraction of peak performance, representing the ability of an application to utilize hardware efficiently. On the other hand, application efficiency represents the performance achieved as a fraction of best observed performance (using the best known performant application). The performance portability can be quantitatively measured as Figure 2.1 shows:

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 \end{cases}$$

Figure 2.1: Performance portability formula.

where H is the set of hardware platforms and $e_i(a,p)$ is the efficiency of application a solving the problem p on a platform i. Therefore, calculating application

or hardware efficiency depends on the $e_i(a, p)$ used. To compute architectural efficiency, $e_i(a, p)$ should be the performance of application a on platform i divided by the peak performance of platform i. For application efficiency the idea is the same, but using the execution time divided by the best known execution time. Note that if there is a platform i that is not supported in the hardware set of platforms H, the performance portability is 0 (since it is not portable across all platforms).

A quantitative evaluation of performance portability is crucial because it allows comparisons between different applications, but also because it shows precisely how much an application is "performance portable".

2.5 Accelerators and Tensor Cores

In the area of matrix multiplication, accelerators supporting dense and sparse products [11], as well as sparse-only matrix multiplication [150] exist. In computer architecture, a new approach to building accelerators for broader domains is to incorporate domain-specific cores in general-purpose processors [44]. This allows using the general processor for generic tasks while offloading domain-specific ones to specialized cores. Tensor cores [29, 90], included for the first time in Volta microarchitecture, are a good example. They are designed to enhance matrix multiplications performance, which ultimately boosts deep learning applications. When Volta architecture was released, tensor cores were only available through the wmma API. Since then, new APIs are available to use tensor cores directly, like ldmatrix, mma, and mma.sp [181]. Different APIs provide different performances depending on the generation of the tensor core [181]. An easier way to use tensor cores is through CUDA APIs like cuBLAS or cuDNN, which offer an easier interface for the programmer and should always choose the highest performant API. Regarding tensor chores architecture, in Volta they implement a 4x4x4 FP16 matrix multiply and accumulate instruction, HMMA (half precision matrix multiplication and accumulate) [90]. The Turing tensor cores add support for `int8`, `int4` and `int1` data types [89] through a new IMMA instruction. In the Ampere microarchitecture, the matrix multiplication size changes from 4x4x4 to 8x4x8, doubling its FP16 throughput [43]. It also adds new instructions for sparse matrix multiplication, which in turn doubles the throughput of dense matrix multiplications. Tensor cores boost specific applications' performance in an unprecedented way, providing a 4x boost in peak performance compared to CUDA cores, and 8x for the case of sparse matrices [43].

2.6 Program Synthesis and Code Generation

Program synthesis [74] is a well-studied field that has been given many definitions. Generally, program synthesis can be defined as a class of techniques that can generate a program from a collection of artifacts that establish semantic and syntactic requirements for the generated code. In short, program synthesis aims to generate a program that satisfies a certain formal specification. The formal specification can be simple as a description of what the program should do in natural language or a more formal specification like input/output examples. For example, program synthesis is used in [196] to generate code to bind the gap between a C program and an accelerator API, allowing to replace parts of the C program with a library call to the API. In this case, the compiler automatically generates drop-in replacement adapters using input/output (IO)-based program synthesis. When generating a program to solve a problem, the number of potential programs is infinite, even in very restricted contexts, so exploring such search space is computationally prohibitive. Hence, program synthesis always tries to exploit constraints or any information that helps to reduce the search space when generating a program. For that reason, traditional program synthesis approaches are tied to specialized domains, where constraints to reduce the search space of programs are specific to the particular domain.

A step further in program synthesis is automatic code generation using deep learning [17]. This evolution has been made possible thanks to recent deep learning advances like transformer architecture [187]. In recent years, many novel solutions have been proposed, like GitHub Copilot [69], AlphaCode [112] or ChatGPT [146], which can be used for code generation and many other tasks. Unlike traditional synthesis approaches, large language models can generate programs without domain restriction, which is way more powerful. GitHub copilot can be competitive against computer science students generating code [41], while AlphaCode can achieve an average ranking in the top 54.3% in programming competitions against human participants. These tools can aid the programmer to generate code, effectively improving programming productivity. This can be particularly interesting to generate non-complex code like device management or data movement between the CPU and the accelerators, although the usage limits of these technologies are yet not well known. The transformer architecture can also be used for more exotic code generation like compiling C to x86 assembly [16]. Although authors claim that it can only compile 33% of the functions in the benchmark suite, this approach could be improved with a larger network. This approach is interesting in the context of heterogeneous compilation because, in case it would achieve good accuracy, the same could be achieved

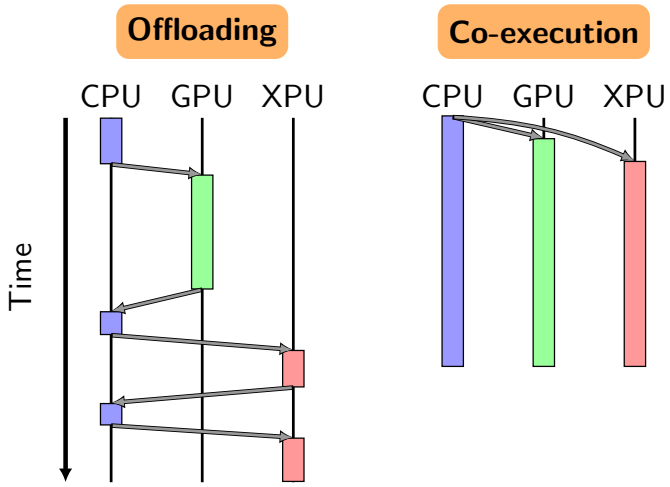


Figure 2.2: Difference between heterogeneous scheduling approaches; offloading (left) and co-execution (right). Arrows indicate data transfers between devices.

for compiling to accelerators rather than x86, allowing to compile code to accelerators directly from traditional programming languages.

2.7 Scheduling and Co-Execution

Task scheduling algorithms have been applied successfully in the past to exploit scenarios where multiple tasks have to be scheduled to different processing elements [191, 190, 116]. Within the same node, scheduling can be divided into two different approaches: offloading and co-execution. In offloading, the idea is to enhance application performance by offloading the compute-intensive part to specialized hardware devices [125, 4]. Traditionally, when an application is scheduled, the scheduler decides whether to run part of the application in one of the available compute elements. To decide on which device the workload should be offloaded, previous works studied the performance of each device and selected the best fitting for this task. In other words, the workload was offloaded to one device at a time, meaning that only one device was executing code in a given period. Unlike task scheduling and offloading, co-execution aims to distribute a single application among different devices and run all of them concurrently. Co-execution provides enhanced performance by employing multiple accelerators, fully exploiting all the capabilities of the system. Differences between offloading and co-execution are depicted in Figure 2.2.

Performance prediction is a common approach in scheduling, and it has been extensively studied in the last decades [59]. This technique consists in predicting the application performance for a given set of hardware resources, allowing to estimate the execution time of an application. In scheduling, we can differentiate between analytical (mathematical models) and non-analytical methods, which often rely on machine learning techniques. However, using standalone analytical or non-analytical models can barely be used to predict performance, so they are typically coupled with some characterization. For example, executing total or partially the application to help deduce its behavior, analyzing the source code, or carrying out some profiling. This characterization may provide information about the hardware and/or software to be measured. In this sense, the roofline model [193] may also be useful to understand the performance behavior. With the heterogeneous computing growth in last years, it has also been proposed for GPUs [114] but also SoCs with different accelerators [79].

Chapter 3

High and Low-Level Programming Languages in the Heterogeneous Era

3.1 Introduction

3.1.1 Motivation

The explosion of alternative architectures to the CPU is bringing unprecedented levels of performance. The downside of this vast hardware variety is on the software side, which has not yet evolved to take advantage of heterogeneous hardware efficiently. Each kind of accelerator needs a different environment, programming languages, libraries and/or tools to be used. Some example languages are CUDA (for NVIDIA GPUs) [139], Hardware Description Languages (HDLs) (for FPGAs) [131], or more generally, Domain-Specific Languages (DSLs) [103], which embrace any language that focuses specific domains, typically used to program specialized accelerators. Since we need different approaches to deploy software on each device, we introduce high complexity in software development. Today, to develop an application to work with CPUs and NVIDIA GPUs, we need to write two different versions of the same algorithm, one for each device. If we are willing to add FPGAs, we would need to add a new version. Additionally, if we have a wide variety of accelerators designed for machine learning, we need to use different DSLs to take advantage of them, we can't just use one for all, even if they all work in the same domain. In the typical

scenario where we need to develop software for each hardware device, software development's complexity grows exponentially.

Major computer manufacturing companies rely on different accelerators to run the heavy workload instead of running it on CPUs. Therefore, the software must join hardware in this challenging but promising evolution, letting programmers take advantage of the unprecedented performance and energy efficiency that these novel architectures can deliver. In this context, the ability to preserve the performance across a wide variety of hardware, known as performance portability, becomes of the utmost importance. In the last year, we have witnessed an increasing interest in this topic. Academia and companies are pursuing the same target. For example, Xilinx is working on triSYCL [199] and Intel recently launched oneAPI [84, 87]. These new solutions are centered around the idea of having a single-source code capable of running on multiple devices.

The appearance of multiple devices of execution has created a problem in parallel programming languages often known as P^3 [160]. This concept refers to three desirable features of modern programming languages:

- *Portability*: The portability refers to the number of hardware devices that the language supports (CPUs, GPUs, FPGAs). C++ and CUDA are not portable, since they only support CPUs and GPUs, respectively. On the other hand, oneAPI is highly portable, supporting CPUs, GPUs and FPGAs.

- *Productivity*: The productivity is reflected in the effort that the developer has to make in order to develop a program. Languages like python are highly productive, because they are very easy compared to others, like assembly language, which is very hard and thus not particularly productive.

- *Performance*: The performance in must not only concern CPU performance, but also the performance of the other supported devices.

Thus, in this new era of heterogeneous computing, we seek programming languages to provide portability, productivity, and performance.

3.1.2 Research Context

To conduct this research, we use the processing of deep neural networks as our case study. DNNs are the central part of today's most relevant applications, like image processing, speech recognition, robotics, games and medical applications [183]. Nowadays, the performance of DNNs is critical and very sensible

to hardware and software characteristics. It is indeed a very relevant way of measuring the performance of different languages, but also the productivity.

In this context, the purpose of this chapter is twofold: First, to show the value of single-source programming in real-world applications, and second, to propose a new single-source, domain-specific language for deep neural networks. The main benefits of single-source development come from the programming side, as this approach allows programmers to be more expressive and develop hardware-agnostic software, thus increasing their productivity. This way, the single-source code-base is shorter and more general, i.e., less tied to the underlying architecture. For the first part of this chapter we use high-level languages that allow single-source coding, while for the second part we use a low-level approach to design and implement our DSL. In this chapter, we:

1. **Study performance portability with single-source programming.** We study the re-implementation of Caffe [88], a machine learning framework specialized in CNNs. Caffe was implemented for CPU and GPU using C++ and CUDA, so each device had a different source code base. We study how to achieve performance portability using:

    - PHAST (Section 3.2.1). We start evaluating the performance of a previous basic PHAST implementation of Caffe [76]. By thoroughly analyzing this implementation, we identify the sources of inefficiencies and improve them by intervening on two different levels. Firstly, from the programmer's point of view, i.e., enhancing the Caffe source code. Secondly, at the internals of the PHAST library. In most cases, working at the programmer level is enough to solve the performance issues. However, to achieve a competitive version of Caffe, we had to include a native convolution in the PHAST library.
    - oneAPI (Section 3.2.2). We start from scratch isolating the softmax and convolution layers in Caffe. Then we implement two version for each layer: one version using DPC++, and another version using oneDNN. We compare each other from the usability and performance standpoints.

    Since machine learning workloads are heavy, the PHAST and oneAPI version must be very efficient against native CPU and GPU code. We selected PHAST over other concurrent single-source approaches like Kokkos [55] or OpenCL [177] because it proved to be a better solution from productivity and performance standpoints [153]. Several works have already compared PHAST with other state-of-the-art approaches [154, 155, 153]. We

also chose oneAPI (developed by Intel) because we believe that it is a very recent and solid proposal for heterogeneous computing that will gain relevance in the upcoming years.

2. **Propose a novel heterogeneous DSL for DNNs:** We present Heterogeneous Deep Neural Network (HDNN), a proof-of-concept MLIR dialect for deep neural networks. HDNN currently supports convolution and softmax layers along with basic I/O functionality. This heterogeneous language supports CPUs, GPUs and TPUs, a domain-specific accelerator for machine learning.

   HDNN programs are portable thanks to our MLIR-based ecosystem, following an idea of progressive lowering of high-level constructs, device-agnostic to low-level operations and device-specific operations. Regarding productivity, HDNN allows programming using a single device-agnostic source code language using MLIR. Rather than compiling code through MLIR infrastructure like other works do [75, 102], HDNN uses optimized libraries for performance-critical operations and compiles only parts of the code that do not have an optimized library available. This way, HDNN achieves competitive performance against state-of-the-art approaches. Moreover, this approach has negligible overhead and thus allows for taking advantage of the full potential of the underlying libraries.

The rest of this chapter is organized as follows. Section 3.2 shows the process of re-engineering an application to achieve performance portability. We divide this topic into Section 3.2.1, where we use PHAST to re-implement Caffe and Section 3.2.2, where we use oneAPI and oneDNN. Then, Section 3.3 presents HDNN, our novel domain-specific heterogeneous language for DNNs. Our experimental methodology, alongside the performance evaluation, is shown in Section 3.4. We discuss related work in single-source programming languages and libraries in Section 3.5. Finally, Section 3.6 concludes the chapter and gives some hints for future work.

3.2 Achieving Performance Portability

3.2.1 Using PHAST

The re-engineering process begins with a performance analysis of the Caffe-PHAST base version. This analysis helps to understand the weak points and where to focus to enhance the performance. We analyze different parts of the

implementation from two points of view. First, we explore possible opportunities to improve the performance from the programmer's point of view. In other words, improvements that come from a better implementation at the Caffe front-end. Second, we explore deficiencies and lack of support inside the PHAST library that can be responsible for other performance shortcomings. To ease the reading of this section, we omit the source code listing and implementations details here. We only discuss the final implementation of each layer here and omit alternative, lower-performance implementations. See Appendix B (or our paper [120]) for an extended, detailed explanation of the final and preliminary PHAST implementations.

3.2.1.1 Caffe-PHAST Base Version

We start revisiting our correct functional Caffe version using the PHAST library we developed in the past. In this Section, this implementation is briefly outlined (see [76] for a more extensive description). The base version is functional, but it does not support all kinds of layers and networks. The blocks identified for the porting were:

- Blob: Stores the network data.
- Inner Product: The inner product layer.
- Convolution: The convolution layer.
- Pooling: The pooling layer.
- ReLU: The ReLU layer.
- Accuracy: Computes the network's accuracy for a specific set of inputs.
- Softmax: The softmax layer.
- Softmax with loss: The softmax layer with the computation of the loss.

This implementation was tested for correctness using Caffe's test mode, which executed different unit tests on each of the ported layers. The tests showed that all ported functionality worked on both CPU and GPU.

An elementary performance evaluation was conducted which reported a loss of 5x and 15x on CPU and GPU, respectively. To sum up, the base PHAST version can run LeNet based neural networks, but the performance was poor. That exercise was a working proof-of-concept but useless in a real-world scenario. It was developed with the sole purpose of having a functionally working implementation, with no attention to performance.

3.2.1.2 Opportunities for Performance Enhancements

Caffe framework is executed with Caffe time mode, which depicts the execution time by the time spent in each of the network layers. Using the hardware platform machine1 (detailed in Section 3.4.1), we find that most layers present in the convolutional networks provide the same performance in the native version of Caffe and the PHAST version, while others perform much worse in the latter. Therefore, we decide to work only on the layers that suffer from performance degradation against native Caffe. We highlight three modules:

- **Softmax layer (feedforward)**: PHAST version is around ∼450x slower in the feedforward stage on the GPU, compared to the original version. In the CPU, the loss is much smaller but still noticeable.
- **Convolution layer (feedforward and backpropagation)**: PHAST version is around 6-7x slower in both feedforward and backpropagation on the CPU and the GPU. The loss factor is much smaller compared to the softmax layer. The convolution layer is the most computationally intensive in the whole network. Thus, this layer is the main cause of the slowdown of the PHAST version.
- **Adam solver**: Adam solver was not ported to PHAST in the base version. As such, even selecting the GPU as the target platform, caused the Adam solver to run on the CPU, which is the fallback choice for non-ported functionalities. This code is ∼200x slower than the original implementation of Caffe, which, on the contrary, takes advantage of the GPU.

The following three sections detail all the steps in order to solve the problems just described. An additional section (Section 3.2.1.6) describes the new primitives and other enhancements introduced into the PHAST library.

3.2.1.3 The Softmax Layer (Feedforward)

The softmax layer implements the softmax function: a mathematical function that maps any set of numbers to probabilities that always add up to 1. The original source code of the softmax layer in Caffe can be found online [1]. To implement the feedforward stage of the softmax layer from scratch, we first study the implementation of the layer from beginning to end. This way, we can

[1] CPU version at: `https://github.com/BVLC/caffe/blob/master/src/caffe/layers/softmax_layer.cpp`
GPU version at: `https://github.com/BVLC/caffe/blob/master/src/caffe/layers/cudnn_softmax_layer.cu`

port the layer in a more general way than trying to replace line by line. While this approach enables global optimizations and modifications to the original algorithm, it requires much more development effort.

The code for the softmax layer is detailed in Appendix B, Listing B.2. In this new implementation, the outer loop is replaced with a single `for_each` algorithm that processes in parallel all the elements. Two changes were made to allow these modifications:

- The input tensor was transposed to swap the two minor dimensions.
- The scale container was transformed into a matrix.

This way, each of the vectors from the input tensor can be mapped to an element in the scale matrix. Additionally, all the previous calculations were moved into a single `for_each` algorithm that uses a single functor. Inside it, the vector is manipulated taking advantage of two in-functor `for_each` algorithms that can leverage an additional axis of parallelism on NVIDIA GPUs [153]. By moving every operation into a single functor and eliminating the outer loop, this new version extracts as much parallelism as possible. Besides, it is much more expressive, simple, concise, and clean than the previous implementation.

3.2.1.4 The Convolution Layer

The original source code of the convolution layer in Caffe can be found online[2]. To compute the convolution, Caffe loops over the batches. At each iteration, Caffe does the convolution calling the `im2col` method (images to columns) and computing a general matrix multiplication. Later, it applies the bias if necessary.

Feedforward. The base PHAST version also used a general matrix multiplication in the convolution layer. Native Caffe benefits from high-performance matrix multiplications on both CPU and GPU using BLAS-based libraries. In PHAST, the matrix multiplication takes advantage of cuBLAS on the GPU side, while no special-purpose library has been integrated on the CPU which leads to worse performance than original Caffe. Instead of improving the PHAST backend for matrix multiplication, we decided to add a native convolution algorithm in the PHAST library. We decided to do so because we believe that

[2]CPU version at: `https://github.com/BVLC/caffe/blob/master/src/caffe/layers/conv_layer.cpp`
GPU version at: `https://github.com/BVLC/caffe/blob/master/src/caffe/layers/conv_layer.cu`

this enhanced convolution algorithm can improve the performance of convolutions. Rather than reusing the matrix multiplication algorithm, a new algorithm specifically designed for convolution can enhance performance thanks to higher specialization. Actually, as we will discuss later, we did not opted for a GEMM-based convolution. Our new algorithm is reviewed in detail in Section 3.2.1.6. In this case, the re-engineering process is more relevant on the PHAST backend than in the Caffe frontend. Hence, the convolution layer at the programmer level is way simpler. The main task to accomplish is preparing the data to pass them to the PHAST native convolution. The PHAST convolution algorithm takes one image (in PHAST terminology, a PHAST cube) as input and computes the convolution.

Essentially, the PHAST implementation calls PHAST native convolution directly with all the batches, which are processed internally in the PHAST library. Thus, the Caffe frontend has to arrange all the data as a PHAST cube container and call the PHAST native algorithm. In this approach, the PHAST convolution algorithm parallelizes both the batches and the filters, so this is a coarse-grained parallelization scheme. Listing B.4 shows this implementation. The new code is even shorter and more concise than before because the new PHAST primitive (`batch_convolution`) contains more logic than the previous one. In the re-engineered version of the convolution, the computation workload goes directly to the PHAST library, instead of relying on the Caffe front-end. Since convolution is the heaviest layer of the convolutional networks, the performance of the native PHAST convolution is crucial to achieve good performance in the whole network.

Backpropagation. The backpropagation phase can be divided in three different steps:

1. The bias gradient calculation.
2. The weight gradient calculation.
3. The input data gradient calculation.

The second and third steps require the calculation of a convolution, whereas the bias gradient does not. Since the backpropagation phase also needs to compute convolutions, it also benefits from the new PHAST convolution primitive. With the backpropagation, we only propose one version, unlike the previous layers. The bias gradient computation is done by accumulating all the matrices. We show this step of our implementation in Listing B.5. In the case of the weight gradient, the computation is essentially a convolution. The new PHAST native primitive is used, as shown in Listing B.6.

There is an essential detail in this step: Caffe performs the convolution transposing the input data, telling the underlying library to transpose the matrix after the matrix multiplication. Under the hood, this transpose is implemented as an implicit operation instead of transposing the data. Thanks to that, the transpose operation is very efficient. Therefore, we decided to do the same inside the PHAST library. We added a new version of the convolution called `batch_convolution_channel_major`, that implements the data transposition as previously explained.

Lastly, regarding the input gradient calculation, the PHAST library has to be extended with two new primitives, `phast::ai::pad` and `phast::ai::rotate_and_pad`. The PHAST code for this phase is given in the Listing B.7.

3.2.1.5 Adam Solver

The original source code of the Adam solver in Caffe can be found online[3]. The CPU code matches almost exactly the algorithm presented by [100], which makes it easier to port the solver using PHAST. The solver algorithm is divided into four computations:

- $m = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
- $v = \beta_2 \cdot m_{t-1} + (1 - \beta_2) \cdot {g_t}^2$
- $t = m/sqrt(v) + eps\_hat$
- $np\_diff = t * (local\_rate \cdot correction)$

where `m`, `v`, `t` and `np_diff` are vector variables inside the Adam solver. Those structures can be retrieved from Caffe directly.

Caffe computes all of them using mathematical functions, provided by BLAS-like libraries under the hood. Like the Caffe version, the PHAST version is also very simple, since we translated the calculation into a single functor that encompasses the same mathematical operations. As `np_diff` is a vector, the `phast::for_each` algorithm iterates over its elements and applies the functor to all of elements in parallel. The other vectors (`val_m`, `val_v`, and `val_t`) are passed to the functor's constructor and accessed by index inside its body. The PHAST port of Adam solver is shown in Listings B.8 and B.9.

[3]CPU version at: `https://github.com/BVLC/caffe/blob/master/src/caffe/solvers/adam_solver.cpp`
GPU version at: `https://github.com/BVLC/caffe/blob/master/src/caffe/solvers/adam_solver.cu`

Note that, except for the square root, all the solver calculations are written directly in a PHAST functor. Unlike the original Caffe code, PHAST implementation of Adam solver does not call optimized libraries for each architecture. Therefore, performance will depend on the quality of the generated code by the PHAST library for each architecture from the code written inside the PHAST functor.

Like in the softmax version, the Adam module was ported directly from the CPU Caffe code. A small refactoring process is needed in the solver to include all the algorithm code inside a single algorithm invocation. Nevertheless, the Caffe code for the softmax layer wrapped this set of steps inside a loop that iterated over the data to be processed, while the Adam solver does not have any `for` loop; it computes each of the steps for all of the data at once.

3.2.1.6 PHAST Upgrades

The re-engineering process was successful in softmax and Adam without any modifications to the PHAST library. Nevertheless, we detected that the convolution needed better support from the PHAST side because a significant percentage of the execution time is spent inside the convolution layer. Henceforth, we prefer adding a native convolution primitive instead of relying on a manual implementation on the Caffe frontend. Following the library's philosophy, we have been able to wrap two architecture-specific implementations (for multi-core CPUs and NVIDIA GPUs) behind a standard high-level PHAST interface without introducing performance-related constraints.

New primitives. The quickest solution was to provide a thin PHAST layer around a BLAS-based `im2col`+GEMM convolution that mimics that found in Caffe's code. However, in recent years, this method has been questioned by studies that propose alternative techniques, such as [52, 67, 203]. We took this opportunity to investigate further. After studying various possibilities, the direct convolution algorithms described by [203] and [67] proved to be promising on the multi-core CPU. They both allow computing a convolution "from the definition": they do not use any matrix multiplications, but use the base algorithm improved with cache and register blocking. Georganas et al. [67] focus also on JIT compilation and architectures with wide SIMD units (e.g., 512 bits), while Zhang et al. [203] investigate loop re-ordering. The latter approach proved the most promising for our workloads, so we investigated in that direction. After studying various possibilities, we choose a direct convolution with a novel loop ordering. We achieve parallelism by processing multiple batches and multiple filters independently.

Also for the GPU convolution we addressed recent literature to provide a better implementation than the classic one. The GPU implementation is a generalization of the technique described by [27]. We enhanced it to handle multiple filters, multiple batches, non-unary strides, padding, and "big" filters with widths significantly bigger than the warp-size. This implementation takes advantage of systolic techniques that permit accumulating values between threads of the same CUDA warp without the shared memory intervention, made possible by CUDA intrinsics like `__shfl_up_sync` or `__shfl_down_sync`. Thus, the shared memory is necessary for extensive filters, where two or more warps need to cooperate to calculate each output element.

New primitives are available under the `phast::ai` namespace (omitted in the next listing for simplicity). The new PHAST primitives developed during this research are:

- `convolution`: We implemented it using direct convolution algorithm on the CPU and a systolic-algorithm on the GPU.
- `batch_convolution`: The same as convolution, but taking a whole batch of images instead of a single image.
- `batch_convolution_channel_major`: The same as batch convolution, but simulates the transposition operation without any computational cost.
- `pad`: Pads each face of an input cube with the passed values and writes them into an output cube.
- `rotate_and_pad`: The same as the pad call, but each face is also rotated by 180°.

Configuration file. In conjunction with the aforementioned primitives, we developed another improvement to the PHAST library during this research. When a PHAST algorithm is launched on a hardware platform, the launch parameters are chosen by heuristics. However, these heuristics do not always provide the best configuration for all of the algorithms calls, and programmers should be able to provide explicit values through API calls.

In this work, we have added the concept of *configuration file*, which is loaded at startup and translated into a read-only global structure. This file contains the parallelization parameters to be set at each algorithm invocation, mimicking the user-provided values. Through extensive search techniques, we populated configuration files to store the best-performing values for each benchmark in Section 3.4.1.

3.2.2 Using oneAPI

In this Section, we have focused in porting two different layers from Caffe, rather than re-implementing the whole framework. We choose the softmax and convolution layer. The softmax layer is a simple, very common layer, that can be useful to compare a native implementation (Caffe in CPU and Caffe in GPU) against a hardware-agnostic one (oneAPI). On the other hand, the convolution layer exhibits weakness and opportunities inside the main component of a convolutional neural network, where performance is crucial. Without losing generality, we decide to concentrate on the feedforward phase. It is the heart of convolutional neural networks in inference mode, and its algorithms have features and access patterns similar to those used in backpropagation [183]. For these reasons, results emerging from this study may well be valid also for most parts of backpropagation too.

To design the oneAPI version of a given layer, we first need to isolate a layer from the Caffe framework. We have to remove all the dependencies of a given layer from the framework. It allows us to run just the layer itself, filling it with arbitrary data and content of various sizes. Therefore, in the oneAPI code, we must respect the data layout of Caffe, because we are going to receive the data in the same way as the Caffe original layer. This way, we can check the correctness of our implementation of the layer by comparing the output of the original Caffe code with our oneAPI implementation. To ease the reading of this section, we omit the source code listing and implementations details here. See Appendix C for an extended and detailed explanation of the DPC++ and oneDNN implementations.

3.2.2.1 Softmax Layer (Feedforward)

Before the softmax is computed, oneAPI's environment has to be initialized (the same happens in the convolution layer). The softmax layer in oneAPI receives the same parameters as Caffe plus one additional parameter: the queue of the device. In the initialization, a device is selected, and a queue for such a device is created and passed to the layer. Note that this allows for generating a single executable that can be run on multiple hardware platforms. The device selected by oneAPI can be controlled:

- Before program execution: using environment variables, or:
- At runtime: relying on hardware detection and selecting one of the available devices.

To implement the layer, we divide the softmax computation in four parts:

1. *Definition of the buffers that store the data to be computed by the softmax layer*. In this part of the code, shown in Listing C.1, the initialization takes place. More specifically, we define the SYCL buffers and workgroup sizes for the rest of the computations.

2. *Exponentials computation (Step 2 in Section A.1)*. First, we request access to write to the output buffer and to read from the input buffer. Them, the exponential is computed in parallel inside a kernel.

3. *Accumulation of the values previously computed with the exponential (Step 3 in Section A.1)*. Both SYCL and CUDA rely on the SIMT model. Therefore, SYCL and CUDA kernel implementation structure are often very similar. We reused the CUDA implementation of this part of the softmax for implementing it in DPC++. Essentially, we adapted some CUDA specific constructs. Also, in this part of the kernel, we use local memory. This will likely have no impact on the CPU performance, but GPUs and other accelerators will probably benefit from this optimization.

4. *The division of all the exponentiated values (Step 4 in Section A.1)*. Again, we implement this part of the softmax reusing and adapting the CUDA implementation. In this kernel, we find some room for performance improvement. However, we are unable to exploit it due to some DPC++ limitations (see Appendix C for more details).

oneAPI support. We have been able to build the `dpcpp` compiler from source successfully, even though we have encountered unexpected issues [4]. We have enabled CUDA support in the compilation process in order to deploy the layer to NVIDIA devices too. In the end, our custom oneAPI softmax layer builds and runs successfully on both CPU and GPU.

3.2.2.2 Convolution Layer (Feedforward)

To implement the convolution layer in oneAPI, we use oneDNN. The oneDNN library is integrated in the oneAPI environment, and allows the acceleration of machine learning workloads. As we mentioned in Section 2.2.2, oneDNN was renamed from MKL-DNN, and in the future, it will work with `dpcpp` to deploy the workloads to the same platforms as those supported by `dpcpp`.

[4] See our issue in the oneAPI GitHub repository for more details: `https://github.com/intel/llvm/issues/2696`

The initialization of oneDNN is similar to the one explained in Section 3.2.2.1. The only difference is that, instead of a device queue, the layer receives the engine kind to be used. The engine creation is shown in Listing C.5. After the engine creation, we retrieve the Caffe variables for populating our variables in oneDNN (Listing C.6). Then, we create the memory descriptors (in Listing C.7) and memory objects (Listing C.8). Those are used to specify the memory layout of the data, the location of the data, as well as the data itself. After creating the memory object and descriptors, we create the forward convolution description (Listing C.9), specifying the convolution algorithm we want. We are using the direct convolution (`algorithm::convolution_direct`), but the Winograd convolution is available in oneDNN too. Then, we link the memory descriptor with its memory objects (Listing C.10) and run the convolution (Listing C.11). Lastly, after the convolution is computed, we have to read the data from the oneDNN memory object and copy it back to the Caffe structure (Listing C.12).

Implementing the layer is much easier with oneDNN than DPC++. The former allows straightforward, but also limited versions of the layer, since we are restricted to what is supported in the oneDNN library. The latter allows more expressive implementations at the cost of lower performance.

oneAPI support. We have been able to build oneDNN from the source (again, with some issues [5]), enabling CUDA support. Nonetheless, we have not been able to run the convolution layer on the GPU. We build an NVIDIA compatible oneDNN library, but at the moment, our implementation is incompatible with oneDNN in NVIDIA GPUs. The underlying implementation of the convolution uses a USM memory (Unified Shared Memory) approach, while the CUDA backend currently only supports a buffer-based model [6]. Although we were able to build the code for CPU and GPU, only the CPU version ran successfully.

3.3 A Novel Heterogeneous Language for Deep Neural Networks

Heterogeneous Deep Neural Network (HDNN) is a proof-of-concept MLIR dialect for deep neural networks that supports CPUs, GPUs and TPUs. HDNN programs are portable thanks to our MLIR-based ecosystem, following an idea of progressive lowering of high-level, device-agnostic to low-level operations

[5] See our issue in the oneDNN GitHub repository for more details: `https://github.com/oneapi-src/oneDNN/issues/885`

[6] See our issue in the oneDNN GitHub repository for more details: `https://github.com/oneapi-src/oneDNN/issues/888`

Operations for Creating Regions	

hdnn.launch {`dev` = *device*} {
...
}
	Create a region and launch everything inside it to the device specified in the dev parameter, which can take the following values:
	"`cpu`", "`gpu`" or "`tpu`"

Operations for Deep Learning	
hdnn.softmax(%i) {`iters` = *N*} : (tensor<NxCxWxf32>) → tensor<NxCxWxf32>	Runs the softmax layer with input %i. iters parameter is optional and allows to run the layer for the specified number of iterations. The softmax operation receives a 3D tensor and outputs a 3D tensor of the same dimensions.
hdnn.conv(%i, %w, %b) {`iters` = *N*} : (tensor<NxCxHxWxf32>, tensor<N'xCxFxFxf32>, tensor<N'xf32>) → tensor<NxN'x?x?>	Runs the convolution layer with the input passed as: • %i: Input image (4D tensor with dimensions NxCxHxW) • %w: Input weights (4D tensor with dimensions N'xCxFxF) • %b: Input bias (1D tensor with dimension N') and outputs the result as a 4D tensor. `iters` parameter is optional and allows to run the layer for the specified number of iterations.
Auxiliary Operations	
hdnn.print(%i) : tensor<?x?x?x?xf32>	Prints a tensor of arbitrary size %i to the standard output
hdnn.random() : tensor<?x?x?x?xf32>	Creates a tensor of arbitrary size with random data
hdnn.return	Used to end a MLIR function (cannot be omitted)

Table 3.1: Summary of HDNN available operations to the user with their description. `f32` refers to simple precision data types.

and device-specific operations. Unlike other MLIR-based approaches, HDNN does not compile code to hardware devices directly using the MLIR ecosystem. Instead, HDNN uses optimized libraries for performance-critical operations, and only compiles code directly when there is no optimized library to use. This way, HDNN is able to achieve competitive performance against state-of-the-art approaches without needing to engineer complex compiler optimizations. HDNN is a step forward in the P^3 problem by providing performance, portability and productivity.

3.3.1 HDNN Frontend

The `hdnn` dialect provides a set of operations to work with neural networks, but it does not add new data types to the MLIR ecosystem since it is designed to cooperate with the already existent data types in MLIR, like the `tensor`. Currently, the dialect is not particularly productive from the programming standpoint because it has to be used directly at the MLIR IR level. For that reason, the `hdnn` dialect is not designed as a user-friendly DSL. We leave for future work to design and implement a top-level DSL to ease the programming task.

```
func @main() -> i32 {
  hdnn.launch {dev = "gpu"} {
    %imgs = hdnn.random() : tensor <10x1x28x28xf32>
    %weig = hdnn.random() : tensor <20x1x5x5xf32>
    %bias = hdnn.random() : tensor <20xf32>

    %cout = "hdnn.conv"(%imgs, %weig, %bias) :
             (tensor <10x1x28x28xf32>, tensor <20x1x5x5xf32>,
             tensor <20xf32>) -> tensor <10x20x24x24xf32>
    %sout = "hdnn.softmax"(%cout) : (tensor <10x20x24x24xf32>) ->
             tensor <10x20x24x24xf32>

    hdnn.print %sout : tensor<10x20x24x24xf32>
  }
  hdnn.return
}
```

Figure 3.1: An HDNN program that runs the convolution and softmax layer in GPU.

3.3.1.1 HDNN Operations

The `hdnn` dialect provides three kinds of operations: operations for creating regions, operations for the deep learning domain, and auxiliary operations. The only operation available for creating regions is `hdnn.launch`, which encompasses an MLIR region that may contain any operation. Those operations will be launched to the device specified in the operation argument. For deep learning, the `hdnn` dialect currently provides two layers; softmax and convolution. Both layers can only work in inference mode since they implement the feedforward phase. Finally, `hdnn` provides auxiliary functions to print an arbitrarily sized tensor, create an arbitrarily sized tensor with random data (useful to fill the layers with data), and an operation to mark the end of an MLIR function. `hdnn` operations are detailed in Table 3.1.

3.3.1.2 HDNN Programming

In addition to the `hdnn` dialect, the HDNN compiler supports the `tensor`, `affine`, `memref`, and `standard` dialects. In essence, this means that the programmer can use any of these dialects to build an HDNN compliant program. However, the normal procedure in an HDNN program is to use only the `hdnn` and the `tensor` dialects, while the rest of them are only used in further lowering passes.

Running a layer in HDNN is straightforward, as can be seen in Figure 3.1.

The first operation corresponds to the launch operation. This operation dictates the device in which the computations will be executed (in the example, the GPU is selected). Then, random 3D tensors are generated. After that, the convolution layer is executed and the output is used by the softmax. Note that the MLIR code inside the launch operation is simple but, more importantly, device-agnostic. Only the `hdnn.launch` operation parameter must be modified to change the target device. At the moment, one limitation of HDNN launch operations is that they cannot be mixed together as they are treated as completely different tasks. Thus, the current implementation dictates that only the data created inside a region can be used. Still, it is not possible to use data from another region or data created outside of the region.

HDNN programming is straightforward because the function calls hide the complexity of an HDNN program. This makes sense because programmers often do not need to know or modify the code of a neural network layer. Furthermore, other common constructions like conditions or control flow can still be used in HDNN. One of the strengths of HDNN is the fact that data is stored in tensors, which are extremely flexible (following the MLIR idea, they are expressed in a very high-level way). Thus, connecting different layers in HDNN is straightforward, as shown in Figure 3.1.

3.3.2 HDNN Backend

The HDNN backend architecture is built up of two components. The first one is the HDNN compiler (which we refer to as `hdnn-opt`), and the second one is the HDNN runtime (composed by the CPU, GPU and TPU runtimes).

3.3.2.1 HDNN Architecture

The HDNN compilation flow is depicted in Figure 3.2. First, the HDNN compiler transforms the MLIR code to LLVM. The original MLIR code suffers different modifications (partial lowering) before being converted to the LLVM IR code. This code is then compiled to an object file using clang. The HDNN runtime is written in C++, so it can also be compiled to object files using any compiler. Finally, object files are linked into a single binary file. Depending on the original MLIR code, the final binary file is executed on CPU, GPU or TPU. However, as it is tied to a specific device, different devices cannot execute the MLIR code concurrently. Nonetheless, thanks to the HDNN design, this limitation could be eliminated easily to allow co-execution. We explore co-execution opportunities in Chapter 5.

3.3.2.2 HDNN Compilation Process (*Lowering*)

An HDNN program has to be *lowered* to transform the HDNN high-level IR code (MLIR) to a lower-level IR (LLVM). The lowering process is divided into three different passes, as depicted in Figure 3.2.

First transformation pass. The first pass (marked as 1º in Figure 3.2) takes as input the HDNN source file. In this file, operations are used inside the launch operation, so they are device-agnostic. The main goal is to replace device-agnostic with device-specific operations. For this task, the HDNN dialect has not only device-agnostic operations but also device-specific ones. For example, for the convolution, HDNN has the device-agnostic `hdnn.conv` and the device-specific `hdnn.conv.cpu`, `hdnn.conv.gpu`, and `hdnn.conv.tpu`. The multi-level nature of MLIR is convenient in this case; it is very interesting for heterogeneous compiling, as multi-level IR can transform a generic IR to one that is device-specific. In addition to the mentioned transformations, this pass also adds the operation `hdnn.init_gpu` when a launch operation on GPU is detected (explained in the following pass). Another objective is to lower high-level data types to lower-level ones, e.g., the `tensor` data type is lowered to the `memref` dialect to work with lower-level operations, like loads and stores.

When the pass has finished, only the `affine`, `memref`, and `standard` dialects are legal. Lastly, the `hdnn` dialect is said to be *dynamically legal* because not all the operations in the dialect are legal, just the device-specific operations generated by the compiler in the current pass. These device-specific operations are not available to the user as only the compiler can generate them. Note that the MLIR code generated in this pass is no longer device-agnostic since we have already

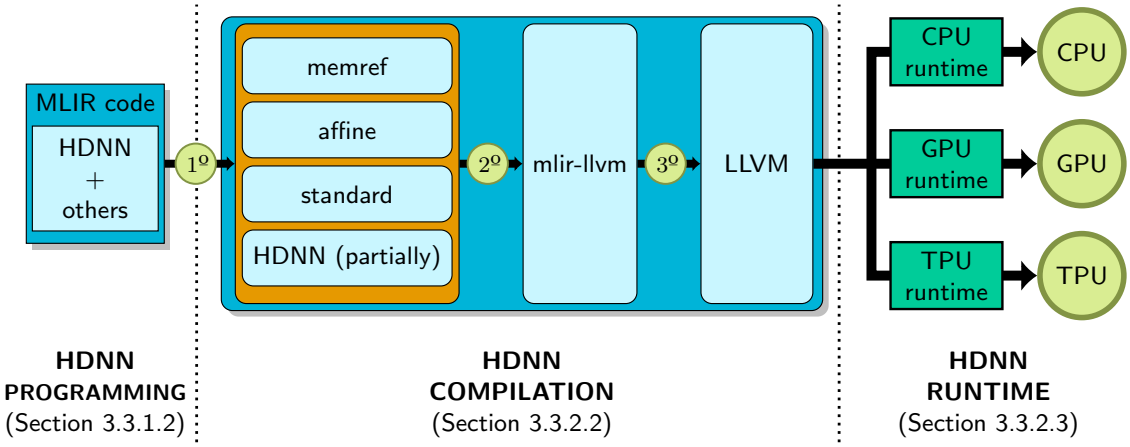


Figure 3.2: HDNN compilation flow.

specialized the operations to target a specific device. We leave for future work to experiment with more sophisticated approaches, like dynamically selecting the device depending on the system's load, thus allowing co-execution.

Second transformation pass. The input of the second pass (marked as 2^o in Figure 3.2) is the output of the previous one, which contains deep learning device-specific operations (neural network layers). The device-specific operations found in this pass have to be lowered. To do so, the `hdnn-opt` inserts calls to the HDNN runtime, which uses optimized libraries for running the layers. Depending on the layer and the device selected, the compiler also inserts calls to initialize the library used to do the computations. Data types used by the library also have to be initialized, and the compiler generates the IR code at this moment. It is worth noting that when this pass has finished, only the `mlir-llvm` dialect is legal.

The rest of the transformations performed by the second pass are dependent on the device being targeted:

- **CPU lowering:** This lowering is the easiest because there are no memory movements. Thus, the compiler inserts the calls to the CPU runtime to lower the code when the CPU is selected.

- **GPU lowering:** When the IR contains GPU-specific operations (e.g., a convolution launched to the GPU), the compiler needs to provide mechanisms for data management between CPU and GPU. For this duty, MLIR provides a GPU dialect. During the second pass, the compiler generates GPU operations (that are part of the `gpu` dialect). However, the `gpu` dialect operations cannot be converted to `mlir-llvm`[7]. Our approach to circumvent this problem is to do a transitive lowering of the `gpu` dialect. Therefore, these operations (part of the `gpu` dialect) are immediately lowered to other lower-level operations instead of leaving them to be translated in a further pass (transitive lowering). That is why the `gpu` dialect does not appear in Figure 3.2: The dialect appears in the second pass but it immediately disappears, as it is lowered to lower-level operations. Another important thing to do is to lower the `hdnn.init_gpu` operation. This HDNN operation aims to initialize both a CUDA stream and a cuDNN handler. This operation is lowered in this pass, and it is done by calling the HDNN GPU runtime, which will take care of this.

[7]While it has been discussed if this is how it should work or not, at the moment of writing, converting the `gpu` dialect to `mlir-llvm` is not possible.

- **TPU lowering:** In standard MLIR, there is a specific dialect for the GPU, but not for the TPU. The `gpu` dialect is used to manage memory allocations and memory transfers between CPU and GPU. Even though in HDNN we use a specific dialect to handle the GPU, we do not follow the same approach for TPU. Note that this is our design decision for HDNN, it is not something imposed by MLIR: we decided not to design a TPU dialect due to a limitation in the TPU runtime of HDNN. Currently, the memory allocation and movement in the TPU are managed implicitly, so it does not make sense to have a dialect that can express these operations as they cannot be executed using our current TPU runtime. The TPU lowering inserts calls to the TPU runtime, which handles both the memory and the computations in the TPU. A lower-level TPU runtime, along with a `tpu` dialect, would allow explicit memory management, as well as other optimizations. We leave this research line as future work.

Third transformation pass. The last pass (marked as 3º in Figure 3.2) transforms the `mlir-llvm` into LLVM. This transformation is an automated process managed entirely by MLIR. Thus, the HDNN compiler invokes the appropriate MLIR function to do it. The output of this pass is LLVM code, which will be further compiled with the HDNN runtime into the executable file.

3.3.2.3 HDNN Runtime

The HDNN runtime consists of one generic runtime and three device-specific runtimes; the CPU, GPU and TPU runtimes. The generic runtime is executed on the CPU. It provides functions for time measurements (useful for the evaluation) and functions for generating random floating-point values, which are called when the operation `hdnn.random` is used.

The HDNN runtime essentially acts as a middleware between the LLVM code generated by MLIR and the corresponding optimized library. Each runtime defines custom functions to operate with the corresponding library. These functions are used by the IR code, as depicted in Figure 3.3. The HDNN compiler needs to generate function definitions for all of the functions defined in the runtime to connect the IR with the libraries. These definitions are not linked when the LLVM code is generated since they reference functions that do not exist in the LLVM code. They are linked when the LLVM code is compiled altogether with the HDNN runtime because the needed functions are provided by the device-specific runtime. Therefore, when an HDNN program is compiled, the executable contains the HDNN high-level code, together with the HDNN runtime needed to run the program. If the programmer writes code for

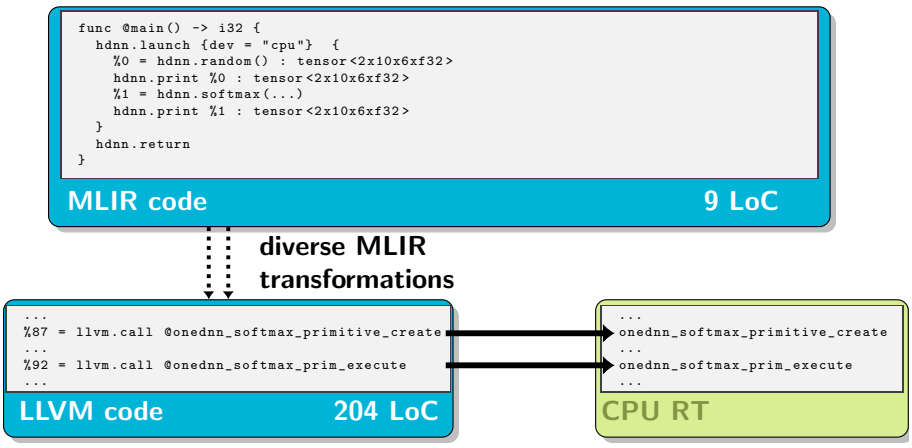


Figure 3.3: HDNN example lowering with runtime communication.

something that does not have an optimized library (e.g., loops, `hml.print`, etc), HDNN compiles code directly to the device. Then, the softmax layer written in Figure 3.3 is executed using the optimized library, so the generated code simply calls the HDNN runtime (in this case, the CPU runtime). The rest of the code is compiled to the CPU using the MLIR ecosystem, so the HDNN runtime is not invoked for that code section.

Device-specific runtimes serve two purposes: to manage library-specific data structures and run both layers. Runtimes delegate the layers' computation to the optimized libraries. We choose the best performant deep-learning library for each device: oneDNN is used for the CPU runtime, cuDNN for the GPU runtime, and PyTorch for the TPU runtime. The case of the TPU is different because there is no optimized, ready-to-use library (like oneDNN and cuDNN). Hence, we explore different alternatives like PyTorch, TensorFlow, or XLA. PyTorch and TensorFlow provide an easy interface to run any layer on the TPU, but they only work with Python (both libraries offer limited support with languages different than Python). Therefore, to use TPUs from HDNN, we design a TPU runtime that takes the inputs from HDNN, sends them to a Python code and returns the data to HDNN coming from Python. This Python code uses PyTorch to run the layers on the TPU and XLA to communicate with the TPU itself. However, running a layer using PyTorch directly only makes use of 1 TPU core. Thus, we implemented a basic algorithm inside the TPU runtime to parallelize the code among the TPU cores. The algorithm follows an `allreduce` scheme, a common approach in the execution of distributed DNN workloads [20]. Essentially, the number of batches is divided between the number of TPU cores, and the master core gathers the partial results at the end of the execution.

3.4 Evaluation

In this Section, we compare PHAST, oneAPI and HDNN with the P^3 approach: performance, portability and productivity. First, we detail our test bed in Section 3.4.1. We then study the P^3 results of PHAST (Section 3.4.2), oneAPI (Section 3.4.3) and HDNN (Section 3.4.4). First, we show a detailed evaluation of the performance, followed by an analysis of the portability and productivity of each approach. Lastly, we offer a summary of each proposal in Section 3.4.5, showing how they perform in these three metrics compared to each other. In this comparison, we also consider a popular general-purpose language like C++ to illustrate the benefits of single-source programming languages. Next, we detail how we evaluate P^3 and give first results for C++:

- **Performance.** We evaluate performance with execution time. C++ is a well-known language with deeply studied compilers like gcc or icpc. In general-purpose languages, there is probably no better language for high performance than C/C++. Therefore, in our analysis we consider that C++ has optimal performance.

- **Productivity.** We evaluate productivity as the effort made by a developer to write efficient software. A good way of measuring productivity is through lines of code. Thus, we compared the complexity of programs using different programming languages. However, comparing a program's complexity is not something purely objective. Different metrics have been proposed over time, but here we will use the lines of code of a program for simplicity. Before calculating the lines of code, non-essential ones (blank lines, comments) were removed, allowing for a fair comparison. We evaluate two layers' implementations of convolutional networks: softmax and convolution. The C++ programs were taken from the official Caffe repository.

  We also consider our experience programming in each language. In particular, we consider development time, which does not always have to be correlated with lines of code. We did not measure our development time, so this cannot be quantitatively measured like lines of code, but we believe that programming experience is an essential aspect of productivity. We consider C++ to have good, although not optimal productivity.

- **Portability.** We evaluate portability as the support range for diverse hardware devices. C++ is clearly the less portable language in our analysis since it can only be compiled for CPU. We rate C++ portability as the worst of all considered languages.

3.4.1 Test Bed

The evaluation platform is divided into three machines: machine1, machine2 and machine3. PHAST is evaluated in machine1, with an Intel Core i9-9900K and an RTX 2080. oneAPI is evaluated in machine2, with a dual-socket Xeon Gold 6238 and an RTX 2080 Ti. HDNN is evaluated in machine2 and machine3, which features a TPU v2. Each experiment is repeated five times and the values shown are the average over these independent runs (measured in seconds). In the PHAST and oneAPI evaluation, we study the performance of isolated layers. We design a set of inputs to gather information about the performance of each implementation.

3.4.1.1 Hardware

The summary of the hardware platform is shown in Table 3.2 and a detailed hardware specification for each device is shown in Table 3.3.

Table 3.2: Hardware configuration for the three machines used in the evaluation.

	CPU (Intel)	GPU (NVIDIA)	TPU (Google)
machine1	Core i9-9900K	RTX 2080	-
machine2	Xeon Gold 6238	RTX 2080 Ti	-
machine3	-	-	TPUv2

Table 3.3: Hardware specifications for the testbed environment (per chip).

	CPU 1	CPU 2	GPU 1	GPU 2	TPU
Model	Xeon Gold 6238	Core i9-9900K	RTX 2080 Ti	RTX 2080	TPUv2
Release date	Q2 2019	Q4 2018	Q3 2018	Q3 2018	Q2 2017
Manufacturing process	14nm	14nm	12nm	12nm	16nm
TDP	140W	95W	250W	215W	280W
Chips per host	2	1	1	1	4 [1]
Cores/chip (total)	22 (44)	8	4352	2944	2 (8)
Maximum Frequency	3700 MHz [2]	5000 MHz [3]	1545 MHz	1710 MHz	700 MHz
Peak performance (SP)	2.95 TFLOP/s	921 GFLOP/s	13.4 TFLOP/s	10.0 TFLOP/s	3.00 TFLOP/s
Cache memory	30.25 MB L3	16 MB L3	5.5 MB L2	4 MB L2	32 MB
Main memory type	DDR4	DDR4	GDDR6	GDDR6	HBM
Memory frequency	2933 MHz	2666 MHz	1750 MHz	1750 MHz	-
Memory bandwith	140.7 GB/s	42.6 GB/s	616.0 GB/s	616.0 GB/s	∼600.0 GB/s

[1] A single TPUv2 board contains 4 TPU chips with 2 TPU cores each.
[2] While maximum frequency is 3.7 GHz, real frequency when the CPU is using all the cores and running AVX-512 code is 2.1 GHz. Therefore, we used 2.1 GHz to calculate the peak performance.
[3] While maximum frequency is 5.0 GHz, real frequency when the CPU is using all the cores and running AVX2 code is 3.6 GHz. Therefore, we used 3.6 GHz to calculate the peak performance.

3.4.1.2 Software

The software configuration of the three machines used in the evaluation is as follows:

- **machine1:** Runs Ubuntu 18.04 with kernel 5.0.0-36-generic. The NVIDIA driver version is 430.50. The CUDA version is 10.1, and the cuDNN library version is 7.5.0. The installed C++ compiler version is gcc 8.3.0. The Caffe implementation and the PHAST library itself are compiled with this configuration. The PHAST library version used is 1.1.1. The base Caffe framework was obtained from the official git repository using the `99bd997` commit. Finally, the original Caffe is built with openBLAS as the underlying BLAS library for CPU and cuDNN for GPU.
- **machine2:** Runs CentOS Linux 8.2 with kernel 4.18.0-193.14.2.el8_2.x86_-64. The NVIDIA driver version is 450.51.06 The CUDA version used is 11.0, and cuDNN is in version 8.2.4.

  BVLC Caffe is compiled using CUDA and the GNU C++ compiler version 8.3.1. The Intel Caffe version is compiled using icpc 19.1.2. The oneAPI softmax and convolution implementations were compiled using clang 12.0.0. The `dpcpp` compiler was built from source using the commit `6336913` [8]. The oneDNN version is v2.0-beta10 (dnnl_lnx_1.96.0_cpu_-iomp.tgz) [9]. The TBB version is `tbb-2021.1-beta08`, MKL version is 2021.1 Beta Update 9, and openBLAS version is 0.3.3.

  HDNN is built using LLVM, which was downloaded from the official git repository, obtaining the code corresponding to the commit `cf72768`. Our custom implementations of softmax and convolution examples of oneDNN and cuDNN are built using gcc 8.3.1. Furthermore, we use the same oneDNN and cuDNN version as previously mentioned for HDNN.
- **machine3:** This machine belongs to the Cloud TPU service in Google Cloud Platform. This system runs Debian 10 with kernel 4.19.0-14 kernel. In this case, we used the same LLVM version to build the HDNN compiler, and for the TPU backend we used python 3.7 and PyTorch 1.9.

3.4.2 PHAST

3.4.2.1 Performance

The performance evaluation is divided into three parts:

[8] Available at `https://github.com/intel/llvm`

[9] Available at `https://github.com/oneapi-src/oneDNN/releases`

- Isolated layers: We compare the execution times of the only layers that were modified in our work, which we isolated from the rest of the framework.
- Whole network: We measure the total execution time of the whole network.
- Performance portability analysis: We study the performance portability of our PHAST implementation.

In this Section, we only show the performance results for the whole network, as well as the performance portability analysis. For the evaluation of the isolated layers, refer to Appendix B.5.

When speaking about performance, we consider the inverse of the elapsed time as our performance metric. Thus, we adopt the usual notion of the ratio between two achieved performance scores for speedup, which is the inverse of the ratio between the elapsed times. We use a LeNet network in inference mode with the MNIST, and CIFAR-10 datasets for the full network runs. We run the networks for 1000 iterations. To provide a better insight into the performance, we use Caffe time mode. This way, we can obtain the execution time for the whole network and each of the layers.

Figures 3.4, 3.5, 3.6 and 3.7 compare the performance of the native Caffe version in C++ and CUDA, against the single-source version with the PHAST library.

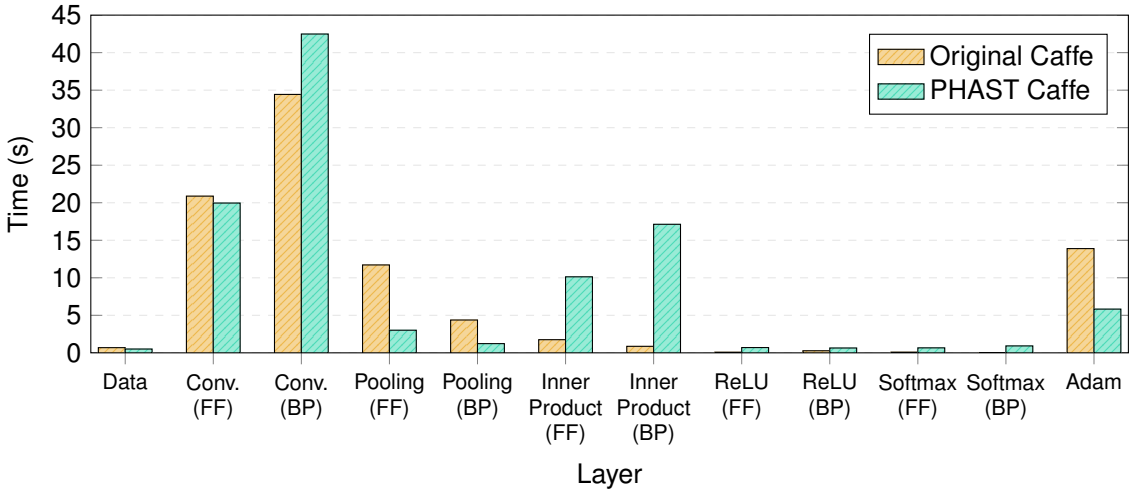


Figure 3.4: Time spent in each of the layers (MNIST) (running in CPU).

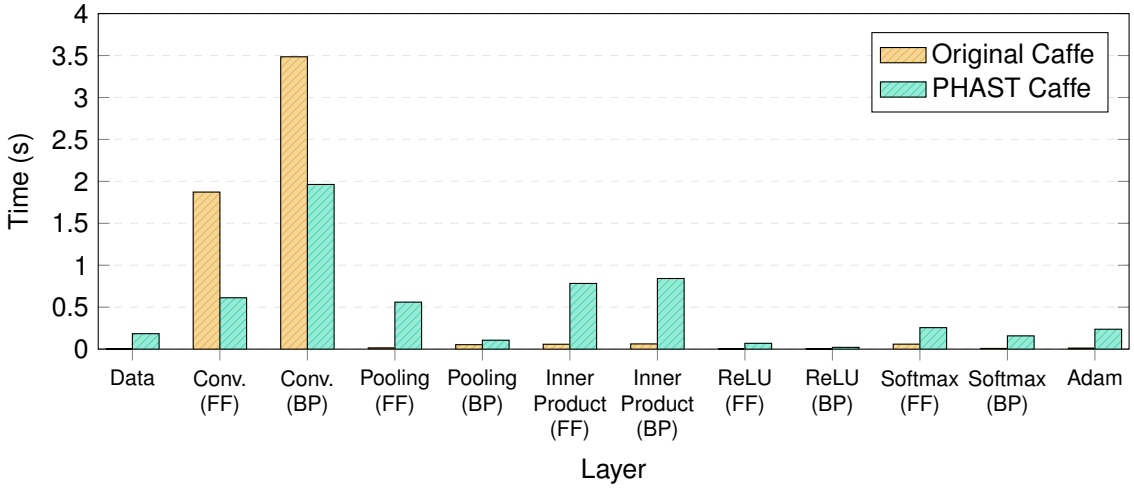


Figure 3.5: Time spent in each of the layers (MNIST) (running in GPU).

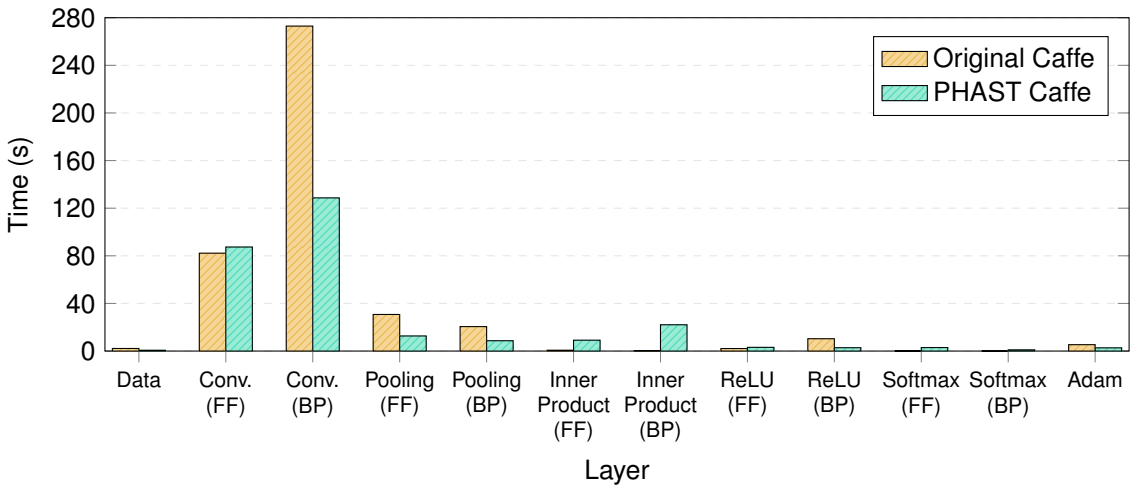


Figure 3.6: Time spent in each of the layers (CIFAR-10) (running in CPU).

On the CPU, Figures 3.4 and 3.6 show that Inner Product is the only layer where PHAST is slower than the original Caffe in both networks. It is a simple layer and, apparently, the startup harms the performance of the PHAST version. The convolution, which is the core layer of convolutional neural networks, shows different results. On the one hand, PHAST implementation is slower in backpropagation in MNIST (-34%) and in feedforward in CIFAR-10 (-6.3%). On the other hand, it is significantly faster in the backpropagation in CIFAR-10, obtaining a 2.12x speedup.

On the GPU, as depicted by Figures 3.4 and 3.6, the PHAST version is gen-

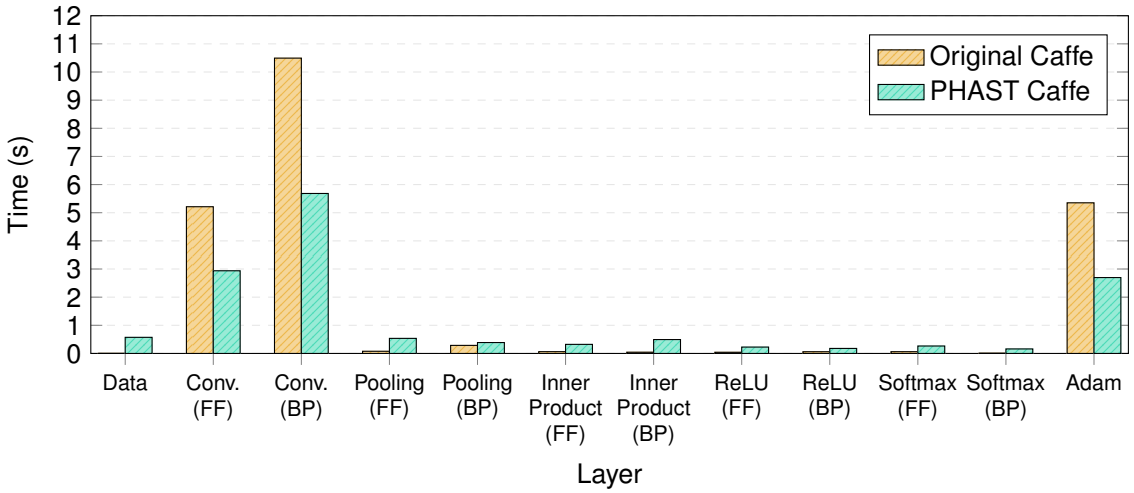


Figure 3.7: Time spent in each of the layers (CIFAR-10) (running in GPU).

Table 3.4: Performance comparison between Original Caffe and PHAST version for MNIST dataset.

	Original Caffe	PHAST Caffe
CPU	89.0	103.2
GPU	5.63	5.79

(a) Total execution time of the full network for MNIST, measured in seconds.

	Original Caffe	PHAST Caffe
CPU	1.0x	-1.15x
GPU	1.0x	-1.02x

(b) Performance gain/loss against original Caffe for MNIST. Minus means loss, plus means gain.

erally a bit slower than the original one. This is true except for the convolution layer and the Adam solver in the CIFAR-10 dataset. In MNIST, the increased speed is balanced by the other layers, while in CIFAR-10, the weight of the layers where PHAST performs better dominates the overall execution time.

Tables 3.4 and 3.5 sum up the performance of the two versions in the case of full-network execution. In the light of the results, we can conclude that the PHAST version is competitive against the original one. PHAST version is faster in CIFAR-10 on both CPU and GPU, while performs very similar in the MNIST dataset on the GPU and loses around 15% on the CPU. Looking at the execution time of each layer separately, the small layers are always a bit slower on PHAST than in the original Caffe. This difference is caused by a small overhead present in PHAST that appears each time a layer is run. This issue reduces a bit the

Table 3.5: Performance comparison between Original Caffe and PHAST version for CIFAR-10 dataset.

	Original Caffe	PHAST Caffe
CPU	852.8	562.8
GPU	21.70	14.55

(a) Total execution time of the full network for CIFAR-10, measured in seconds.

	Original Caffe	PHAST Caffe
CPU	1.0x	+1.51x
GPU	1.0x	+1.49x

(b) Performance gain/loss against original Caffe for CIFAR-10. Minus means loss, plus means gain.

performance of the small layers. On the contrary, big layers like convolution also suffer from the overhead, but the overhead is much smaller compared to the overall execution time. Therefore, the bigger the workload is, the less (or almost none) the effect the overhead has on the performance (this can be corroborated when comparing MNIST and CIFAR-10 results).

3.4.2.1.1 Measuring performance portability. Following the metric proposed by [159], we apply the formula to the data shown in Tables 3.4a and 3.5a. We choose to measure performance portability using the application efficiency, for which we use these time measurements. To do so, we compare the elapsed times for the full network simulation in the Caffe framework comparing the PHAST implementation against the native one. Data is summarized in Table 3.6.

Let Φ_M and Φ_C be the performance portability of PHAST Caffe for the MNIST and CIFAR-10 dataset, respectively. Using data from Table 3.6, we have:

Table 3.6: Performance portability metrics obtained from total execution time (application efficiency).

Dataset	Platform	Time (s)		Application Efficiency
		Achieved (PHAST)	Best (Original)	
MNIST	CPU	103.2	89.0	86.24%
	GPU	5.79	5.63	97.23%
CIFAR-10	CPU	562.8	852.8	100%
	GPU	14.55	21.70	100%

$$\Phi_M = \frac{2}{\frac{1}{0.8624} + \frac{1}{0.9723}} = 91.24\%$$
$$\Phi_C = \frac{2}{\frac{1}{1.5152} + \frac{1}{1.4914}} = 100\%$$

We achieve a near optimal performance portability for the MNIST dataset and even surpass the optimal performance portability for the CIFAR-10 dataset. Note that, even though we achieve better performance with PHAST compared to native Caffe, performance portability is expressed as 100% since PHAST is now the best implementation. This study demonstrates that our PHAST version achieves performance portability on the evaluated hardware platforms for both datasets. The best results are obtained with CIFAR-10, which is the biggest dataset.

3.4.2.2 Performance, Productivity and Portability Analysis

Peformance. According to our performance results, PHAST has proven to be good but not excellent. PHAST achieved similar performance compared to the Caffe framework, which is implemented separately using C++ and CUDA. Because Caffe is a relatively old framework, it should be relatively easy to find more efficient frameworks. Like Figure 3.8 shows, we consider that PHAST provides acceptable performance.

Productivity. Table 3.14 summarizes the results for C++ and PHAST in terms of lines of code. As we can see, PHAST proved to be much more verbose than C++. Now, speaking from our experience, PHAST has been one of the hardest

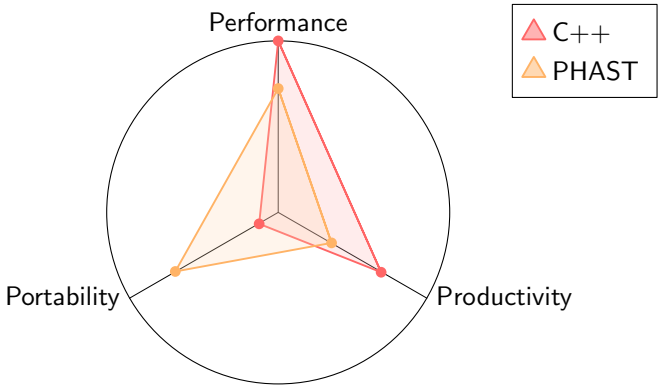


Figure 3.8: P^3 analysis of PHAST and C++.

languages regarding programming productivity, mainly because of the difficulty to achieve good performance. We give PHAST a slightly worse result in productivity compared to plain C++, since the latter is also often difficult to achieve performance with.

Portability. PHAST currently supports CPUs and NVIDIA GPUs. It is significantly more portable than C++ thanks to the support for NVIDIA GPUs.

3.4.3 oneAPI

In this Section, we show an evaluation study of the ported layers to oneAPI: softmax and convolution, comparing our single-source implementation against the original Caffe framework.

3.4.3.1 Performance

Softmax. The inputs used in the softmax layer are detailed in Table 3.7. The dimensions of the first input are the same as the dimensions of the softmax input in a LeNet layer running MNIST. Because a common workload is to run the net for 1000 iterations, we also used 1000 iterations for input 1. We intend to create a realistic input. However, because MNIST is a small data set, we design input 5, which is similar to bigger workloads. We are interested in how the layer reacts to a different number of iterations, so we create input 4 from input 5. Finally, inputs 2 and 3 illustrate the behavior of the layer in a large instance.

Table 3.8 depicts the execution times for each version of the softmax layer. Because original Caffe allows us to select the math library backend (which remarkably impacts the execution time), we decide to build a version using openBLAS (which we observed to run sequentially) and one version using MKL (which we observed to run in parallel). For the MNIST-like input, Caffe with openBLAS is the winner, and oneAPI is the loser by far. The input is tiny and the number of

Table 3.7: Different inputs for isolated softmax layer.

Input	Iterations	N	W	C
1	1000	2	6	10
2	1	2000	600	100
3	1000	2000	600	100
4	1	200	600	100
5	1000	200	600	100

Table 3.8: Execution times in seconds for isolated softmax layer (CPU).

Version	Input 1	Input 2	Input 3	Input 4	Input 5
Caffe (openBLAS)	0.001	0.709	635.000	0.071	63.051
Caffe (MKL)	0.004	0.670	406.000	0.100	40.900
oneAPI	0.474	0.309	70.000	0.150	7.395

iterations is large, so the sequential version of softmax wins against the parallel, which is incapable of benefitting from parallelism. A good point to explain the oneAPI loss is because of the overhead of the library, which is quite large compared with the actual work that needed to be done. However, oneAPI completely outperforms Caffe in bigger instances. In input 3, oneAPI achieves 5.8x and 9.0x speedup against Caffe, while the gain in input 5 remains the same. At a first glance, it does not make any sense. Figure 3.9 sheds a bit of light on this issue.

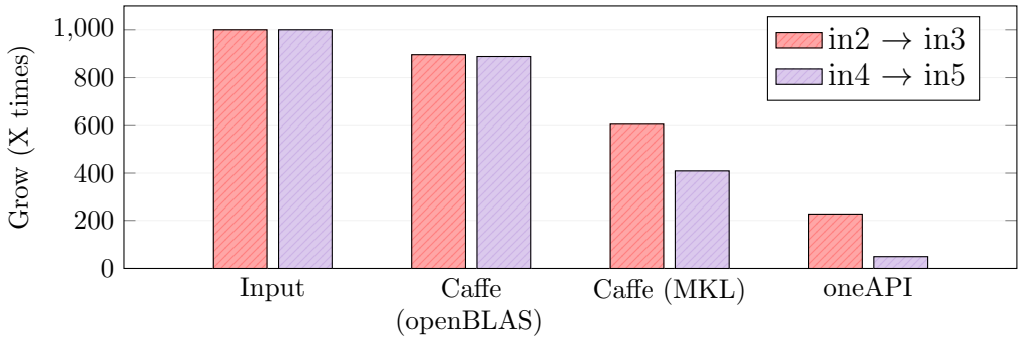


Figure 3.9: Execution time evolution from 1 to 1000 iterations in each version.

In this plot, we evaluate the execution time from one input to another. From input 2 to 3 we can see that the workload should be 1000x larger since the input is the same, but the layer runs a thousand times. Caffe with sequential math library evolution is close to 1000x, reaching around 900x. However, the parallel version using MKL is around 600x, which means that one single iteration is more expensive than one of the 1000 iterations. Finally, oneAPI drops to 200x. In the transition from input 4 to 5, the difference is even more dramatic, with a growth of 50x in the execution time. This explains the gain of oneAPI over the other versions. We found that this situation is caused by the overhead of oneAPI. For example, in input 5, oneAPI takes almost 7.4 seconds. We found that the first iteration of oneAPI takes 0.150s, while the remaining 999 iterations take $\sim$0.007s. Therefore, $0.150 + 0.007 * 999 = 7.143 \approx 7.395$. In other words, oneAPI

Table 3.9: Execution times in seconds for isolated softmax layer (GPU).

Version	Input 1	Input 2	Input 3	Input 4	Input 5
Caffe (BVLC)	0.015	0.041	7.842	0.004	0.821
oneAPI	0.336	0.397	300.000	0.040	31.83

initialization is heavy, which causes poor performance when the network is run for one iteration. However, the execution time of an iteration itself is great, so the performance in a real world scenario, running for 1000 iterations, is much better than the one offered by the original Caffe.

Table 3.9 shows the results for the softmax layer in GPU. In this case, oneAPI loses against Caffe with all the considered inputs. It happens even though the softmax layer was written in a similar way as CUDA kernels. The oneAPI version is slower for small inputs (like inputs 1 and 2) but also for big ones (like input 3), where the difference is even more dramatic. These results highlight that oneAPI support for NVIDIA GPUs is still in a very experimental stage and is thus not mature enough to be used in a real-world application.

Convolution. Table 3.10 shows the different inputs used for evaluating the convolution layer. As we did in the softmax layer, we chose the input sizes based on real world datasets:

- Input 1 represents the input size of the MNIST dataset (gray-scale 28x28 images) with 5x5 filters.
- Input 3 represents the input size of the CIFAR-10 dataset (32x32 RGB images) with 5x5 filters.
- Input 5 represents the input size of the Imagenet dataset (227x227 RGB images) with 11x11 filters.

The rest of the inputs (2, 4 and 6) represent the same dataset but with 1000 iterations, instead of one. This differentiation helps to understand the behavior of the layer when it is run a single time and when it is run many times, emulating that the layer is inside a real network. To find the input sizes, we checked Vivienne Sze's paper [183], and we ran Caffe using different datasets.

In the evaluation of the convolution layer we also included the Intel Caffe version, so we can make a fairer comparison against oneAPI. To create the isolated layer from the Intel version, we took the `conv_layer.cpp` [10]. There are two other

[10] Available at `https://github.com/intel/caffe/blob/master/src/caffe/layers/conv_layer.cpp`

Table 3.10: Input sizes for isolated convolution layer.

Input	Iterations	Image Size	Number of filters	Filters Size	Batches
1	1	28x28x1	20	5x5	100
2	1000	28x28x1	20	5x5	100
3	1	32x32x3	32	5x5	100
4	1000	32x32x3	32	5x5	100
5	1	227x227x3	96	11x11	100
6	1000	227x227x3	96	11x11	100

additional implementations of the layer, but we decided to compare this one for consistency. One of the other implementations used MKL-DNN (oneDNN now), so we should expect a similar performance. A detail to take into account when comparing the Intel convolution layer against the original BVLC one is the compiler used. Caffe BVLC uses g++ compiler by default, while Intel implementation uses icpc (Intel compiler). To be fair in the isolation layer phase, we keep the compiler in both cases, so we build each of three versions of layers (Caffe BVLC, Caffe Intel, and oneAPI) with a different compiler.

Execution times of the different convolution layers are presented in Table 3.11. In the MNIST dataset (inputs 1 and 2), we can see that Caffe BVLC is far from the other two versions. oneAPI's version is close but a bit far from Intel's version when running for 1000 iterations. This situation is reversed in the case of the CIFAR-10 dataset (inputs 3 and 4) because oneAPI achieves slightly better results than the Intel Caffe version. Moreover, BVLC Caffe competes in a lower league because the difference, in this case, is even greater. In the Imagenet dataset, the biggest one (inputs 5 and 6), oneAPI is much faster than the Intel version of Caffe, and both of them are much faster than the BVLC Caffe. Compared to the Intel optimized version of Caffe, the oneAPI implementation achieves 1.69x and 2.73x speedup in inputs 4 and 6, respectively. We can see that oneAPI is faster than the Intel version when the input size is big enough, and

Table 3.11: Execution times in seconds for isolated convolution layer (CPU).

Version	Input 1	Input 2	Input 3	Input 4	Input 5	Input 6
Caffe (BVLC) MKL	0.058	3.872	0.069	11.971	2.536	1970.1
Caffe (Intel)	0.085	0.734	0.087	1.462	0.924	300.5
oneAPI	0.063	1.122	0.053	0.862	0.267	109.8

we believe this may be due to two different facts. First, the overhead in oneAPI may be harmful when computing convolution. Second, the algorithm used in both versions is not the same (matrix multiplication in the case of Intel Caffe and direct convolution in the case of oneAPI).

As mentioned in Section 3.2.2, our convolution implementation is not supported in GPU with the current oneDNN version, so we were unable to carry out a benchmark on the GPU. Furthermore, we used the direct convolution algorithm in oneDNN. We tried Winograd convolution too, but the program crashed with an exception (`dnnl::error`: could not create a primitive descriptor iterator).

Finally, we believe that the solid performance oneAPI demonstrated in both benchmarks is strongly influenced by the hardware platform used. Our dual-socket Xeon Gold has 44 cores and 88 threads, which is a considerable level of parallelism. oneAPI seems to be well optimized for Intel parallel architectures and benefit very well from a large number of cores.

3.4.3.2 Performance, Productivity and Portability Analysis

Peformance. In our experiments, oneAPI provided excellent performance in CPU, especially using oneDNN. On the contrary, GPU performance is particularly bad, probably because due a very early state of the GPU backend in DPC++. Compared to the other approaches, oneAPI achieved similar performance portability to PHAST, and far from optimal performance due to the deficient GPU results. We show the oneAPI results in Figure 3.10.

Productivity. Like PHAST, we find that DPC++ is very verbose and its implementation is significantly larger than the implementation in C++. Our experience indicates that DPC++ has similar programming complexity as PHAST,

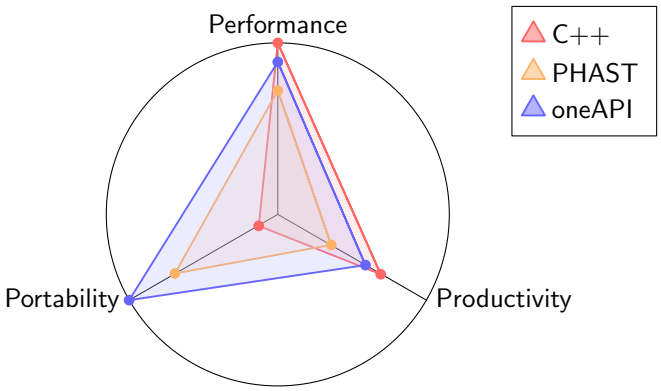


Figure 3.10: P^3 analysis of oneAPI, PHAST and C++.

while oneDNN library is easier to manage than direct programming. We also had troubles optimizing code using DPC++, which reminds us of our experience using PHAST. A good point for oneAPI's productivity is `dpct`, a tool integrated in oneAPI that can translate up to 95% of CUDA code to DPC++ automatically [37], which can be really useful when porting CUDA code to be used within oneAPI.

Portability. oneAPI supports CPUs, NVIDIA and Intel GPUs and Intel FPGAs. Intel GPU support was initially limited to integrated GPUs in Intel chips, but it was recently extended to support dedicated Intel GPUs like the Ponte Vecchio architecture [91]. Thus, oneAPI has excellent hardware portability, being significantly better portability than PHAST, since also supports Intel dedicated and integrated GPUs and also a new type of device, FPGAs. Even though we have seen that NVIDIA GPUs support is very preliminary, it has great potential for broader hardware adoption, so we consider that oneAPI achieves optimal portability. To the best of our knowledge, no language or framework has wider support than oneAPI.

3.4.4 HDNN

This Section compares the performance using the internal HDNN library against using a machine learning framework. This analysis is not meant to be a comparison of machine learning frameworks but instead to shed a bit of light on the competitiveness of HDNN concerning already existent approaches. In CPU and GPU, HDNN was compared against Caffe, and in the TPU, against PyTorch. For the first comparison, we developed custom softmax and convolution implementations using oneDNN and cuDNN. Essentially, these programs create tensors with random data and call the appropriate libraries. This way, the overhead caused by the communication between HDNN and the corresponding runtime could be measured. For the second comparison, we developed tiny testing programs using Caffe and PyTorch. The performance evaluation was accomplished using only the feedforward phase, as the current HDNN implementation does not support backpropagation.

The evaluation platform was divided into two parts. On one hand, we have the CPU and GPU hardware platform, which was equipped with a double-socket Cascade Lake Intel Xeon Gold 6238 and a Turing NVIDIA RTX 2080 Ti. On the other hand, we have the TPU hardware platform, which was equipped with a TPUv2. Hardware details for the CPU, GPU and TPU used are shown in Table 3.3 (data for the TPUv2 extracted from [94, 93]).

The evaluation was performed using simple precision data types. It is important to note that the TPU was designed to work in half precision, so its potential is reduced in this scenario, compared to the CPU and the GPU. We leave for future work the support of half precision workloads which would make much better use of TPUs and other machine learning accelerators.

For the performance evaluation, we measured the execution time of each implementation (given in seconds), taking into account only computation times. Each experiment was repeated 5 times, and the values shown are the average over these 5 independent runs. For the softmax layer, we run each input during 1000 iterations and for the convolution, we run each input during 100 iterations.

3.4.4.1 Performance

Softmax. For the softmax layer, we designed 4 input sizes expressed in the triple (N,C,W), where N is meant to be the number of classifications that softmax needs to calculate, C is the number of channels, and W is the width of the vector containing the softmax data. Input 1 is $(2, 10, 6)$, input 2 is $(200, 100, 600)$, input 3 is $(2000, 100, 600)$ and input 4 is $(200, 100, 6000)$.

Performance results are shown in Table 3.12. It is worth noting that Caffe was compiled using openBLAS since it achieved better results than MKL[11]. In cuDNN, the `CUDNN_SOFTMAX_ACCURATE` algorithm was used (both for HDNN and the cuDNN test program) to avoid possible overflows when computing the softmax. When we compared the performance achieved by HDNN against the optimized libraries (oneDNN and cuDNN), we found that HDNN achieves the same performance as using the libraries directly. Therefore, we omitted the results in the tables for brevity. As we theorized before, we can conclude that HDNN suffers no overhead when communicating to the specialized backends.

Except for the first input, HDNN achieves speedup near or bigger than 5x compared with Caffe in CPU. The performance degradation in the first input comes from the fact that this input is tiny. For bigger workloads, HDNN consistently outperforms Caffe. In GPU, HDNN is around 1.5x faster than Caffe for all input except the first, reaching 15x. Finally, the results in TPU are very similar to the ones obtained by PyTorch.

Convolution. For the convolution layer, we designed a set of inputs based on real neural networks, for which we used the Sze et al. survey [183]. In this evaluation, we set the number of batches to 100 for all inputs. Inputs 1 and 2 are representative of the MNIST dataset. Although the MNIST dataset images

[11]MKL outperformed openBLAS for all the operations needed in the softmax layer except for the division, which ran very slow compared to openBLAS.

Table 3.12: Execution time of the softmax layer in CPU, GPU and TPU (in seconds).

Input	CPU			GPU			TPU		
	HDNN	Caffe (openBLAS)	Speedup	HDNN	Caffe (cuDNN)	Speedup	HDNN	Pytorch	Speedup
1	0.237	0.001	0.01x	0.001	0.015	15.0x	0.990	0.950	0.96x
2	33.32	151.88	4.55x	0.562	0.821	1.46x	1.220	1.248	1.02x
3	263.4	1529.1	5.80x	5.123	7.482	1.46x	31.69	31.88	1.06x
4	333.1	1554.5	4.66x	4.482	7.727	1.59x	30.31	30.25	0.99x

Table 3.13: Execution time of the convolution layer in CPU, GPU and TPU (in seconds).

Input	CPU			GPU			TPU		
	HDNN	Caffe (MKL)	Speedup	HDNN	Caffe (cuDNN)	Speedup	HDNN	Pytorch	Speedup
1	0.101	0.505	5.01x	0.004	0.132	30.0x	1.016	1.063	1.05x
2	0.145	1.029	7.08x	0.012	0.162	13.2x	0.902	0.944	1.05x
3	0.145	1.029	7.08x	1.288	1.720	1.33x	2.226	2.227	1.02x
4	7.091	9.561	1.34x	1.550	2.094	1.35x	2.001	2.018	1.01x
5	7.898	13.722	1.73x	4.158	5.801	1.39x	5.130	5.229	1.02x
6	14.428	39.663	2.74x	1.157	2.120	1.83x	3.102	3.100	1.00x

are gray-scale, we also tried using color images (3 channels instead of 1). Thus, the size of input 1 is 28x28x1, and input 2 is 28x28x3. Both have 5x5 filters, but input 1 has 20 filters, and input 2 has 50. Inputs 3, 4, and 5 represent AlexNet networks (input sizes 227x227x3 with 96 filters, and filter sizes of 3x3, 5x5, 11x11, respectively). Input 6 is based on ResNet (input size of 224x224x3, with 64 7x7 filters).

We used the direct convolution algorithm in oneDNN (CPU), the only available algorithm that worked. In cuDNN, we used `cudnnGetConvolutionForwardAlgorithm` to automatically get the fastest algorithm for the convolution in each case. We used this approach for HDNN and the cuDNN test program. In HDNN, the best algorithm is queried once and is used for all the iterations. According to this function, the best algorithm was `IMPLICIT_GEMM` for input 1, `IMPLICIT_PRECOMP_GEMM` for inputs 2,3,4 and 6, and `FFT_TILING` for input 5[12].

Convolution performance results are shown in Table 3.13. We evaluated the performance using oneDNN and cuDNN directly, and we did not appreciate any

[12]The full name of the algorithms (which always starts with `CUDNN_CONVOLUTION_FWD_ALGO_`) was omitted for brevity.

negative impact on the performance in this case either, so we also omitted the results of oneDNN and cuDNN. In both the CPU and the CPU, HDNN achieves significant enhancements against Caffe, which highlights the fact that HDNN is competitive, thanks to the use of optimized libraries. As happened with the softmax layer, HDNN performs similarly to PyTorch. Therefore, we believe that the TPU distributed algorithm implemented in the TPU runtime and the use of PyTorch optimized primitives effectively take advantage of the full power of the TPU.

In the light of the results, we can conclude that HDNN achieves competitive results against other machine learning frameworks. Even though a given HDNN program is only written once, it can be targeted to different hardware devices without any changes, and it also provides solid performance results.

3.4.4.2 Performance, Productivity and Portability Analysis

Performance. HDNN does not compile code directly to the device, but rather relies on optimized libraries for heavy computations. Specifically, HDNN uses oneDNN, cuDNN and PyTorch under the hood for CPUs, GPUs and TPUs, respectively. Vendor optimized libraries often provide the best performance possible. Because the HDNN environment based on MLIR has little to no overhead, we consider that HDNN performance is near optimal. Figure 3.11 shows the final verdict of P^3, including all evaluated languages.

Productivity. For evaluating HDNN productivity, we also included DeepDSL [204], a DSL for deep learning, for fair comparison against HDNN. The source code for convolution and softmax is shown in Figure 3.12. HDNN and DeepDSL needed similar lines to implement both layers, followed by C++.

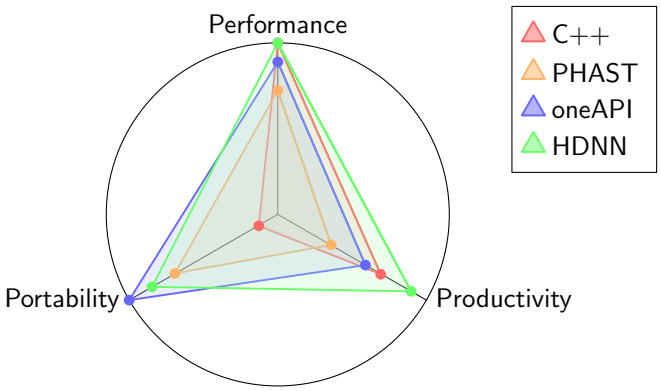


Figure 3.11: P^3 analysis of HDNN, oneAPI, PHAST and C++.

```
val K = 10 // # of classes
val N = 500;
val C = 1;
val N1 = 28;
val N2 = 28 // batch size, channel, and x/y size
val y = Vec._new(Mnist, "label", "Y", N) // labels
val x = Vec._new(Mnist, "image", "X", N, C, N1, N2) // images
val cv1 = CudaLayer.convolv("cv1", 5, 20) // kernel size
val network = f2 o relu o f o flat o mp o cv2 o mp o cv1
val x1 = x.asCuda // load x to GPU
val y1 = y.asIndicator(K).asCuda
val loop = Loop(c, p, (x, y), param, solver)
cudnn_gen.print(loop)
```

```
val K = 10 // # of classes
val N = 500;
val C = 1;
val N1 = 28;
val N2 = 28; // batch size, channel, and x/y size
val y = Vec._new(Mnist, "label", "Y", N) // labels
val x = Vec._new(Mnist, "image", "X", N, C, N1, N2) // images
val softmax = CudaLayer.softmax // softmax
val network = f2 o relu o f o flat o mp o cv2 o mp o cv1
val x1 = x.asCuda // load x to GPU
val y1 = y.asIndicator(K).asCuda
val loop = Loop(c, p, (x, y), param, solver)
cudnn_gen.print(loop)
```

Figure 3.12: DeepDSL convolution (top) and softmax (bottom) programs used in the evaluation.

Results also emphasize that HDNN source code is straightforward, so good productivity can be achieved with it, in the line of existent DSL. In terms of programming difficulty, HDNN is very easy to use and has very short development times. HDNN, like all DSLs, tradeoff productivity for applicability; they are targeted for narrow domains and can be hardly used for other applications. Even though HDNN has proved to be more productive than its competitors, we do not consider that HDNN's productivity is optimal. For existent code, rewriting is necessary if the developer want to use accelerators. This fact harms productivity.

Portability. HDNN supports CPUs, NVIDIA GPUs and TPUs. In other words, it supports the major machine learning hardware devices but lack support for other more general accelerators like FPGAs. Therefore, we consider that HDNN

portability is slightly behind oneAPI's, which proved to be the best in our evaluation.

3.4.5 Overall

3.4.5.1 Portability

C++ is clearly the less portable language in our analysis, as is a traditional programming language that only compiles for CPU. Regarding single-source languages, PHAST is the next, currently supporting CPUs and NVIDIA GPUs. HDNN adds also TPUs to the list compared to PHAST. And lastly, oneAPI supports CPUs, NVIDIA and Intel GPUs and Intel FPGAs. HDNN supports CPUs, NVIDIA GPUs and TPUs. oneAPI and HDNN have similar coverage but we considered HDNN to be slightly worse because the domain is narrower.

3.4.5.2 Productivity

Table 3.14 summarizes the source lines of code needed by different languages to implement softmax and convolution. PHAST and oneAPI are clearly the most verbose languages, needing more than the double amount of lines compared to C++, which is the next one in the list. It is worth noting that the C++ Caffe implementation of the convolution relies on GEMM functions to perform the computations, hiding much of the complexity inside the functions. As we anticipated, DSLs achieve the shortest programs; HDNN and DeepDSL are significantly shorter than C++ programs. Speaking now about programming experience, DPC++ (oneAPI) and PHAST are the hardest approaches, specially when trying to achieve performance portability. They are closely followed by C++ and far from the DSLs. Not surprisingly, our programming experience matches the results derived from program length.

3.4.5.3 Performance

High-performance computing code is typically written in C/C++, which allows very efficient code in CPU. HDNN achieves similar performance because re-

Table 3.14: Source lines of code (SLOC) measured in different languages.

Layer	HDNN	DeepDSL	oneAPI	PHAST	C++
Softmax	7	13	60	58	28
Convolution	9	13	78	57	17

lies on optimized libraries (where CPU optimized libraries are typically implemented using C++). Thus, we consider that both are able to achieve the highest performance. They are followed by oneAPI, which offers a competitive environment where common workloads can be dispatched using oneDNN, and when they are not implemented the developer can use DPC++. Lastly we find PHAST, which oftentimes use optimized libraries under the hood for the computations, but sometimes data is managed with handcrafted code, and some other algorithms are implemented by hand.

3.5 Related Work

Since heterogeneous hardware is steadily growing, many new single-source languages are emerging. There are many works that study the existent languages [9] and ways to classify them [2].

Works with general purpose single-source languages. In addition to PHAST [153], some examples are OmpSs [53], Kokkos [55] and SYCL [99] based implementations. SYCL is a standard, just like OpenCL, that allows the design for specific languages. Some examples of implementations of the SYCL standard are Intel's oneAPI [87], Codeplay's ComputeCpp [172], or Heidelberg University's hipSYCL [5]. SYCL implementations are usually based on the LLVM stack and thus make extensive use of intermediate representation (IR). In addition to the mentioned techniques, HPVM [104, 56] is a very relevant compiler framework based on dataflow graphs. HPVM introduces a hardware-agnostic parallel intermediate representation (IR) that allows the compilation of different devices. Instead of programming at the IR level, like in HDNN, HPVM provides a custom high-level language, HeteroC++. In essence, it is a C++ extension that allows programming code easily for using the HPVM infrastructure. Like HDNN, HPVM uses a runtime system and IR, also featuring a virtual instruction set architecture (ISA). HPVM also supports Keras and PyTorch as frontends [56], and features many hardware backends, supporting CPUs, GPUs, FPGAs, fixed function accelerators and machine learning accelerators.

From the mentioned languages, oneAPI has shown promising results in various computer fields, such as machine learning [70] or decision making [35]. Recent works have also studied the use of `dpct`, a tool integrated into the oneAPI environment used to migrate CUDA programs to DPC++, showing that the migrated DPC++ code reports similar efficiency compared to the CUDA implementations [36]. In [37], authors performed a similar study to the one we have presented when porting Caffe using oneAPI, but applied to bioinformatics, also

showing that oneAPI can handle heterogeneous programming with small or none performance degradation. An SYCL-based solution was proposed for Fast Fourier Transform (FFT) in [151], where authors also reach good performance portability results.

Works with Domain-Specific Languages. In machine learning, there are many examples of compiler-based solutions that aim to provide performance on heterogeneous hardware, such as Glow [169] or MLIR [106, 107], which is currently being used in Tensorflow. MLIR [106] is a compiler infrastructure also based on LLVM, which is intended to be used in heterogeneous systems. One of the targets of MLIR is to be used in machine learning (Tensorflow [1] is currently using it). MLIR was released recently but it already has had a significant impact since many projects use it in many fields. In addition to our proposal for deep learning [118], a DSL based on intermediate representation, DeepDSL [204] has been proposed. Other uses in MLIR comprise image processing [75], quantum computing [123], or polyhedral compilation [102]. Other example of DSL in deep learning is TVM [28], a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware backends.

3.6 Conclusions

In this Chapter, we carried out a detailed analysis of programming languages in heterogeneous environments. By applying oneAPI and PHAST to Caffe, we explored how difficult is to achieve performance portability with such languages in real-world scenarios. Historically, to get good performance, a good understanding of the underlying hardware is mandatory. Most of the hard work of abstraction and performance portability must be done on the library side, but as we have shown, the programmer has many duties that can impact the final result. We can conclude that achieving performance portability using PHAST and oneAPI is possible, but also that doing so is very hard from the programmer standpoint. Sometimes changes to achieve performance portability are needed in the language backend, as we have shown in the PHAST reimplementation of Caffe. Also, the programming effort is higher than in traditional languages. As we have shown, oneAPI and PHAST are typically more verbose than C++, which increases the complexity of software development. Thus, we proposed HDNN, a deep learning MLIR dialect for heterogeneous computing. HDNN provides a unified interface to run softmax and convolution layers, executed on x86_64 CPUs, NVIDIA GPUs and Google TPUs.

PHAST. The PHAST single-source implementation of Caffe provides similar performance results compared to the native CPU and GPU counterparts. With this, we provide proofs to evidence that performance portability with PHAST is possible. If the Caffe framework had to be created from scratch, the PHAST library would have provided a way to develop the framework with much less programming effort. With this approach, maintenance and bug fixing is more straightforward, as there is a single-source code base instead of two. However, achieving performance portability was a very hard task. Good performance results are brittle because require deep understanding of the original algorithms. To complicate things further, part of the work must be done at the internals of PHAST, which can only be assessed by PHAST developers.

oneAPI. Regarding oneAPI's strong points, we want to emphasize the CPU performance and programming interface. In the evaluation, we found a very mature CPU backend that can challenge and even outperform native alternatives. In the programmability aspect, oneAPI may be difficult because the programmer needs to learn new concepts related to the SYCL environment. However, programmers familiar with CUDA or OpenCL interfaces may learn them quickly because of their similarities with SYCL. On the other hand, two of the weakest points in oneAPI we have found are the build process and the support for NVIDIA GPUs, which is in a very early stage. We found easier to achieve competitive performance compared to PHAST. Still, productivity can be further improved because DPC++ is generally too verbose.

HDNN. Using HDNN, developers can employ a domain-specific language to target domain-specific hardware. This is an opposite approach of a common idea in heterogeneous computing; trying to compile a unique code efficiently for multiple platforms. With HDNN we have shown a novel approach to the P^3 problem. Overall, HDNN provides *portability*, as its source code is written once and can be targeted to three devices (CPUs, GPUs and TPUs). It also provides good *productivity* because it is focused on the DNNs domain, making development easier. And it also provides *performance*, thanks to the use of optimized libraries for the heavy, performance-critical workloads.On the other hand, being a DSL also limits the usability of HDNN because it is only useful for the specific domain of DNNs. Even though it is an excellent candidate for achieving good *portability*, *productivity* and *performance*, it is not applicable in all the use cases.

A step forward in productivity. Despite advances in heterogeneous programming, productivity can still be improved significantly. The code volume that is already written is way larger than the new code that is being programmed

every day. Accelerating old code would require rewriting large code bases with languages like the ones we have studied. New languages are indeed improving programmers' productivity, even more in the case of DSLs. However, rewriting code is inefficient from a productivity standpoint, costing innumerable development hours and money. Ideally, we would like to find alternative approaches to accelerate legacy codes without rewriting. Automatically accelerating old codes would improve productivity to the point of not needing to re-develop software again. We believe this would achieve "optimal" productivity as represented in Figure 3.11, while easily maintaining the other features of performance and portability.

Chapter 4

Compiling Existent Code to Accelerators

4.1 Introduction

4.1.1 Motivation

New generation single-source programming languages [87, 153, 55, 177] improve productivity [195] by targeting many hardware devices with a single, hardware agnostic code. However, these approaches need writing code from scratch, or porting existent code with new languages [120]. Therefore, to accelerate existing code, the traditional but inefficient approach is to rewrite it. The combined importance of linear algebra and tensor computations, as well as the difficulty of rewriting legacy code to accelerators has led to recent work which attempts to automate the process. Automating the replacement of code with calls to optimized APIs can radically improve productivity in heterogeneous environment for existent code, since developers would not need to rewrite code anymore but use a compiler that does it automatically. In this context, the importance of linear algebra is reflected in the large number of accelerator libraries and hardware devices devoted to fast linear algebra. It is indeed the building block of many of today's critical applications; from weather modeling [31] to ubiquitous DNN [48] workloads. Accelerator range from specialized devices such as Google's TPU [95] to the tensor cores on NVIDIA [29] among many others [86, 15, 93, 12, 62]. While such devices promise significant performance for an important class of applications [44], their uptake is limited by their pro-

grammability [51]. Typically, these accelerators and libraries are accessed via calls to specialized APIs, meaning existing code has to be rewritten. Given the volume [96] and variety [113] of existing legacy code, such rewriting is a significant undertaking [44].

IDL [68], KernelFaRer [46] and Polly [73] aim to compile existing code (typically, C/C++ code) to accelerators automatically. These techniques search user code for matrix multiplications using constraints [68, 46] or polyhedral analyses [21] and replace regions of code with appropriate API calls or instructions. However, as we show in Section 4.3.4.1, these approaches are fragile. Constraints capture only a limited set of program patterns and small variations in the user code defeat them. While they work well on curated benchmarks, they perform poorly on real-world code [46, 196], defeated by function calls, optimized code and inline assembler. To illustrate the problem, consider the code in Figure 4.1. It shows a straightforward matrix multiplication program fragment, from the parboil benchmark suite [179]. The aforementioned approaches aim to detect this matrix multiplication and replace it with a call to the library, shown at the bottom of the diagram. To replace code with an API call they have to both detect the code performing a matrix multiplication and also determine which user program variables correspond to the arguments of the API call. IDL, KernelFaRer and Polly are able to detect that this is a matrix multiplication, and can determine the mapping between user variables and API parameters. Unfortunately, in practice, user code can be complex such that code structure or pattern-based approaches inevitably fail. As an example, consider the code found on GitHub shown in Figure 4.2, which implements a matrix multiplication algorithm (only a fragment of the 120 lines of user code are shown here). The code structure is complex and difficult to understand as it makes extensive use of inline assembler intrinsics which defeats the code structure analysis approaches of IDL, KernelFaRer and Polly, preventing acceleration.

On the other hand, neural classification techniques (e.g. [39]) can effectively detect code despite these challenges. However, it does not provide a path to acceleration, but requires further steps, since detection is not enough for automatic acceleration. These steps include generating variable mappings and checking for equivalence [196] which has shown promising results for Fourier Transforms. However, one of the key challenges in matching code to APIs is the cost of searching for user program variables that map to API formal parameters. As the width of the API and complexity of the user program increase, this becomes combinatorially expensive. As we show in Section 4.3.4.4 existing approaches [196] fail to scale to the challenges that linear algebra APIs present. Critically, it fails to handle the large search space of mappings within General Matrix Mutiplication (GEMM) and tensor code.

Handmade GEMM kernel (CPU)

```
if ((transa != 'N') && (transa != 'n')) {
  std::cerr << "unsupported" << std::endl;
  return;
}

if ((transb != 'T') && (transb != 't')) {
  std::cerr << "unsupported" << std::endl;
  return;
}

for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b
    }
    C[mm+nn*ldc] = C[mm+nn*ldc] * beta +
                   alpha * c;
  }
}
```

```
gemm(A, B, C, alpha, beta, m, n, k, lda, ldb, ldc);
```

Call to optimized GEMM API (XPU)

Figure 4.1: Easy API replacement example. Figure shows the program, taken from the parboil benchmark, a widely-used benchmark suite, and how is transformed into a call to an optimized GEMM accelerator API.

Handmade GEMM kernel (CPU)

```
assert(M % BLOCK_M == 0);
assert(N % BLOCK_N == 0);
assert(K % BLOCK_K == 0);

__m256 vab00 = _mm256_setzero_ps();
...

for (int k = 0; k < K; k++) {
  float pa = &A[lda * (k + i) + 0];
  float pb = &B[ldb * (k + i) + 0];

  __m256 vb0 = _mm256_load_ps(pb + 8 * 0);
  __m256 vb1 = _mm256_load_ps(pb + 8 * 1);

  __m256 va0 = _mm256_broadcast_ss(&pa[8*i+0]);
  __m256 va1 = _mm256_broadcast_ss(&pa[8*i+1]);
  ...

  vab00 = _mm256_fmadd_ps(va0, vb0, vab00);
  vab01 = _mm256_fmadd_ps(va0, vb1, vab01);
  ...
}

__m256 vc00 = _mm256_load_ps(C + ldc * 8);
...
vc00 = _mm256_add_ps(vc00, vab00);
...
_mm256_store_ps(C + ldc * 0 + 8 * 0, vc00);
```

```
gemm(false, false, A, B, C, 1., 0., m, n, k, lda, ldb, ldc);
```

Call to optimized GEMM API (XPU)

Figure 4.2: Hard API replacement example. Figure shows the program, taken from GitHub, consisting of 120 lines of hand-optimized intrinsics for AVX2, and how ATC matches the code to the accelerator API.

4.1.2 Research Context

We present Algebra and Tensor Compiler (ATC), a compiler that applies program synthesis [74] to match and replace general user code to specialized APIs, which ultimately translates into highly-tuned CPU code or hardware accelerators (source code, as well as the artifact for reproducing the results in this study, is available at Zenodo [121]). We identify and solve key challenges enabling the detect/synthesize paradigm to scale to the more complex APIs of linear algebra acceleration. In addition, ATC employs a trained platform predictor to determine whether acceleration is profitable or not. We applied our approach to 50 GitHub GEMM and 15 convolution projects and discovered between 2.6 and 7x more linear operators compared to KernelFaRer [46], IDL [68], Polly [73]

or FACC [196]. This resulted in more than an order of magnitude performance improvement. Our research makes the following contributions:

- We present ATC, which maps matrix multiplication and convolution programs to specialized APIs, up to 7x more frequently than existing techniques.
- We introduce novel heuristics to reduce the mapping search space by four orders of magnitude.
- We develop novel dynamic analyses to determine higher-level information about variables, enabling synthesis without costly whole-program analyses.

Rather than relying on code structure to guide detection, ATC uses program synthesis and behavioral equivalence to determine if a section of code is a linear algebra operation. Firstly, ATC uses neural program classification [39] to detect that the code in Figure 4.2 is probably a GEMM. It then searches variable matches to determine the potential source and output arrays. As the search space is combinatorially large, we introduce scalable, algorithm-independent heuristics (which we discuss in Section 4.2.3) that keep the number of mappings manageable. Next, ATC generates different input values for the arrays and records the output. After generating many randomized inputs, it observes that it has the equivalent behavior to the corresponding API and is able to replace the AVX2 code with the GEMM call at the bottom of Figure 4.2.

Legality. Now, IO behavioral equivalence is not proof that a section of code is a particular linear algebra operation - similarly IDL and KernelFaRer do not prove equivalence. For proof, bounded model checking based on Kleene [32] can be deployed. In practice, as demonstrated in our experimental section, IO equivalence gives no false positives. For further guarantees, we can ask for programmer sign-off or employ model checking. Furthermore, emerging accelerators are now adopting low-precision operators [183] such as BF16, FP16 or INT8. ATC can easily cover corner cases in IO testing like these low-precision issues by comparing the precision of the user code and the accelerator API, triggering a warning where applicable.

Profitable. Once we have detected and can replace a section of code with an accelerator call, we need to determine if it is profitable to do. Due to hardware evolution, we do not use a hard-wired heuristic to determine profitability. Instead, we learn, off-line, a simple predictive model to determine if the target

accelerator is faster than a CPU implementation. The model is called at runtime, determining if offloading is worthwhile.

FACC. Behavioral equivalence is also employed in FACC [196]. Unfortunately, it is restricted to FFTs and one dimensional arrays, and cannot detect the replacement in Figure 4.1. Therefore, we extended FACC to FACC* to consider GEMMs and multi-dimensional arrays. This, however, exposes its weak variable binding model which is combinatorial in the number of user array variables and their dimensionality. Furthermore, it relies on program synthesis to determine the length of arrays, which scales poorly to problems with many potential length parameters for arrays such as GEMM. FACC also relies on brittle interprocedural liveness analyses to determine the liveness status of variables. This restricts it to running only at link time, rendering it invalid for use in shared libraries. We will see in Section 4.3 that the combination of these issues results in excessively large search spaces.

4.2 Matching Linear Algebra and Tensor Code to Accelerators

4.2.1 System Overview

Figure 4.3 gives a system flow overview of ATC, showing how the combination of different components enables detection of linear algebra within user code and their replacement with accelerator calls. Those components based on prior work are colored blue while new contributions are colored green. We first detect regions of code that are likely to be linear algebraic operations using a neural program classifier. The classifier is trained ahead of time, based on example programs of linear algebra code. Once candidate code sections have been identified, we apply program analysis to match user program variables with the particular API formal parameters. Given the combinatorially large search space, we develop novel techniques to make the problem tractable. For each candidate matching, we generate multiple data inputs, execute the user code section and record the output values. If the input/output pairs correspond to the input/output behavior of the accelerator API, we can say they are behavioral equivalent and candidates for replacement. While candidate user code may be replaceable with a call to an accelerator API, it may not be profitable. Therefore, we employ a simple machine learning classifier, trained offline and invoked at runtime to see if acceleration is appropriate for the user code for the runtime known array sizes.

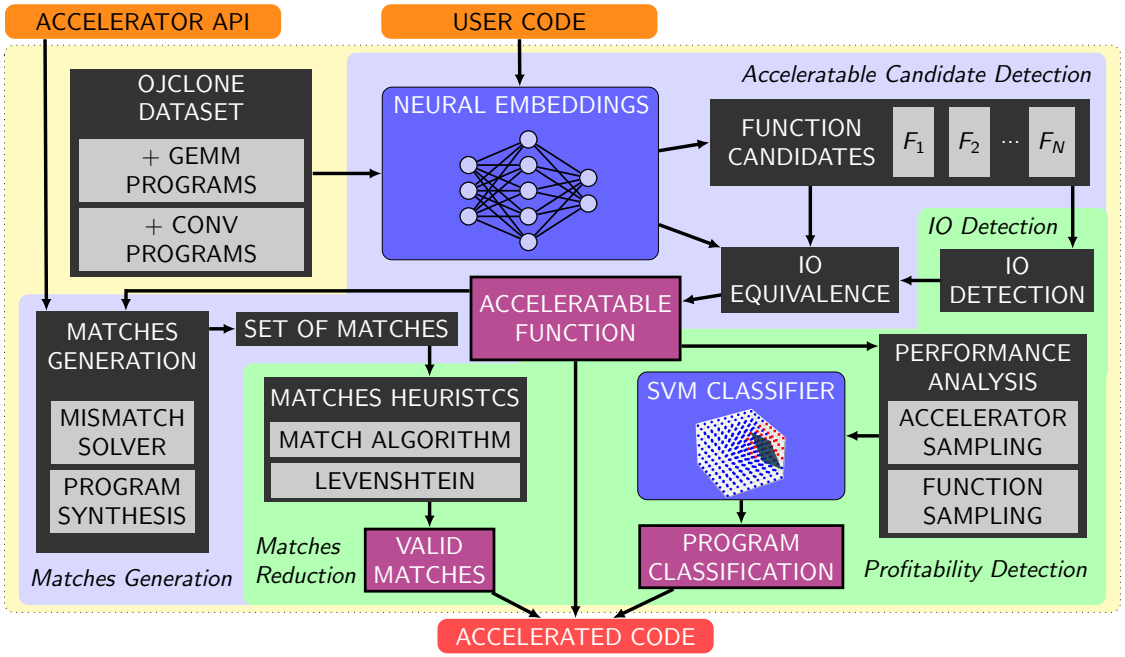


Figure 4.3: ATC compiler architecture.

To detect potentially acceleratable parts of a program, we use prior work in neural program classification [39]. A network is trained with multiple instances of different program classes. We use the OJClone dataset [129], which includes 105 classes of different programs, and add examples of the programs that we want to detect *e.g.* GEMMs and convolutions, gathered from benchmark suite repositories other than GitHub. At compile time, a new candidate program is divided into functions, which are presented to the neural classifier. The classifier assigns each function in the program a probability of belonging to a certain class. We consider the most probable class, which in the case of a GEMM or convolution is then considered for variable matching and eventual code replacement as described in the following sections. Classification is fast ($\leq$ 1.5 sec) and has negligible impact on compilation time (see Section 4.3.4.4).

4.2.2 Variable Matching

To check if a section of user code is behaviorally equivalent to the API, we have to match up the user program variables with API formal parameters. We first detect what variables are livein/liveout (Section 4.2.2.1) and then the dimensions of arrays (Section 4.2.2.2).

Algorithm 1 Livein/Liveout detection algorithm.

```
1: for ptr in funpointers do
2:     arr = generateRandomArray(ptr)
3:     hash = computeHash(arr)
4:     Add arr to A
5:     Add hash to H
6: end for
7:
8: V = generateVariables
9: ffi_call(A, V)                          ▷ Calls the corresponding function
10:
11: for arr in A do
12:     newhash = computeHash(arr)
13:     if newhash = hash then
14:         Add arr to Livein
15:     else
16:         Add arr to Liveout
17:     end if
18: end for
19: return Livein, Liveout
```

4.2.2.1 Detecting Livein and Liveout Variables

To find if two codes are behaviorally equivalent, ATC needs to know which variables are livein and liveout in order to execute the code with the appropriate input and output variables. Detecting livein and liveout variables via standard static analysis is straightforward for well-structured programs but fails for more diverse real-world codes, which may use assembly code or intrinsic functions. ATC uses dynamic analysis to determine which variables are livein and liveouts inside a function. In C, variables are passed by value, so non-pointers variables are always livein. In the case of pointers (or arrays), the compiler generates random inputs with arbitrary sizes, computing the hash of the data to quickly check value equality. Afterwards, it automatically detects the function signature and runs the function using the foreign function interface (FFI) mechanism using `libffi` [72]. After running the function, the hashes are computed again and compared with the original ones. If the values in memory change after executing the program, the array is considered liveout. This allows us to detect which variables are livein or liveout, but not those that are both livein and liveout at the same time. Therefore, the compiler saves the hashes of the outputs and

executes the function again, changing the input data of the liveout variables. It generates again a new random input for liveout variables and re-executes the function. After running the function for the second time, it compares the hash of the first execution with the second one. If hashes matches, we can conclude that the variable is just liveout, whereas if the hashes differ we know that the variable is also livein. The livein/liveout detection algorithm is shown in Algorithm 1. We implement it as a just-in-time compiler pass in LLVM [105].

4.2.2.2 Detecting the Dimensions of Arrays

Detecting arrays length enables offloading of appropriately-sized regions of codes, so it is a critical step in ATC. For some programs, lengths can be found using static analysis (e.g., [164]), but this fails in more complex cases. We use runtime analysis to determine which program variables define array size using a modified form of runtime array bound checking. For each set of variables that could define an array's size (typically, from the argument list), we set such variables to a fixed value. For example, in 2D array detection, two variables are set to an small value (those variables are the guess of the array dimensions) and the rest are set to a big value. We then execute the user code, which is modified to check runtime array accesses.

First, the compiler selects a target array to find its size. Then, to generate the modified program, the compiler tweaks the load and store instructions in the user program, replacing them with custom function calls in the IR. If a load or store does not access the array that is being analyzed, the compiler modifies it to load/store at/from a constant, safe location. If it does, the instruction is replaced with a function call that will check at runtime if the access is out of bounds. The program is executed and the compiler checks if the program failed or not. If the guess of the variables that define the array size is wrong, a load or store instruction will access an illegal memory address, which will be cached by the custom function in the IR. In such case, the program exits with a custom

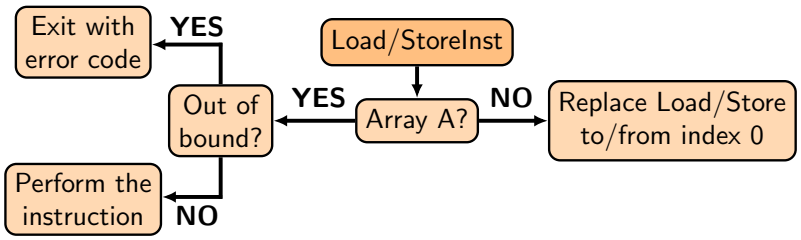


Figure 4.4: Dimension detection algorithm overview for a target example array called A.

Algorithm 2 Dimensions detection algorithm.

```
1: for arr in function do
2:     fakeLoadAndStoresExcept(arr)
3:     replaceLoadAndStores(arr)
4:     repeat
5:         c = getNextCombination(arr)
6:         ffi_call(A, V)
7:         if not failed then
8:             found = True
9:         end if
10:    until not found
11:    Add c to C
12: end for
13: return C
```

error code. If the guess is correct, the program will complete successfully, which indicates the compiler that the chosen variables are indeed the ones that define the array size. The basic idea is depicted in Figure 4.4. This is used by our JIT analysis as shown in Algorithm 2 and implemented in LLVM.

This way, the compiler can assign different input sizes to a given array and check the exit code. Therefore, the compiler iterates over all the possible dimensions combinations until one of the executions does not end with the custom error exit code. That means that the program was completed without any illegal access to the target array, which indicates that it is the right dimension of the array.

4.2.3 Reducing the Matchings Search Space

To match code to APIs, the compiler generates different candidates for the variable to formal parameter mappings and then tests them using IO equivalence. For small APIs, all mappings can be explored, but the combinatorial cost makes it prohibitive for real-world accelerator APIs. We develop techniques that reduce the mapping space by exploiting arrays information and human coding styles.

4.2.3.1 Exploiting Array Information

Using array dimensions (detailed in Section 4.2.2.2), we can reduce the number of possible matches that must be checked, as assigning one array to another means that the dimensions of each array must line up. We first generate all

Algorithm 3 Automatic matching algorithm.

```
1: function dimsMatch(f1a, f2a, p, n)
2:     S = ∅
3:     idx ← 0
4:     for args1 in f1a do
5:         args2 = f2a[p[idx]]
6:         Add {args1, args2} to S
7:         idx ← idx + 1
8:     end for
9:     return Size(S) = n
10: end function
11:
12: function outMatch(f1o, f2o, p)
13:     idx = IndexOf(f2o, 1)
14:     return IndexOf(p, idx) = IndexOf(f1o, 1)
15: end function
16:
17: function findMatchings(f1a, f2a, f1o, f2o, n)
18:     B = ∅
19:     for p in permutations(0...n) do
20:         if dimsMatch(f1a, f2a, p) and
21:             outMatch(f1o, f2o, p) then
22:             Add p to B
23:         end if
24:     end for
25:     return B
26: end function
```

$n!$ permutations of the n array variables to n parameters mapping. We discard all permutations where variable livenesses do not match (e.g., a livein variable cannot be mapped to a liveout variable). Then, for each candidate user array and parameter array pair, we generate the constraints defining how their dimensions match. If we find contradictory constraints for any permutation, we discard it. The algorithm is shown in Algorithm 3.

Automatic Matching Algorithm: Example. To illustrate this, Figure 4.5 shows an example where we have two functions with three 2D arrays each. First, the algorithm generates all the permutations between 0 and $n-1$ ($n = 3$ in this example). Then, for each permutation, it tries matching each variable in every

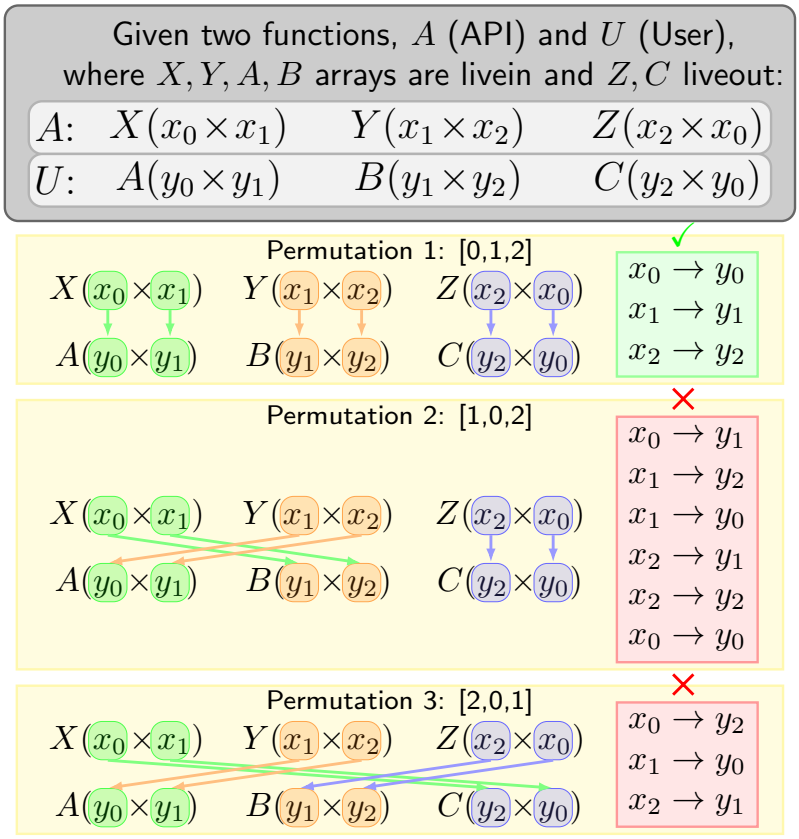


Figure 4.5: Example application of the matching algorithm. Given two functions, A and U, with three 2D arrays each, the algorithm generates the $3! = 6$ permutations (only the first three shown), finding the right combination (the first one) automatically.

array in the user code (function U) with the corresponding variable in the array of the API (function A). We represent arrays dimensions of the user code with x_0, x_1, x_2 and the API arrays dimensions with y_0, y_1, y_2. For simplicity, we show only the first three of the six possible permutations.

In the first case (permutation $[0, 1, 2]$), the algorithm tries matching the array variables of the user program X, Y, Z with API parameters A, B, C. We then examine each of the variables defining each of the corresponding arrays. Comparing X and A gives a match of $x0 \to y0$ and $x1 \to y1$. For the second array variable Y and API parameter B, we have $x1 \to y1$ and $x2 \to y2$ and for the third variable pair Z, C, we have $x2 \to y2$ and $x0 \to y0$. All of these are consistent with n=3 constraint, which satisfies the condition (`dimsMatch` in Algorithm 3). This condition checks that the number of connections between user and API pa-

rameters is equal to the number of arrays. In the end, this means that there is no variable in the user program that matches more than one variable in the API, which is a clue of inconsistency. In our example, liveout information is also satisfied because Z is matched with C (both being liveout), so this permutation is added as a potential mapping. In the second permutation $[1, 0, 2]$, where X, Y, Z maps to B, A, C, the constraints are inconsistent *e.g.* $x1 \rightarrow y2$ and $x1 \rightarrow y0$ leading to $6 \geq 3$, so it is not a valid match. In the third and last example, constraints are equal to n, but the liveout arrays do not match. X is not liveout and it is matched with C which is liveout, so the liveness condition (`outMatch` in Algorithm 3) is not satisfied. Thus, the only valid match is the one in the first permutation.

4.2.3.2 Using Argument Names

Programs are developed by humans, so we can assume that the functions that humans write follow common patterns. We exploit this by analyzing the argument names of the API and the user program to find lexical similarities.

To compare argument names, we use the Levenshtein distance [111] to compute the distance between each of the user programs and API arguments. Figure 4.6 shows the definition of the Levenshtein distance, which calculation is based on the minimal number of modifications needed to transform one word into another, representing how close are those words. After computing the distance, the compiler selects the combination that minimizes the Levenshtein distance.

Figure 4.7 shows an application example of the Levenshtein distance to a real case of GEMM matching. For calculating the distance, we strip the API suffix (`tc_`) and convert all names to lowercase. Results show that the most probable mapping for `tc_A` is `A` in the user code, and for `tc_lda` is `lda`, which are the right matches.

$$lev(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ lev(tail(a), tail(b)) & \text{if } a[0] = b[0], \\ 1 + min \begin{cases} lev(tail(a), b) \\ lev(a, tail(b)) \\ lev(tail(a), tail(b)) \end{cases} & \text{otherwise} \end{cases} \tag{4.1}$$

Figure 4.6: Levenshtein recursive definition.

```
gemm_api(float* tc_A, float* tc_B, float* tc_C,
         int tc_m, int tc_n, int tc_k,
         int tc_lda, int tc_ldb, int tc_ldc,
         float tc_alpha, float tc_beta) {

gemm(int M, int N, int K, float alpha,
     float *A, int lda, float *B, int ldb,
     float beta, float *C, int ldc) {
```

	tc_A	...	tc_lda	...
M	1	...	3	...
N	1	...	3	...
K	1	...	3	...
alpha	4	...	3	...
A	0	...	2	...
lda	2	...	0	...
.....	.....	...	.....	...

Figure 4.7: Levenshtein distance calculation for the arguments of the tensor core API (above) and an example user program.

4.2.3.3 IO Generation

Once we have a candidate match, we generate random inputs of different sizes and test for input-output (IO) equivalence. We use 30 inputs of varying sizes. Although IO behavioral equivalence is not proof, we can increase the number of tests for increased confidence. No existing technique such as IDL or KernelFaReR can prove that a matched piece of code is provably equivalent to an API and therefore rely on user sign-off. Although all library replacement techniques require ultimate sign-off, our approach acts as a kind of "API code pilot", drastically reducing the programmer effort.

Behavioral Equivalence and the Limits of Verification. ATC, like prior work on floating-point accelerators [196], uses behavioral equivalence. The downside of this strategy is that it requires programmer sign-off to make any substitution. However, due to the complexities of verifying floating-point programs [196], verification of such liftings are some way off.

In summary, the key challenges that all competing techniques face are:

- Floating-point numbers often raise challenges in theorem provers as they are challenging to reason about.

- Floating-point functions may have different accuracies in different input ranges, meaning that the obvious checks of correctness (even within bounds) are difficult to apply.

The backend of ATC is not tied to using behavioral equivalence. As we will see, the use of such behavioral equivalence results in no false positives. Further development of theorem prover technologies would mean that the weak behavioral equivalence in ATC could easily be replaced with a theorem prover guaranteeing correctness and enabling automatic transformations.

4.2.4 Automatic Profitability Detection

If a workload can be accelerated or not heavily depends on the input size, which is only known at runtime. An approach to profitability prediction is to analyze the performance of the user program and the accelerator using its API. This would involve analyzing the performance of each user program (for example, extracting static and dynamic features). A machine learning approach might be used to predict the speedup of the code with respect to the accelerator. Normally, we would have to go through all this complex checking. However, we can exploit the fact that, whatever the user code is, it must be semantically equivalent to a well-known, optimized version of that program (executed on CPU). Because if it is not semantically equivalent, we would not be able to compile it to the accelerator. Then, we assume (and evaluate in Section 4.3) that the optimized version of that program is always equal to or faster than the user program. Even though this might not be true for tiny inputs, where the user code might be slightly faster than an optimized library, the effect on the performance would be negligible. Therefore, instead of running the user code as it is, we can always replace it with a call to the best implementation of that kernel, using an optimized library. This approach not only greatly simplifies the predictor (because now it does not depend on the user code) but also improves overall performance since when the accelerator is not used, the CPU code will be dramatically faster than the handmade implementation by the user.

Then, our predictor must decide whether it is best to run on a CPU (e.g., MKL for GEMM) or accelerator version (e.g., cuDNN for GEMM) of the library. Data is CPU resident, so input size also affects the cost of data movement between the CPU and the accelerator, which can incur enough overhead to outweigh the benefit from acceleration. To detect when offloading to the accelerator is profitable, we use a predictive model based on empirical data to enable accurate predictions as platforms and libraries evolve by retraining the model. Our model takes into consideration both compute time and memory movement overhead,

Table 4.1: Input sizes used by the predictor for matrix multiplication and convolution.

	Parameters
GEMM	M, N, K
CONV	N, C, H, W

so it will only offload computations when it's profitable, also considering data movement overhead.

SVM. We build a classifier that predicts the most profitable platform to execute on depending on runtime data values. More precisely, we use the well-known support vector machine (SVM) classifier with a polynomial kernel of degree 3 with gamma=1 and C=100. We sample the CPU and the accelerator with a common dataset of input sizes, which produces a dataset that is small enough to be processed in less than five minutes, but large enough to be highly accurate. Data is labeled with 0 or 1 meaning that the CPU or the XPU is faster. The model is then trained and deployed at runtime, when matrix sizes are known, The training phase is done only once, at "factory time", and the resulting model when deployed has negligible ($\leq 0.3msec$) runtime overhead (see Section 4.3.4.3). For ATC, we build two predictors: one for matrix multiplication and another for the convolution. Table 4.1 summarizes the input sizes used by ATC for both applications.

4.3 Evaluation

4.3.1 Test Bed

We evaluate GEMM and convolution acceleration on specialized platforms. For GEMM, we used an Intel i7-11700 (CPU) with an NVIDIA Quadro RTX 5000 (tensor cores) (XPU). For convolution, we used the Google Cloud Platform (GCP) services equipped with a TPUv3 with 8 TPU cores. Compilation benchmarks in Section 4.3.4.4 are executed in an AMD EPYC 7413.

The Intel/NVIDIA platform runs CentOS 8.3 with kernel 4.18.0. LLVM was downloaded from the official Git repository, using commit `329fda3`. User codes were compiled using gcc 11.2.0 with `-O3 -march=native` flags. We used cuBLAS 11.2 and MKL 2020.2.254 for compiling codes to the XPU and CPU, respectively. For compiling convolution programs to the CPU, we used oneDNN v1.96. The

TPU system runs Debian 10 with kernel 4.19.0-14. Codes were compiled to a wrapper using PyTorch 1.9 (Python 3.7).

4.3.2 User Code

Matrix multiplications. We explored GitHub looking for C and C++ GEMM codes, analyzing more than 400 programs from which we selected 50 programs. We discarded the rest of them because of wrong implementations, compilation errors or duplicated code. The final list of programs is shown in Table 4.2. We categorize the codes as follows:

- *Naive*: Naive implementations with the traditional 3-loop structure.
- *Naive Parallel*: As Naive, but with simple outer loop parallelization.
- *Unrolled*: Naive implementation with unrolled loops.
- *Kernel Calls*: Implementations that divide the loops into different function calls.
- *Blocked*: Tiled implementations.
- *Goto*: Implementations of the Goto algorithm [71].
- *Strassen*: Implementations of the Strassen algorithm [178].
- *Intrinsics*: Implementations using Intel intrinsics.

In addition, we selected 50 non-GEMM projects to check whether any of the approaches gave false positives.

Convolutions. We explored GitHub looking for C and C++ 4D convolution implementations. We analyzed around 50 programs from which we a selected list of 15 programs based on the same methodology used for selecting GEMMs. The list of convolution programs is shown in Table 4.3. There are some of the codes that do not support multiple batches ($N = 1$) or that have fixed kernel sizes ($FW = FH = 3$) which further complicates the compilation. We have included codes from the most relevant convolution implementations:

- *Direct*: The direct convolution algorithm.
- `im2col+gemm`: An algorithm that casts the input as matrices (im2col) and later uses a GEMM, as in Caffe [88].
- *Winograd*: The Winograd algorithm [194].

Table 4.2: List of GEMM codes.

Algorithm	Code	LoC	Layout	Sizes	Optimizations
Naive	1	22	Column-major	Squared	None
	2	127	Both	Any	None
	3	18	Row-major	Any	None
	4	41	Column-major	Squared	None
	5	11	Row-major	Any	None
	6	11	Row-major	Any	None
	7	30	Row-major	Any	None
	8	18	Column-major	Any	None
	9	40	Column-major	Any	None
	10	39	Column-major	Any	None
	11	43	Row-major	Any	None
	12	11	Row-major	Squared	None
Naive parallel	13	39	Row-major	Squared	OpenMP
	14	28	Column-major	Squared	OpenMP
	15	164	Row-major	Any	OpenMP
	16	22	Row-major	Multiple of nthreads	C++ threads
	17	107	Row-major	Squared	C++ threads
Unrolled	18	57	Row-major	Any	None
	19	50	Row-major	Any	None
	20	63	Row-major	Squared	OpenMP
	21	38	Row-major	Squared, multiple of bs	None
Kernel Calls	22	46	Column-major	Any	None
	23	115	Column-major	Any	OpenMP
	24	61	Column-major	Any	None
	25	105	Column-major	Any	Unrolled
	26	164	Column-major	Any	Unrolled
Blocked	27	104	Row-major	Any	Block
	28	30	Row-major	Squared	OpenMP
	29	52	Column-major	Any	None
	30	35	Row-major	Squared	None
	31	38	Column-major	Squared	None
	32	42	Row-major	Multiple of bs	Unrolled
	33	49	Row-major	Squared	None
	34	18	Row-major	Squared	None
	35	21	Row-major	Squared	None
Goto	36	247	Column-major	Squared	Intrinsics (SSE)
	37	89	Row-major	Squared	None
Strassen	38	210	Row-major	Squared	None
	39	315	Row-major	Squared, power of 2	None
	40	162	Row-major	Squared	None
Intrinsics	41	102	Row-major	Squared	Intrinsics (AVX2)
	42	91	Row-major	Multiple of 8	Intrinsics (AVX2)
	43	82	Row-major	Multiple of 8	Intrinsics (AVX2)
	44	58	Row-major	Any	Intrinsics (SSE)
	45	112	Row-major	Multiple of bs	Intrinsics (AVX2)
	46	136	Row-major	Multiple of bs	Intrinsics (AVX2)
	47	120	Row-major	Any	Intrinsics (AVX2)
	48	143	Row-major	Multiple of bs	Intrinsics (AVX2)
	49	57	Row-major	Multiple of bs	Intrinsics (AVX2)
	50	60	Row-major	Any	Intrinsics (SSE)

Table 4.3: List of convolution codes.

Algorithm	Code	LoC	Nº Args	Optimizations	Constraints	C struct?
Direct	1	35	12	None	None	No
	2	36	10	OpenMP	FW = FH = 3	No
	3	34	8	OpenMP	FW = FH = 3	No
	4	43	11	None	FW = FH = 3	No
	5	39	8	OpenMP	FW = FH = 3	No
	6	76	16	None	N = 1	No
	7	209	18	Vectorized	N = 1	Yes
	8	102	12	None	None	No
	9	42	16	None	None	No
im2col+ gemm	10	189	15	None	N = 1	Yes
	11	286	15	BLAS	N = 1	Yes
	12	179	17	BLAS	FW = FH	Yes
Winograd	13	687	17	Intrinsics + OpenMP	FW = FH = 3	No
	14	254	12	None	N = 1	Yes
	15	782	12	Intrinsics + OpenMP	FW = FH = 3	No

4.3.2.1 Methods

We evaluate our approach against 4 well known schemes:

- IDL: Idioms are described using an idiom description language [68], which is translated into a set of constraints over LLVM IR. It's extensible, because the description language can be adapted to different domains, but it's also brittle, because detecting code patterns based on constrains is usually hard.
- KernelFaRer: Uses different pattern matching to detect specific code constructs, matching specific matrix multiplication structures [46]. Works very well for basic GEMM programs but fails detecting and replacing more complex structures.
- Polly: Detects static control parts (SCoPs) in the code using the polyhedral model [73]. It does not replace the code with a call to an optimized library. Instead, it optimized the code using different transformations, following the polyhedral philosophy. Its limitations are equivalent to polyhedral's compilation.
- FACC*: FACC uses neural embeddings and behavioral synthesis to detect candidates for acceleration [196]. It is limited to 1D arrays so we developed an extended version, FACC*, which supports multi-dimensional arrays. But, unlike ATC, it does not have heuristics to reduce the matching search space or any profitability analysis.

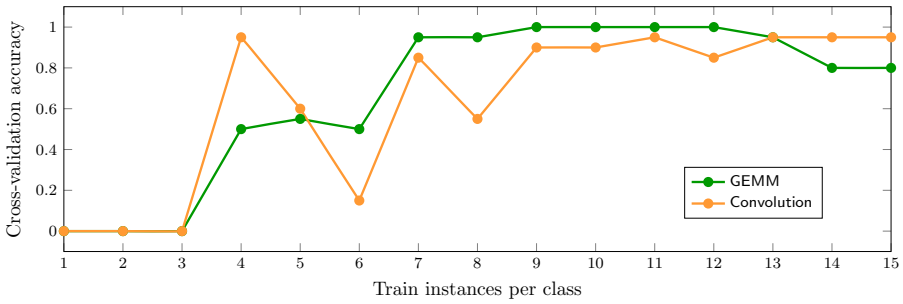


Figure 4.8: Cross-validation accuracy with mean and standard deviation of the neural classifier in terms of the number examples per class when trained using a reduced version of the OJClone dataset with GEMM and convolution examples.

4.3.3 Neural Code Classification

We rely on prior work on neural program classification to detect candidates for ATC. We added a new class of GEMM examples to the OJClone dataset and retrained the model. Figure 4.8 shows the accuracy of classification as we increase the number of training examples. It takes only a relatively small number of examples (10 codes) to build an effective classifier. We include examples for all algorithm types in GEMMs and convolutions, both sequential and parallel implementations, to make sure the network is trained to detect all possible code kinds. When applied to the 50 GEMM programs it was 100% accurate in classification.

4.3.4 GEMM

4.3.4.1 Detection

Figure 4.9 shows the percentage of GEMM programs matched by each technique across each of 8 categories listed in Table 4.2.

IDL. The constraint based scheme [68] only matches 6 out of 50 cases. These programs are largely naive implementations of GEMM, with a simple loop structure. It is able to manage 2 programs containing unrolled loops but fails on anything more complex. Matching more diverse cases would require writing a new IDL constraint description for each sub-class.

KernelFaRer. This code matching approach [46] is more successful, matching 11 GEMMs due to a more robust pattern matcher. For straightforward sequential

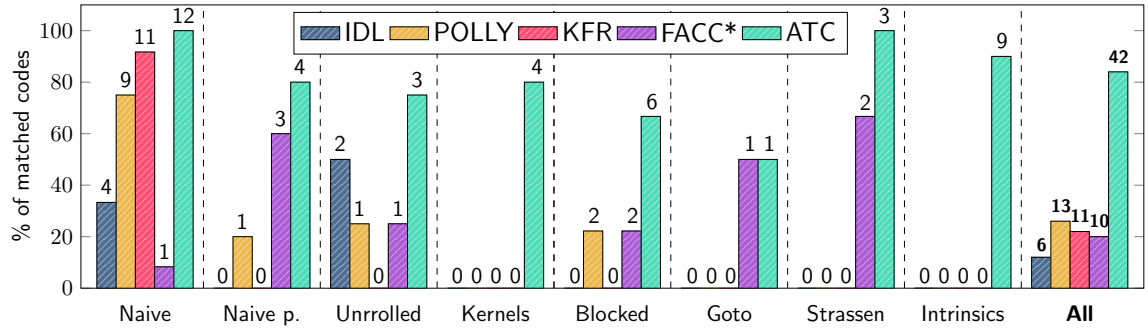


Figure 4.9: Percentage of matched GEMM codes by different techniques.

implementations, it is able to match all but one of the cases. However, any code variation, including loop unrolling, defeats it.

Polly. Although it does not match and replace GEMMs, it can detect SCoPs which may be candidates for replacement with appropriate API calls. It is less successful than KernelFaRer in detecting naive implementations but is more robust across other more complex categories including one parallel and unrolled version and 2 blocked cases. It slightly outperforms KernelFaRer, matching 13 vs. 11 out of 50 cases. While Polly finds SCoPs in many programs (it finds at least one SCoP in approximately 40% of the programs), sometimes they correspond to other structures that do not represent GEMMs. This is especially true in complex programs (from program 22 onwards).

FACC*. Unlike the other approaches, FACC* performed poorly on naive implementations, but better on others. Here, the size of the mapping search space is the limiting factor. It was able to find 10 cases in the available time (timeout $\leq$ 10 mins). We examine the reasons for this in Section 4.3.4.4.

ATC. Our approach is significantly more robust across all categories, matching 42 out of 50 cases. It is able to detect all naive implementations and the majority within each other category, outperforming all other techniques in each category. It detects more naive parallel implementations, unrolled and blocked programs than Polly and is the only technique to detect GEMMs in codes containing kernel calls and intrinsic instructions.

4.3.4.2 Accuracy

Figure 4.10 provides a summary of ATC's success and failure by type. In 8 cases ATC failed to detect that the program contained a GEMM. In one case, program

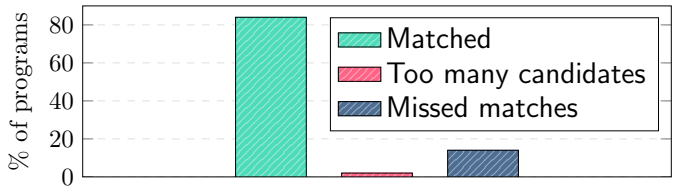


Figure 4.10: Percentage of matched GEMM codes by ATC divided by failure reason.

23, this is due to there being too many candidate matches, 280 which is above our timeout threshold of 100 candidates. The remaining cases are due to overly aggressive search pruning, missing a legal match. Improved search heuristics are likely to improve program coverage.

False positives. None of the methods classified any of the 50 non-GEMMs as a GEMM. Across all methods, there were no false positives.

4.3.4.3 Performance

The performance of each approach is shown in Figure 4.11. Polly is not included here as although it can detect SCoPs, it does not explicitly identify them as GEMMs for API replacement. We show two bars for KernelFaRer, which correspond to the strategy of GEMM code with an optimized CPU implementation as described in [46] and KFR (XPU) which is our extension, replacing the CPU library with the optimized XPU implementation. IDL and FACC* directly target the accelerator, while ATC chooses the CPU or accelerator based on its SVM platform predictor. This runtime prediction cost is negligible $\leq 0.3msec$ and included in Figure 4.11.

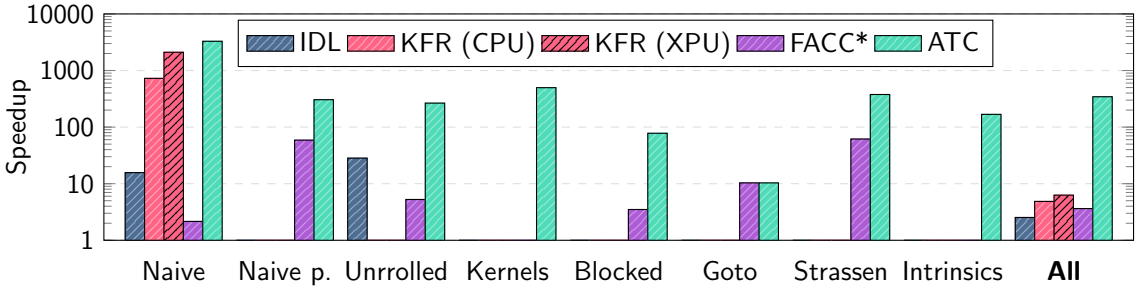


Figure 4.11: Geometric mean speedup obtained by IDL, KernelFaRer, FACC* and ATC in GEMM programs with $n = 8192$.

What is immediately clear is that detecting more GEMMs leads to better overall speedup. In the Naive category, KFR and ATC are both able to achieve good performance, with a speedup of 726x and 1031x, respectively. The gap is narrowed when using KFR (XPU). However, KFR is unable to detect GEMMs in any other category leading to just a 6.2x speedup overall while ATC achieves 344.0x. Unsurprisingly, there is more performance available on naive sequential implementations than in those cases where the programmer has spent effort in optimizing the program (e.g., parallel, blocked and intrinsics). Despite Strassen algorithm being theoretically more efficient, in practice, acceleration gives significant performance improvement.

4.3.4.4 Candidate Search Complexity and Compile Time

One of the key challenges in matching code to APIs is searching for program variables that map to API formal parameters. As the width of the API and complexity of the user program increase, this becomes combinatorially expensive. Figure 4.12 evaluates FACC* naive matching of variables and our approach based on the Levenshtein distance. Naive matching varies considerably from just 4 candidates to over 1 million. Our approach greatly reduces the number of candidates for the majority of the programs. There are a few exceptions (e.g.: 20), where the number of candidates is already small, where our approach is unable to improve. There is one special case, code 23, where we reduce the number of candidates, but it is still too high, making ATC unable to compile the program in a reasonable amount of time.

Figure 4.13 shows the compilation time of ATC, which strongly depends on the number of candidates. The initial neural classifier has a negligible constant execution time of 1.3 seconds, while the other phases' compilation time grows with the number of candidates. We omitted the SVM classification time, which

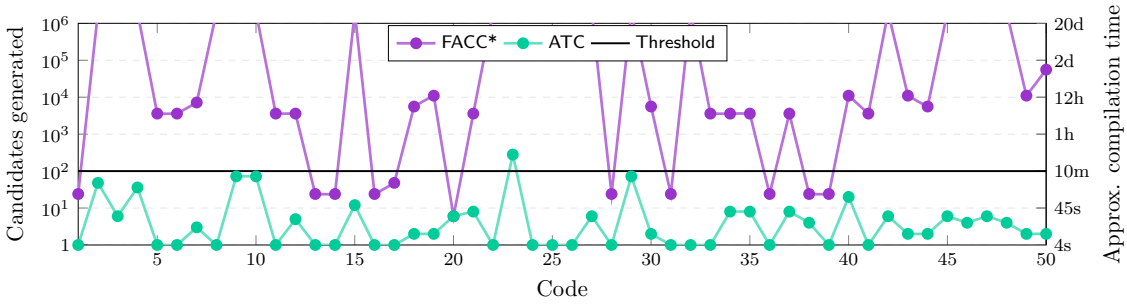


Figure 4.12: Comparison of the number of candidates generated for matching GEMM codes: FACC* vs our approach.

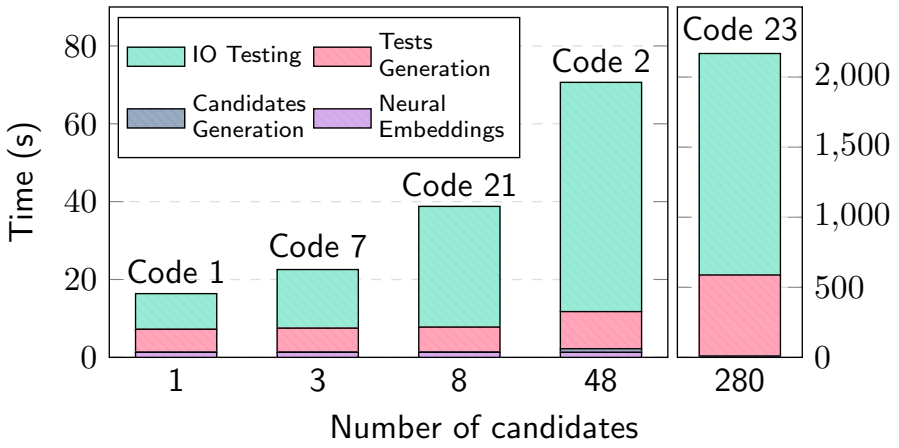


Figure 4.13: ATC compilation time for different number of candidates.

takes around 0.3 msec, since the influence on the execution time is negligible. As the number of candidates begins to increase (e.g., more than 50), the compilation time becomes prohibitively expensive. Code 23 has 280 candidates which would take 35 mins more to evaluate. We limit the number of candidates considered to 100 which corresponds to a timeout of $\leq$ 10 minutes.

4.3.4.5 Profitability Accuracy

To measure the accuracy of the SVM platform predictor, we built a model offline and tested it on unseen data values.

Table 4.4 summarizes the SVM accuracy with different input sizes and shapes. In the table, 111 means m = 1 $\times$ m, n = 1 $\times$ m, k = 1 $\times$ m, 123 means m = 1 $\times$ m, n = 2 $\times$ m, k = 3 $\times$ m, etc. The SVM achieves a global accuracy of 99.7%, where the misprediction occurs between $m = 2000$ and $m = 8000$ which is the "edge" between the CPU and the XPU. In all other intervals, the prediction is always correct. The best accuracy is achieved with non-squared matrices, while square matrices give slightly lower accuracy. Overall, this is a highly accurate predictor with a negligible runtime overhead of $\leq 0.3msec$.

In practice, the classification error has little impact on performance as shown in Figure 4.14, where we plot the % of maximum performance achievable with optimal platform selection. For $m = 6400$ the performance achieved drops to 84% of the maximum, but otherwise achieves an average of 96%. The accuracy is high when there is a significant difference between XPU and CPU performance. When accuracy drops, the relative cost of miss-classification is much lower.

Table 4.4: SVM accuracy for different sizes.

Parameter Value (mnk)	m					Global Accuracy
	2000	4000	6000	8000	10000	
111	100%	100%	100%	70.0%	100%	93.8%
123	100%	78.9%	100%	100%	100%	95.9%
312	100%	84.3%	100%	100%	100%	96.9%
136	100%	89.5%	100%	100%	100%	97.9%

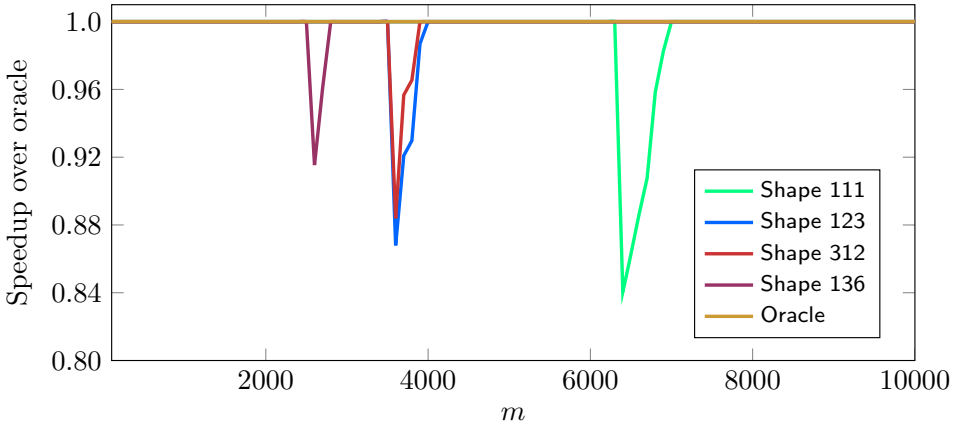


Figure 4.14: Percentage of speedup lost by ATC compared to optimal switching between CPU and XPU depending on matrix shapes.

4.3.5 Convolutions

Our approach is generic and can be applied to other APIs other than GEMMs. As an example, we consider tensor convolutions which are a significant component of DNN workloads. We provide a shorter evaluation of the convolution that includes the key components: detection and performance.

4.3.5.1 Detection

While IDL, KernelFaRer, Polly and FACC* were unable to detect any of the convolutions, ATC detected 10 of the 15 convolutions as shown in Figure 4.15; The neural program classifier was able to detect all convolutions, but we were unable to match 5 due to the excessive number of candidates.

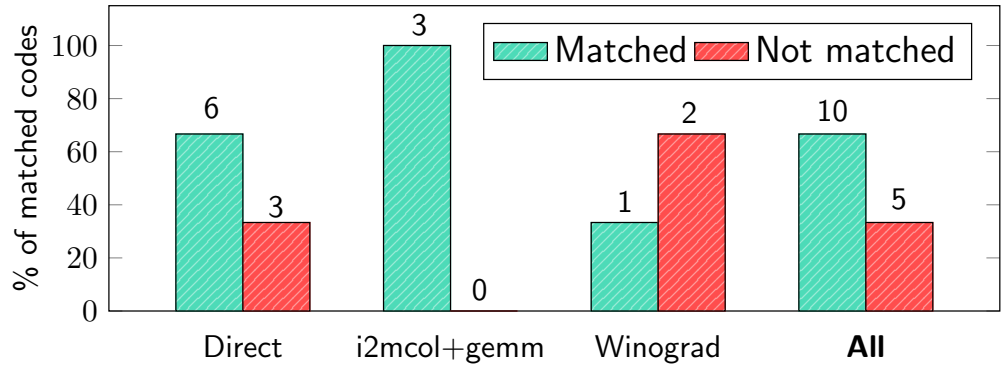


Figure 4.15: Matched convolution codes by ATC.

4.3.5.2 Performance

Figure 4.16 shows the performance achieved by replacing programs with library code for each of the programs we are able to accelerate. Across all codes, the SVM predicts that the TPU accelerator outperforms the CPU, giving an average 17.8x performance improvement across the programs. The input size correspond to a valid input of a real-world CNNs, AlexNet [183]. Because there are some programs that have input constraints (see Table 4.3), we adapted the input size to match that constraints in the user codes and the CPU/TPU for a fair comparison. The TPU is optimized for full inputs size (with no restrictions regarding batch size), which can be confirmed in our experiments, where we found that those user programs achieve lower speedups on the TPU.

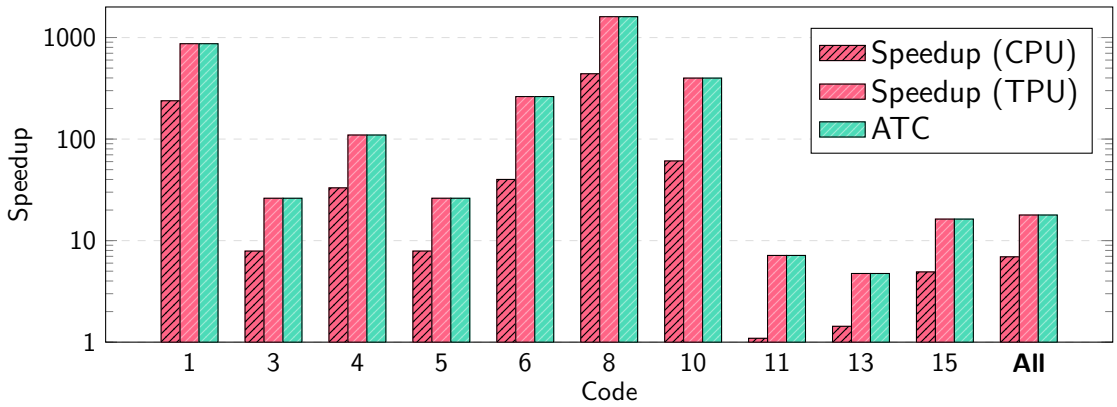


Figure 4.16: ATC speedup in convolution programs with $h = w = 224$, $kw = kh = 11$, $c = 3$, $k = 96$ and $n = 100$.

4.3.6 Productivity

In Section 3.4.5 we evaluated different single-source languages with focus in their portability, productivity and performance. We mentioned that productivity was harmed when code was already written, since using accelerators would require rewriting code. With ATC we are able to solve this productivity problem. Following the comparison shown that section, ATC would be classified like shown in Figure 4.17. First, ATC can provide performance in line with the best approaches shown in Section 3.4.5 because it also uses vendor libraries for optimal performance. Furthermore, ATC supports a wide hardware diversity: CPUs, GPUs (CUDA cores), tensor cores and TPUs. Thanks to our novel technique based on search and replacement, ATC achieves superior productivity for existent code, outperforming previous works. We plot ATC with dotted lines for performance and portability because ATC is not considered a programming language itself, like the ones we have evaluated previously, but it can be integrated with such languages easily, achieving a result like the one we show in Figure 4.17.

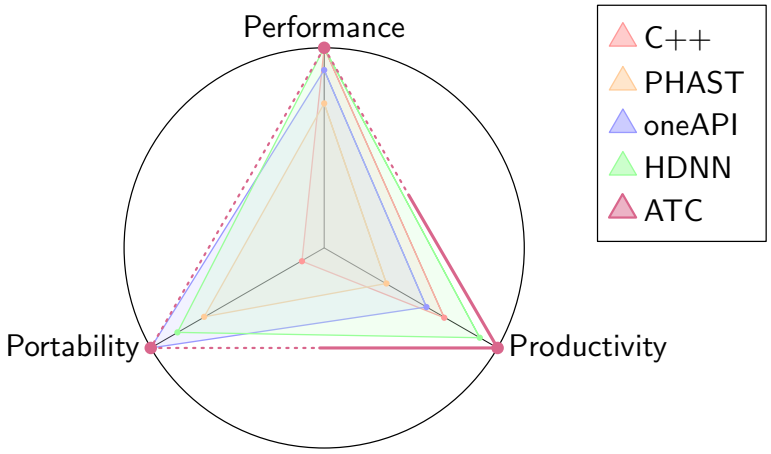


Figure 4.17: P^3 analysis of ATC compared to other approaches.

4.4 Related Work

Matching in Programs. Matching high-level program structure has been used to discover parallelism [50], heterogeneous offloading [13, 130] and many other core compiler tasks [65]. Constraint languages make these tasks easier [68, 23, 65] but their constraints are very sensitive to code structure [46]. For

matrix multiplications in particular, KernelFaRer [46] provides a more robust approach, detecting characteristics that define matrix multiplications. Polyhedral analyses can also be used to target matrix multiplication accelerators [21, 182]. Similarly to neural classification, IO equivalence could be proven by creating a polyhedral representation and comparing it against the polyhedral specification of the API. FACC [196] uses IO equivalence, which is robust to program structure, but only addresses the challenges of FFTs and does not scale to longer function signatures used for GEMM. To support any accelerator type, the compiler should support multi-dimensional arrays, while FACC only supports 1D arrays. Because in 1D arrays and FFTs the search space in matching the API parameters is small, FACC does not include anything to reduce it. With more complex programs and domains, this limitation makes compiling programs intractable. Mask [170] uses symbolic execution to prove equivalence, which does not work well for floating-point problems. Fuzzy classification techniques based on code clone detection [115, 180], domain-classification [186], pattern matching [33], code embeddings [7, 6, 47] and identifiers [138, 101] can be used to help compile to accelerators [196]. These classification strategies are able to classify diverse code structures, but do not provide a compilation strategy for using an accelerator on their own. A large class of techniques focus on migrating *between* APIs. These techniques often use program synthesis [34], NLP [135] and code embeddings [134, 161]. These techniques are unable to extract existing code into APIs.

GEMM Accelerators. GEMM and convolution accelerators have exploded with the rise in popularity of machine learning. NVIDIA's NVDLA [142], Intel's portfolio of ML hardware accelerators [86] and Arm's NPU [15] are examples of the proliferation of ML accelerators. Even companies not traditionally involved in the hardware space have developed GEMM and convolution-capable accelerators, such as the Google [93], Meta [12] and Microsoft [62]. Beyond this, there has been a proliferation of academic hardware accelerators for matrix multiplication [165], optimized for many different special cases such as sparse matrices [149] and low-power domains [184]. Matrix multiplication hardware accelerators have been proposed for novel technologies such as PIM [92, 198, 54] and photonic computation [205].

Compiling for GEMM Accelerators. Existing compilation strategies largely focus on *lowering* code from intrinsics to accelerators using rewrite rules [174, 171, 192] and synthesis techniques [38]. Existing approaches to extracting matrix multiplications [68, 46] are brittle. Synthesis-based techniques [3, 124, 14] and rewriting-based techniques [26, 175] have been developed to extract these DSLs

that can then be lowered: but they largely require flexible DSLs, rather than APIs presented by hardware accelerators.

Performance Prediction. Predicting code the performance of hardware accelerators is challenging, as the break-even point may depend on many different arguments within a function's interface [8]. LogCA [8] introduces static performance comparison models for hardware accelerators and similar models have been applied in offloading tasks [201]. While LogCA and parameterizations provide full performance models, many models only attempt to achieve a binary classification (should or should not offload). Machine learning has often been applied in profitability settings, such as OpenCL Kernels [189, 190] and OpenMP [126]. Similar techniques have been applied to FPGAs, by estimating power/performance [64] and tracking actual performance [166].

4.5 Conclusions

In this chapter we presented ATC, a flexible domain-agnostic compiler that matches legacy linear algebra and tensor code to accelerators. In a world dominated by accelerators, automatic hardware acceleration of legacy programs is a novel and relevant topic in recent literature. Many codebases were written years ago for specific devices (typically CPUs) and are still in use nowadays. Thus, this technique allows compiling legacy codes to specialized accelerators, dramatically improving their performance, without needing to rewrite existent code. By using IO behavioral equivalence and smart search space reduction, we are able to match over 80% of challenging real-world programs to accelerator APIs, significantly outperforming all alternative approaches. Thanks to that, ATC achieves an overall speedup of 344.0x over user codes in GEMM with a medium input size. Supporting new domains different from GEMM and convolution is easy because ATC focuses on behavior rather than code structure, which makes it very flexible and extensible. Furthermore, to support other accelerators in GEMM or convolution, only the accelerator API is needed: ATC adapts to the new specification automatically. We have also proved how to offload compute-intensive code to the accelerator only when it is profitable, which significantly enhances global performance. Overall, we believe that our approach could be implemented with LLVM as a mainstream compiler (like gcc) to feature the acceleration of real-world code.

A step forward in performance. ATC exploits new-generation hardware accelerators by replacing CPU code with accelerator API calls. Despite this notable

advance in productivity, we have not yet analyzed any proposal that exploits execution on multiple accelerators concurrently. In our analysis in Figure 4.17, we show that ATC, like other approaches, achieves optimal performance. But this performance limit is constrained to using a single accelerator. When considering multiple devices, this performance limit is further ahead. Because computing systems with various accelerators are increasingly common, not exploiting concurrent execution leads to hardware underutilization. Then, rather than executing in a single accelerator, using multiple accelerators for a single task could enhance performance and/or energy efficiency. We believe that this will be an essential part of any computing system or compiler in the future. Mainly because otherwise, a high density of accelerators can ultimately lead to highly underutilized systems.

Chapter 5

Exploiting Accelerator-Level Parallelism

5.1 Introduction

5.1.1 Motivation

Due to the end of Moore's law [128], recent work in heterogeneous systems has exploded, both from hardware [156] and software [120, 118, 119, 122] points of view. Yet, most software technologies to support heterogeneous hardware often forget the fact that most systems (especially SoCs) have capabilities for concurrently executing one or more programs on multiple accelerators. Instead, they focus on a single accelerator scenario and forget about the rest of the hardware available in the system, wasting great computing resources. However, having multiple accelerators is not the same as having multiple cores in a multicore CPU; accelerators are highly diverse and cannot be used concurrently easily.

With the evolution of computer science, different paradigms appeared to boost computers' performance. All of them (ILP, TLP and DLP) have provided critical advances in computer architecture [80]. Those innovations were supported by the ability to keep increasing the transistor count inside the CPUs, until the end of Moore's law. Since then, computer architecture is transcending to what some authors call the next computer architecture paradigm; Accelerator-Level Parallelism (ALP) [80]. This new kind of parallelism seeks to execute workloads in multiple accelerators concurrently, thus exploiting parallelism at

the accelerator level. The first manifestation of ALP are SoCs, as they include many accelerators that can be used concurrently, thus providing ALP.

Scheduling is a well-known topic that has been extensively studied in recent years. Previous works in this field have usually considered offloading different parts of an application to specific devices (typically, manycores like the Xeon Phi or GPGPUs). In other words, the workload was typically offloaded to one device at a time, meaning that only one device was executing code in a given period. In contrast, ALP seeks the evolution of this idea, the co-execution, which consists of using many accelerators at the same time, similarly to how ILP concurrently employs multiple functional units. ALP is a new type of parallelism that builds upon the existing ones, meaning that the ALP also exploits other parallelism types. However, co-execution in heterogeneous environments is challenging since the software needs to divide the work into parts and schedule them among radically different devices. In this context, the scheduling may pursue different objectives, like minimizing the execution time, the energy consumption, or even both [163]. In either case, achieving it depends heavily on the target hardware platform.

5.1.2 Research Context

This chapter presents Predict, Optimize, Adapt and Schedule (POAS), a framework for scheduling an application to run concurrently on multiple accelerators. POAS can be configured to focus on minimizing the execution time (high-performance) or minimizing the energy consumption (energy efficiency). Figure 5.1 depicts a general view of our framework, which takes one application and executes it in ALP, significantly improving the application performance and/or energy efficiency. The framework is divided into four general steps. The first one, predict, consists of developing a prediction model of the execution

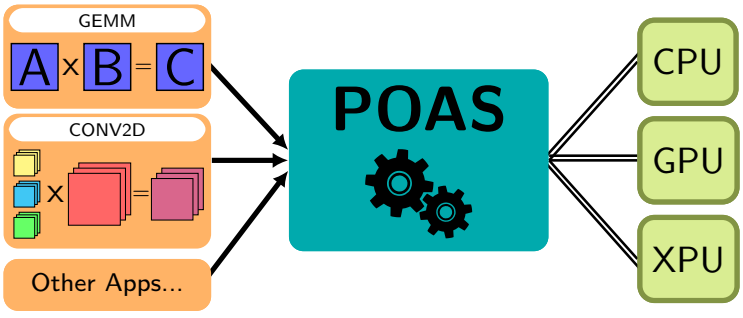


Figure 5.1: POAS operation overview. The framework takes different applications and executes them in co-execution, providing ALP.

time of the CPU and the accelerators, as well as the memory cost to copy the data between the CPU and the accelerators. In the optimization step, the performance prediction model is used to build a constraint satisfaction problem (CSP). The problem is then optimized to find the values so that the objective function is minimal. Lastly, the results given by the solver may need to be adapted so the scheduler can use them in the last step of POAS.

To demonstrate how POAS works, we apply our method to two relevant case studies: matrix multiplication and convolution. We implement POAS as a framework that runs matrix multiplication and convolution workloads in ALP, supporting multi-core CPUs, GPUs and XPUs (tensor cores), an accelerator for matrix multiplication and DNN workloads. Because POAS is easily extensible to other workloads, it is very suitable to provide ALP for a wide range of applications. Ideally, we believe that POAS could be implemented like a middleware at the OS level, similarly to how Intel Thread Director [85] works.

Unlike previous works that offload workloads to one device at a time, POAS aims to execute one single task in many accelerators concurrently. Previous works have already studied scheduling in heterogeneous scenarios, but most of them are application dependant (like [97]), while POAS is completely extendible to any application. Furthermore, experimental results highlight that POAS can exploit ALP with negligible overhead, reaching near optimal results.

The main contributions of this chapter are:

- Defines a novel framework for exploiting Accelerator-Level Parallelism (ALP) in heterogeneous environments.

- Details how the proposed framework works in two real-world applications like matrix multiplication and convolution, running in CPU, GPU and XPU.

- Presents an experimental evaluation of the proposed framework showing that the proposed framework achieves performance close to optimal.

The rest of the chapter is organized as follows. In Section 5.2.1 we present POAS, our framework for allowing co-execution in heterogeneous environments. We detail how POAS works in real-world applications like matrix multiplication and convolution in Section 5.2.2. A performance evaluation of POAS is shown in Section 5.3. Section 5.4 presents the related work in scheduling and co-execution state-of-the-art techniques, as well as heterogeneous matrix multiplication and convolution approaches. Finally, Section 5.5 concludes the section and gives some hints for future work.

5.2 Exploiting Accelerator-Level Parallelism

5.2.1 Predict, Optimize, Adapt and Schedule (POAS)

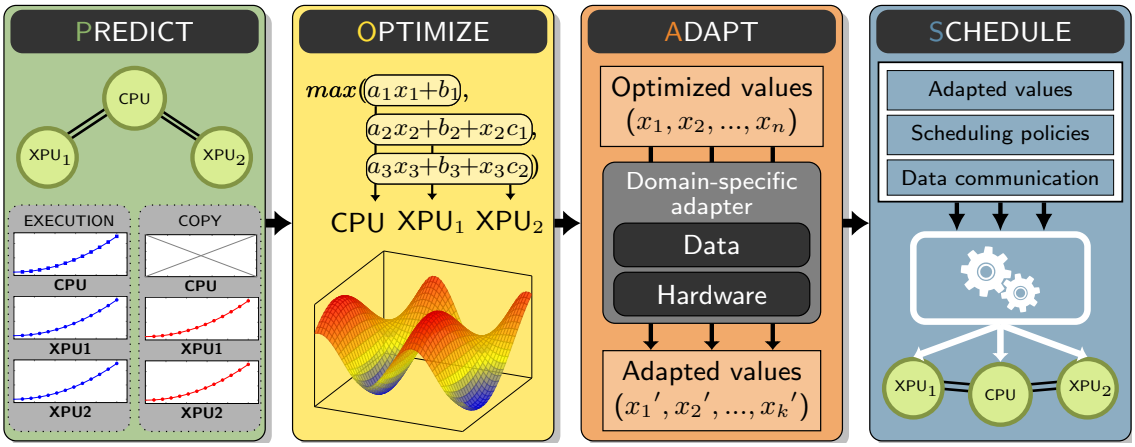


Figure 5.2: General overview of POAS (Predict, Optimize, Adapt and Schedule) framework.

POAS (Predict, Optimize, Adapt and Schedule) is a framework that schedules any application to be executed in ALP environments. A general view of POAS is depicted in Figure 5.2. The framework is divided into four phases (Predict, Optimize, Adapt and Schedule), which must be performed in order. In this sense, the output of each phase is the input of the next one. All phases are mandatory except for the Adapt phase, which is optional. The framework can be tuned to achieve different goals in ALP environments: minimizing execution time, energy, or both, considering a tradeoff between performance and energy. The predict phase must be tuned to pursue one of those goals. It is worth mentioning that, like other scheduling approaches, POAS is designed to for scenarios where there is a significant amount of work to do. If not, ALP would not provide substantial gains over the execution on a single device.

5.2.1.1 Predict

In the predict phase, a performance predictor is designed, and the profiling of the hardware platform is performed.

Predictor. The goal of the prediction is to give a precise estimation of the execution time (or energy needed) of the application. This prediction is software and hardware-dependent, so the prediction must consider both application and

hardware characteristics. POAS is a modular framework, so any performance or energy prediction method can be chosen in this phase. There are many performance prediction approaches, and depending on the domain, one predictor would be more suitable than the others. The POAS framework could implement different predictors that would be used depending on the application. Furthermore, in the case of performance prediction, the performance model must predict both the execution time and the time spent in memory transfers between the CPU and the accelerators over the bus. The only requirement for the performance predictor is to provide a function that, given the input size, predicts the execution time of the application. While the resultant function has no restriction regarding its complexity, it is desirable to have a linear or quadratic function, as discussed in Section 5.2.1.2. Regression or similar methods can be used for computing the function from the measured values in the profiling. To achieve competitive performance, the accuracy of the predictor is vital. If the prediction fails to precisely reproduce the experimental results, the scheduling would be poor. At this point, it is worth noting that with POAS it is not mandatory to have the source code, which makes the framework more flexible since it does not depend on the programming language, which is a limitation in many language-centered models.

Hardware profiling. As part of the prediction phase, a profiling of the hardware platform is also necessary. With profiling, the hardware is sampled with different input sizes, and time is measured to build the function that maps the input size into execution time or energy consumed. One key aspect before profiling is to study the behavior of the hardware executing the application because sometimes the hardware provides different performance results depending on data sizes, alignment, and other factors. For example, in matrix multiplication, tensor cores only provide optimal performance if $m \ \% \ 8 == 0$ and $k \ \% \ 8 == 0$ [140] (where m and k are matrix dimensions).

5.2.1.2 Optimize

The optimization phase takes the prediction model generated in the previous step as input. This phase has two objectives: to define a formulation of the application's behavior and to optimize it. The output of this phase is a set of optimized values, which typically represent the input size of each device, such that the desired objective function is optimized.

Formulation of the problem. The formulation is expressed as a constraint satisfaction problem (CSP), which can be enunciated to achieve different goals. In

many cases, however, the problem can be further specialized into a constrained-optimization problem (COP), which is a generalization of the CSP. It is crucial that the mathematical formulation models all the details of how the application works in the real world (i.e., when the compute and communication phases occur and how). The formulation of the problem is the only manual part of the whole framework since it is application-dependent. Depending on the application, communication schemes, and other factors, different applications may need different formulations. Likewise, one formulation might be reused for many applications if they behave similarly.

Optimizing the problem. Regarding methods for optimizing the model, linear or quadratic programming can be used, providing the optimal solution in very little time. However, these methods can only be used if the function that models the behavior of the application is linear or quadratic. Considering that there might be cases where the performance model is too complex to be represented in these terms (e.g., the function is cubic), the problem should be formulated as a CSP. In this case, alternative methods (like backtracking, local search, etc) can be used to optimize the performance model. The POAS framework implementation can provide different solvers, which would be used by the appropriate application.

5.2.1.3 Adapt

This is the only optional phase of the POAS framework. Depending on the application, the variables that come from the optimized model designed in the previous phase might need some transformations to be used by the scheduler. Therefore, an intermediate phase called adapt might be needed to make the scheduler work correctly. The output of this phase is always a set of valid values to the scheduler. If the input of the adapt phase is a valid input to the scheduler, data is left unmodified, and the adapt phase is a no-op. Otherwise, the adapt phase performs an adjustment of the data. We differentiate between two types of adjustments: data and hardware adjustments.

Data adjustments. This kind of adjustment is needed when the output of the optimized model contains different variables than the one needed to determine how to schedule the application. In these cases, the adapt phase must adjust the values given by the optimization phase to some values that can actually be used in the scheduler. For example, let's say that the scheduler needs the number of elements to be computed by each device within a vector, but the output of the optimize phase is the start and the end of the portion of the vector to be

computed. Data adjustments depend on the application so this procedure is essentially application-dependent.

Hardware adjustments. Generally speaking, hardware is very sensitive to data sizes and other factors, so performance might vary depending on the input size. This is very harmful to prediction accuracy and is something that must be solved in this phase. The goal of hardware adjustments is to ensure that the input of the next phase (scheduling) matches the same performance conditions as the previous phase (profiling).

For example, let's consider the case of tensor cores. As we discussed, tensor cores typically perform differently depending on the size of m and k. Let's say that we perform the profiling phase assuming that input sizes will always be multiple of 8, the best-case scenario. However, the optimized values m and k given by the solver do not have to be a multiple of 8. This phase takes care of these low-level details, which are key for high-quality prediction accuracy.

5.2.1.4 Schedule

Within the POAS framework, the scheduler can work in two different ways: static and dynamic. Other scheduling policies, as well as modifications to the presented ones, are left for future work. The scheduler policy must also include how to manage the communications between the CPU and the accelerators, which might have a significant impact on performance. The framework might implement different schedulers that work better or worse for different applications, allowing users to select the best scheduler for each case.

Static scheduling. The static scheduler uses the performance model and optimizes the problem formulation to get the optimal inputs for each device. It is the simplest mode, as the work distribution does not change over the execution of the program. This mode works well when the application requirements do not change over time and when the performance prediction can model precisely the behavior of the hardware. If one of these requirements is not met, static scheduling would provide inaccurate predictions of the execution time of the application, leading to suboptimal scheduling where the hardware utilization may decrease significantly.

Dynamic scheduling. To overcome the aforementioned problems, a dynamic scheduler can be employed. In the dynamic scheduler, the performance prediction model is used to optimize the function and obtain the optimal values at the beginning of the execution, just like static scheduling. But, unlike static schedul-

ing, the scheduling might vary over time. We differentiate two approaches to dynamic scheduling:

- Adapting the prediction values: Instead of having fixed values for the performance or energy of each device, dynamic scheduling modifies those values during execution. This can be beneficial if an application's performance varies over time (e.g., hardware can be added or removed dynamically, the frequency varies significantly, etc). One approach is to be constantly measuring the execution time of the application and adapting the performance model over certain periods, which granularity can be adjusted as needed (e.g., every second, or every program iteration).

- Pairing with a queue-based system: The static performance prediction can be paired with a dynamic queue to tune the work distribution. This way, prediction acts as a heuristic to indicate where is more promising to send part of the work to be executed, while the queue works in an FCFS manner.

Data communication scheme. In work distribution, the effective use of the memory bus is a performance crucial aspect. In ALP environments (like SoCs), accelerators are usually connected to a shared bus, where all of them can communicate with the CPU. Hence, optimizing applications for exploiting ALP is challenging since the bus (thus, the throughput) must be shared among the accelerators.

As a first approach, we propose a scheduler based on priority scheduling. The idea is to assign a priority to each device connected to the shared bus. Then, data is copied to/from the CPU in the order dictated by the priority ordering. There are many approaches to designing this scheme with different goals, like minimizing the idle time of accelerators. We leave for future work to further investigate more efficient approaches.

5.2.2 Using POAS to Schedule GEMM and Convolution

This section details how POAS can schedule real-world applications. First, we detail all the phases for matrix multiplication. Later, in Section 5.2.2.5, we highlight only the differences between GEMM and convolution, since most of the workflow in POAS for GEMM and convolution remains the same.

We designed a POAS implementation focused on minimizing the execution time, targeting CPUs, GPUs and tensor cores (from now on, XPUs). The implementation relies on optimized libraries to perform the matrix multiplications: MKL (in Intel CPUs), BLIS (in AMD CPUs) and cuBLAS (for both CUDA

and tensor cores). For convolution workloads, our implementation relies on oneDNN (in CPUs) and cuDNN (in GPUs).

5.2.2.1 Predict (GEMM)

Linear regression. To design the performance predictor for GEMM, we used a regression analysis approach. It is well known that GEMM general algorithm has a complexity of $O(n^3)$. But to use linear regression, we must find a way to represent the time with linear complexity. Thus, we model the execution time with the number of operations (from now on, *ops*), such that $ops = m * n * k$, where m, n and k are the matrix dimensions. In other words, the execution time grows with a cubic complexity if we consider the input size, but it grows linearly considering the number of operations.

While this linear function can generally predict the performance of GEMM, there are certain hardware peculiarities which might cause the prediction to fail. For example, the XPU will provide radically different results depending on the input size of the matrix, as the tensor cores can only be optimally used when the input meets some criteria. To eliminate ambiguity, the performance predictor always assumes optimal performance. Therefore, one additional task in the adapt phase is ensuring that real workloads can be computed in the same way that the predictor was trained for. We further contemplate these details in Section 5.2.2.3. In addition to the compute times, we also predict copy times between CPU and GPU.

Profiling. We perform a profiling step of the hardware platform, which is done only once at installation time and takes less than five minutes to complete. The profiling phase measures the computing power of all the hardware devices available in the system and the memory bandwidth between the CPU and the accelerators. Then, the results are stored in a text file that is read when real matrix multiplication workloads arrive. To improve prediction accuracy, we profile the performance of squared matrix multiplication only, rather than profiling many different matrix shapes. Restricting the profiling space can improve prediction significantly since the range of predicted inputs is smaller. Then, when a big, non-square matrix computation arrives, POAS divides the matrix into a list of squared matrices, which are equivalent to computing the whole matrix at once (we detail the slicing algorithm in Section 5.2.2.3). Using this approach, we predict the performance of all matrix shapes precisely. Therefore, the profiling phase consists of two steps:

- Computing power profiling: The program runs a set of squared matrix multiplications (using appropriate libraries like MKL, BLIS or cuBLAS).

The sizes of the squared matrices are variable and adjustable depending on the device (see Section 5.3.1.1 for more details). When all the experiments have finished, linear regression is performed to obtain the linear function that models the execution time of the device.

- Memory bandwidth profiling: The program runs a microbenchmark that measures the bandwidth between the CPU and each accelerator.

5.2.2.2 Optimize (GEMM)

In the optimization phase, we formulate a constraint satisfaction problem (CSP) that minimizes the execution time. Therefore, the goal of the solver is to find a distribution of *ops* among the hardware devices such that the total execution time is minimal.

Problem formulation. We express the execution and copy times as a mixed-integer linear programming (MILP) problem. We define c_x as the independent variables, which represents the number of operations (*ops*) to be computed by device x. We also define y_x as the function that gives the time to copy A, B and C matrices. The goal of the solver is to minimize the following objective function (which models the total execution time of the GEMM in n devices):

$$max(t_{c_1} + t_{y_1}, t_{c_2} + t_{y_2}, \dots, t_{c_n} + t_{y_n}) \tag{5.1}$$

where:

- n is the number of devices in the system.
- t_{c_x} is a linear function in the form $ac_x + b$ that models the execution time of the device x when it computes c_x operations.
- t_{y_x} is a linear function that models the copy time of the device x when it computes c_x operations (if x is a CPU, then $t_{y_x} = 0$).

with constraints:

$$c_1, c_2, \dots, c_n \geq 0 \tag{5.2}$$

$$\sum_{i=0}^{n} c_i = N \tag{5.3}$$

where N is the total number of operations to be computed (i.e., $m * n * k$). To calculate the copy time function (y_x), we first start by computing bytes to be transferred (B) as:

$$B = dt_x * (mk + kn + mn) \tag{5.4}$$

where dt_x is the data type size in bytes and m, n, k are the matrix dimensions. When distributing the matrices across devices, we only vary m (see Section 5.2.2.3). Then, we can find the relationship of bytes copied with the number of operations (c_x) by substituting m in the previous equation:

$$B = dt_x * (\frac{c_x}{nk}k + kn + \frac{c_x}{nk}n) \tag{5.5}$$

if we simplify and account for the memory bandwidth (bw_x), we get:

$$y_x = \frac{dt_x * (c_x \left(\frac{1}{k} + \frac{1}{n}\right) + kn)}{bw_x} \tag{5.6}$$

Equation 5.6 gives the time to copy A, B and C matrices, assuming that the communications happen in a bus exclusively used by device x. This is true when only one device is connected to the bus but is not realistic in a shared bus (e.g., in a SoC). If memory copies of different devices are serialized, the function must take into account the time to copy the data of previous devices of A, B or C matrices. We have more than one accelerator connected to the same bus in our target platform, so we adapt Equation 5.6 to reflect that.

We implement the MILP problem using CPLEX 12.10 [82]. The CPLEX solver is embedded in the framework using the CPLEX API, and the MILP formulation is defined dynamically, depending on the devices being used. When the model has been optimized, the output variables of the MILP solver are $c_1, c_2, ... c_n$, which represent the number of operations to compute by each device.

5.2.2.3 Adapt (GEMM)

The optimized values given by the MILP solver in the previous phase are the number of operations, while the scheduler needs values for m, n and k. Therefore, in this phase, the number of operations is converted into matrix shape values, so they can be used by the scheduler. For this task, we designed an algorithm called `ops_to_mnk` that works on both data and hardware adjustments.

Data adjustments. Regarding data adjustments, the `ops_to_mnk` algorithm must accomplish two tasks:

1. Find m, n and k such that the number of operations matches the operations given by the MILP solver. This gives the m, n and k dimensions for each device.
2. Express the global matrix product as a list of squared sub-matrices products (in a best-effort manner). This divides the m, n and k dimensions for each device into sub-matrices for precise performance prediction.

For the first task, we start setting n and k to their original values. Partitioning a matrix with a different value of n would provide partial results in the output C matrix, so we fix n for convenience. Setting k to the original value makes the `ops_to_mnk` algorithm easier since just the rows of A (m) must be distributed. Then, to map ops to mnk, only m has to be determined, which is computed as $m = \frac{ops}{n*k}$.

For the second task, the algorithm must ensure that resultant matrices are as squared as possible (best-effort). Having squared matrices is the optimal scenario, as we would be performing the matrix multiplications in the same way as the profiling phase. But it can only be accomplished if the input size is divisible by the sub-matrix sizes, which is not always possible. However, matrices that are very close to being squared (e.g., $m = 1.1k$) can also be predicted with very high precision. Let us denote with an apostrophe the dimensions of the submatrix (e.g., k') and without it, the dimensions of the original matrix (e.g., k). The algorithm tries to make m' and k' as similar as possible while keeping n' equal to n. Our algorithm always ensures that the number of horizontal dimensions in A fits perfectly (i.e., $k \,\%\, k' == 0$). Without such restriction, "gaps" may appear in the last column of A. Therefore, the search space in k' is restricted to the divisors of k, which happens to be big enough when the input matrix is also big. For determining m' size, the algorithm iterates over all the possibilities, analyzing how "squared" the resultant matrices using a simple heuristic would be. For a given list of squared matrices with $\{m'_1, m'_2, \dots, m'_n\}$ and $\{k'_1, k'_2, \dots, k'_n\}$, the squareness ($sq$) is computed as:

$$sq = \sum_{i=0}^{N} \left(\frac{min(m'_i, k'_i)}{max(m'_i, k'_i)} * m'_i k'_i n \right) \tag{5.7}$$

This value represents how squared the global set of sub-matrices is. Thus, to find the best sub-matrix distribution, the algorithm chooses the one that maximizes the value of the heuristic.

Hardware adjustments. The `ops_to_mnk` algorithm asserts that the matrix sizes satisfy the requirements imposed by the hardware to achieve optimal performance. In our case study, we consider CPUs, GPUs and tensor cores, so the `ops_to_mnk` algorithm must meet two additional requirements:

- Tensor Cores: To reach optimal performance, the input sizes must meet the following conditions: $m \,\%\, 8 == 0$ and $k \,\%\, 8 == 0$ [140]. To do so, the algorithm reduces the input size until it meets the desired requirements. In the end, this means that the tensor cores get fewer operations than the MILP solver specified, but this is barely noticeable since the size reduction is tiny compared to the global size.
- CPU cores: When profiling the CPU, inputs are designed to fit into cache memory. Therefore, when a real workload arrives, the algorithm must ensure that the generated submatrices also fit into cache.

5.2.2.4 Scheduler (GEMM)

For the scheduler, we use a static scheduling approach, as we found that gives excellent results for our case study. In other words, the scheduler receives the matrix sizes for each device and does not change them over time. We explore some of the possible issues of this approach in Section 4.3.4.3.

Regarding the shared PCIe bus, we use a priority scheduling approach. When the program reads the configuration file, it assigns a priority for each device: the faster the device, the higher priority. Then, A and B matrices are copied in the order established by the priority. Thus, lower-priority accelerators remain idle while the higher-priority devices are copying the data. After the computation, the first device (meaning the faster one) copies C to the host, and the same order is used to copy the remaining parts of C. In this case, higher idle

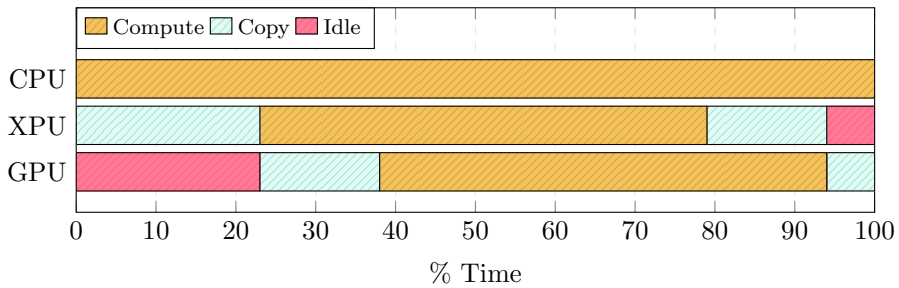


Figure 5.3: Proposed scheduling communication scheme in a shared bus with CPU+GPU+XPU.

times are experienced from high-priority devices, which have to wait for the rest of the devices to complete. Figure 5.3 shows the proposed communication scheme.

5.2.2.5 Convolution

Predict. Similarly to how we divided matrix multiplication by the number of rows in matrix A, we look for a way of dividing a convolution workload to distribute it among the compute elements. We decide to divide convolutions by the minibatch size, which is a common technique in distributed and parallel approaches [20, 148]. Thus, in the profiling phase, convolution is measured by varying all parameters (image sizes, number of filters, filter sizes) except for the minibatch size, which we restrict to a reduced set. Again, we detail values for this set in Section 5.3.1.1. We perform the profiling phase complying with the convolution tensor core restrictions. First, C and N must be multiple of 8 [141]. Second, the 4D tensors layout must be NHWC [144]. For simplicity, we use no padding and a stride of 1.

Optimize. We follow a similar formulation to the one shown in matrix multiplication, where we express the time with respect to the number of operations. Naturally, we have to compute the number of operations for convolution, which is [18]:

$$ops = K_h * K_w * C * H_{out} * W_{out} * K \tag{5.8}$$

where K_h and K_w are the height and width of the filters, C is the number of channels, H_{out} and W_{out} are the height and width of the output image, and K is the number of filters. In our problem formulation, we change this formula to fit our particular needs. First, we observed that DNN implementations are typically parallelized over the number of filters (K), meaning that the execution time is invariant to K (when the filter sizes are small enough). Second, we must account for the number of minibatches in the formula. Therefore, we use the following expression:

$$ops = K_h * K_w * C * H_{out} * W_{out} * N \tag{5.9}$$

where N is the number of minibatches. Likewise, we compute the memory copy function following the same approach as in matrix multiplication. For example, the bytes to be copied (B) for the input image is:

$$B = dt * N * C * H * W \tag{5.10}$$

$$= dt * \frac{c_x}{K_h * K_w * C * O_{out} * W_{out}} * C * H * W \tag{5.11}$$

where, again, c_x and dt are the number of operations and the size in bytes of the data type, respectively.

Adapt. In convolution, we also need to adapt the optimized values (e.g., transform operations into convolution shapes). We implemented a straightforward algorithm called `ops_to_batches` that simply computes the number of mini-batches (N) of each device as:

$$N = \frac{ops}{K_h * K_w * C * H_{out} * W_{out}}$$

The algorithm also ensures that the XPU input sizes passed to the scheduler have N and C multiple of 8.

5.2.2.6 Implementation Details

When tensor cores are used, the output of the matrix multiplication comes in half-precision (FP16), while the CPU can only perform the product in FP32. Therefore, when the results are collected in the CPU, the product has mixed-precision results. In this work, we do not consider how to deal with this problem as it is out of the scope of the research, but it is worth mentioning that related work in this field has shown promising results [145].

In our POAS implementation, we copy the data between the CPU and GPU asynchronously. However, the GPU does not start computing until the whole data stream is copied. This simple approach could be improved using CUDA streams and overlapping the computation with memory copies. In either case, the performance predictor can be adapted to predict the memory copies with or without overlap. Therefore, for our study, it is not particularly relevant whether the implementation copies the data with or without overlap.

5.3 Evaluation

We evaluate POAS using matrix multiplication and convolution applications. Section 5.3.1 details our hardware and software configuration. In Section 5.3.2, we analyze the prediction accuracy of POAS. Lastly, we evaluate the performance of POAS in Section 5.3.3.

5.3.1 Test Bed

Hardware and software configuration. The evaluation platform is equipped with mach1 and mach2, two HPC servers with a CPU+GPU+XPU configuration. During this evaluation, we refer to an XPU as a GPU that uses the tensor cores to perform the matrix multiplication, whereas GPU uses traditional CUDA cores. The hardware configuration of both machines is summarized in Table 5.1, and the specifications for each device are detailed in Table 5.2.

Both systems run CentOS 8.2 (4.18.0-193 kernel in mach1 and 4.18.0-348 in mach2). We build POAS using g++ 8.4.1. Table 5.3 summarizes the libraries that POAS relies on to run the workloads. Regarding the communication between CPU and GPUs, the RTX 2080Ti's in mach1 are connected to a PCIe 3.0 x16 bus, which peak memory bandwidth is 15.75 GB/s. In mach2, both cards are connected to a PCIe 4.0 x16 bus, providing a peak memory bandwidth of 31.75 GB/s. Since the RTX 2080Ti supports up to PCIe 3.0, the card in mach2 works in 3.0 mode, even though it is connected to a 4.0 slot. For the convolution, we use the `CUDNN_TENSOR_NHWC` tensor format, as it is the optimal format for tensor cores [144]. For the experiments, we reserve one physical CPU core for managing the GPU and XPU. Henceforth, mach1 has 5 physical cores and mach2 has 23 cores to run the CPU workloads.

Table 5.1: Hardware configuration for the testbed environment.

	CPU	GPU	XPU
mach1	Xeon v3	RTX 2080 Ti	RTX 2080 Ti
mach2	AMD EPYC	RTX 3090	RTX 2080 Ti

Input sizes. For matrix multiplication, we conceive six different matrix sizes (shown in Table 5.4) sorted in descending order by the number of operations (TOps). We are interested in evaluating relatively small matrices, like the first two inputs, as well as squared and non-squared matrices. We also want to study very skinny matrices like input 3, where the m dimension is much larger than the others. The same idea is explored for n and k dimensions in inputs 4 and 5. Those inputs are useful to understand how solid the predictor is because they allow us to see if the predictor performs well on non-square and skinny matrices.

For convolution, we design four inputs (shown in Table 5.5) based on real CNN workloads [183]. Inputs 1 and 3 are representative of the ResNet 50 ar-

Table 5.2: Hardware specifications for the testbed environment.

Model	CPUs		GPUs / XPUs	
	Intel Xeon E5-2603 v3	AMD EPYC 7413	NVIDIA RTX 2080 Ti	NVIDIA RTX 3090
Architecture	Haswell	Zen 3	Turing	Ampere
Technology	22nm	7nm	12nm	8nm
CPU cores	6	24	-	-
CUDA cores	-	-	4352	10496
Tensor cores	-	-	544	328
Max. Frequency	1.6 GHz	3.6 GHz	1.5 GHz	1.6 GHz
TFLOP/s (FP32)	0.307	2.76	13.45	35.58
TFLOP/s (FP16)	-	-	107.5	284.65
LLC	15MB	128MB	6MB	6MB
Memory size	64GB	512GB	11GB	24GB

Table 5.3: Libraries used in each platform for matrix multiplication and convolution.

	CPU	GPU
mach1	MKL 2020.0.2 oneDNN 1.96.0	cuBLAS 11.2.0 cuDNN 8.0.5
mach2	AOCL BLIS 3.1 oneDNN 1.96.0	cuBLAS 11.8.0 cuDNN 8.4.1

chitecture, while inputs 2 and 4 are based on AlexNet.[1] Due to memory size limitations, inputs 1 and 2 are executed only in mach1, and inputs 3 and 4 are run in mach2, which has a notably bigger GPU memory size.

For each input, we repeat the computations 50 times, therefore executing 50 matrix multiplication and convolutions over the accumulated data. We run each input three times, and the values shown are the average over these three independent runs. We used CUDA event sampling for precise time measurements on the GPUs.

[1] In all cases, we reduced the number of filters due to GPU memory limits.

Table 5.4: Matrix sizes used in the evaluation.

Input	*m*	*n*	*k*	TOps
1	30K	30K	30K	27.0
2	60K	20K	35K	42.0
3	130K	20K	20K	52.0
4	40K	80K	20K	64.0
5	40K	30K	60K	72.0
6	56K	40K	40K	89.6

Table 5.5: Convolution inputs used in the evaluation.

Input	*n*	*c*	h	w	k	kh	kw
1	3500	16	224	224	16	7	7
2	4000	16	227	227	16	11	11
3	3000	32	224	224	16	7	7
4	2500	32	227	227	16	11	11

5.3.1.1 Profiling Configuration

In matrix multiplication, the profiling phase performs 30 squared matrix products with matrix sizes ranging between 1000 and 2000 for the CPU and between 3000 and 6000 for GPU/XPU. For the generation of the list of squared sub-matrices, they are restricted to be of a size such that the number of operations are between the same number of operations that were performed during profiling. In other words, in the CPU, the sub-matrices are restricted to 1000 x 1000 x 1000 (10^9) and 2000 x 2000 x 2000 ($8 * 10^9$) operations, and in GPU between 3000 x 3000 x 3000 ($27 * 10^8$) and 6000 x 6000 x 6000 ($216 * 10^8$) operations. Thus, sizes are computed on the fly depending on the size of n in the original matrix.

In convolution, the profiling phase performs a set of convolutions with a minibatch size of 8, 128 and 256 for CPU, GPU and XPU, respectively. Similarly to matrix multiplication, those sizes are the same as the minibatch size used in real workloads.

5.3.2 Prediction Accuracy

To evaluate the performance predictor used in POAS, we study the prediction accuracy. We measure and compare the execution and memory copy times with the predicted values. Then, we calculate the prediction error e as an expression

Table 5.6: Root mean square error (RMSE) and prediction error for GEMM in mach1 and mach2. The compute (COM) error and RMSE are shown for CPU, whereas the error and RMSE are divided into computing (COM) and memory copy (CPY) for GPU and XPU (in parentheses), along with the global error (GLB) and RMSE.

Input	mach1			mach2		
	CPU	GPU	XPU	CPU	GPU	XPU
	COM	GLB (COM, CPY)	GLB (COM, CPY)	COM	GLB (COM, CPY)	GLB (COM, CPY)
1	4.5%	1.6% (8.8%,5.3%)	0.7% (3.2%,1.2%)	1.0%	4.6% (9.1%,0.0%)	4.7% (10.1%,1.2%)
2	1.4%	2.9% (5.1%,0.6%)	3.1% (6.1%,0.1%)	0.5%	1.6% (2.9%,0.0%)	6.1% (11.6%,1.0%)
3	3.1%	0.7% (0.8%,2.0%)	3.3% (6.2%,0.6%)	0.4%	1.6 %(3.3%,0.0%)	7.4% (12.2%,0.9%)
4	4.6%	9.9% (5.3%,14.4%)	5.3% (6.1%,4.2%)	2.0%	2.0% (3.9%,0.1%)	4.6% (7.8%,1.3%)
5	2.4%	6.9% (11.9%,0.0%)	3.0% (6.6%,0.1%)	1.3%	5.8% (9.9%,0.1%)	5.4% (10.7%,1.1%)
6	0.8%	6.5% (6.7%,6.2%)	3.4% (5.0%,1.4%)	3.6%	4.1% (6.8%,0.0%)	6.7% (11.1%,1.1%)
RMSE	2.42	5.63 (3.55,3.10)	3.13 (2.64,0.76)	1.69	2.85 (2.87,1.68)	4.42 (4.73,0.32)

Table 5.7: Root mean square error (RMSE) and prediction error for convolution in mach1 (above) and mach2 (below). The compute (Comp.) error and RMSE are shown for CPU, whereas the error and RMSE are divided into computing and memory copy for GPU and XPU (in parentheses), along with the global error and RMSE.

Input	CPU	GPU	XPU
	COM	GLB (COM, CPY)	GLB (COM, CPY)
1	0.2%	1.7% (0.0%,2.3%)	1.2% (1.5%,1.0%)
2	6.3%	0.3% (2.9%,3.1%)	0.3% (0.4%,0.2%)
RMSE	2.19	0.36 (0.47,0.66)	0.28 (0.16,0.12)

Input	CPU	GPU	XPU
	COM	GLB (COM, CPY)	GLB (COM, CPY)
3	3.5%	0.0% (0.6%,0.1%)	0.5% (0.2%,0.9%)
4	1.8%	0.5% (1.9%,0.1%)	1.7% (1.8%,1.6%)
RMSE	0.87	0.16 (0.17,0.02)	0.44 (0.23,0.23)

of the relative error: $e = 100 * \frac{v - v_{pred}}{v}$, where v is the measured time in our experiments and v_{pred} is the value given by the predictor. We also compute the root mean square error (RMSE), which gives a general perspective of the prediction robustness across different inputs.

Table 5.6 and 5.7 shows the prediction error and root mean square error (RMSE) for GPU and XPU, where we show the global prediction error (and RMSE) in the first instance, followed by the computing and memory copy prediction error (and RMSE), respectively. Overall, we observe that the prediction error is low (typically, under 5%). This is a key factor to provide high-quality co-execution because otherwise, the load imbalance would be very high, leading to substantial performance degradation. Except for a few cases, the memory prediction error is very low, especially for mach2, whose prediction is close to being perfect. Some inputs are predicted with slightly higher prediction error ratios than the mean (e.g., the latest ones in the GPU and XPU in mach1). In fact, these "outliers" are the main fact that increases the RMSE of the whole evaluation. We believe that these observations are caused by high temperatures, which cause overheating. During the profiling phase, we leave all the device's frequencies unlocked. Because the profiling phase is relatively short, the device does not get significantly hotter than the idle temperature. However, in real workloads, the temperature can increase much more, downscaling the clock frequency to avoid overheating. In other words, the measured frequency in the profiling phase may not match the frequency used in real workloads. This is especially true for mach1 since it has substantially worse heat dissipation capabilities than mach2.

Regarding RMSE, POAS achieves very low values for both use cases, which confirms the great robustness of the predictor, despite the use of static scheduling. However, a more sophisticated solution could employ a dynamic scheduler that considers the frequency in real-time of every device and dynamically balance the workload to further improve accuracy. In either case, POAS fully adapts to the underlying hardware, properly exploiting its computing power. Based on our results, we can confirm that POAS is able to efficiently exploit ALP in CPU+GPU+XPU environments for matrix multiplications and convolutions.

5.3.3 Performance

We show the workload distribution used by POAS in Figure 5.4. As we can see, the CPU provides little help in computing the matrix multiplication (especially in mach1, where it gets less than 1% of the work), while the GPU takes between 20% and 30% of the work. Because matrix multiplication is a very compute-intensive workload, the communication penalty between the CPU and the accelerators is smaller than the higher computational power of the accelerators. On the other hand, convolution has a lower arithmetic intensity, so it is easier for the CPU to contribute more work than in matrix multiplication since memory copy overhead is bigger in the accelerators. Hence, we can observe that

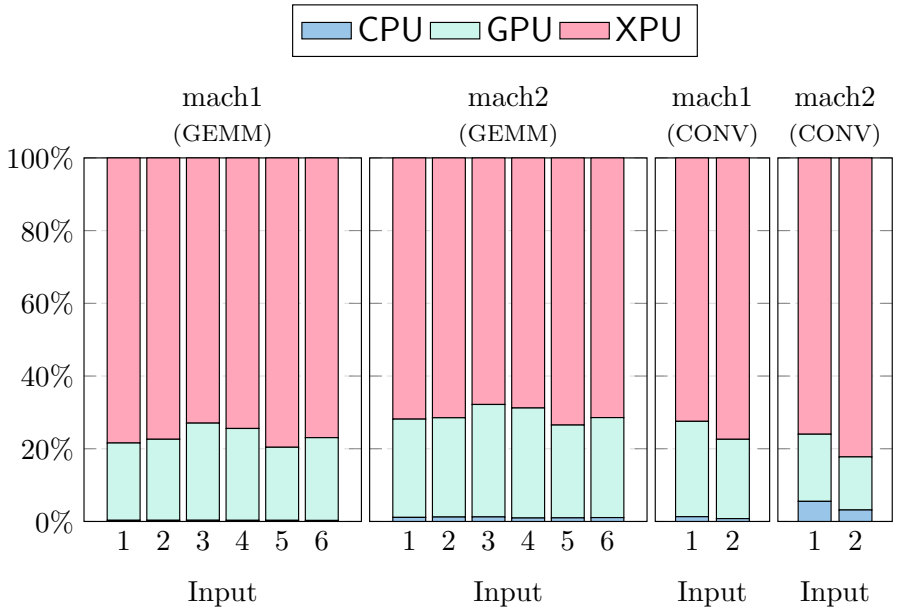


Figure 5.4: Percentage of work distribution among devices in mach1 and mach2 for GEMM and convolution.

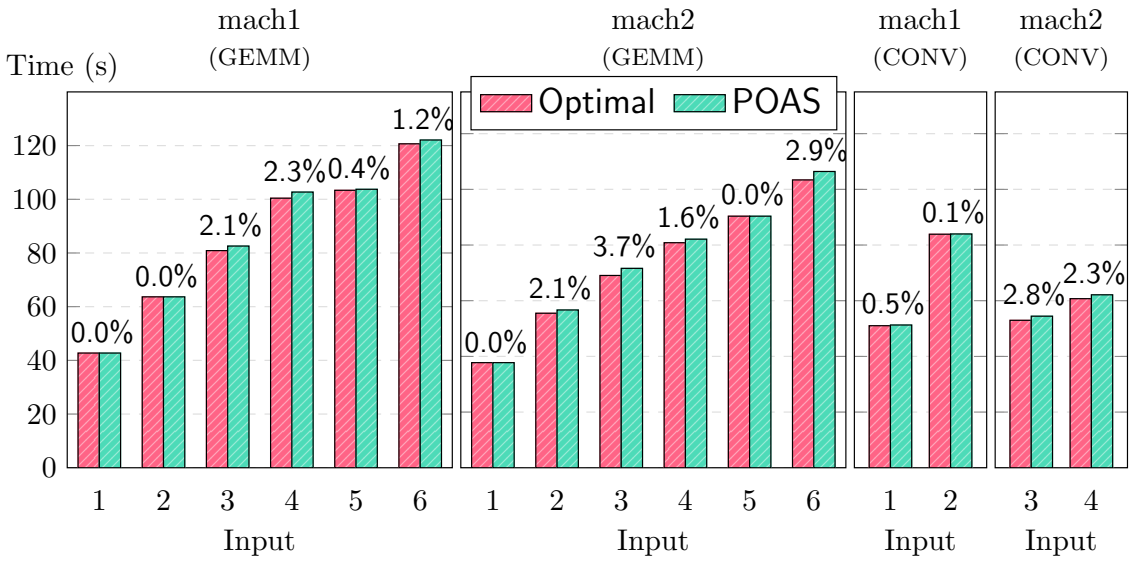


Figure 5.5: Runtime comparison of POAS implementation for GEMM and convolution against optimal distribution.

the CPU participates more in both machines and the GPU participates less since

the shared bus is occupied more often by the XPU, which has a higher priority in accessing the PCIe bus.

Now, we compare POAS execution time against the optimal work distribution. To find the optimal distribution, we explore by hand all the plausible work distributions and selected the one that resulted in the minimum execution time. Figure 5.5 shows that POAS distribution was very close to the optimal in both machines and both applications.

The difference between the POAS and optimal distributions comes from two factors. The first one is prediction errors, which we already studied in depth. The second one is load unbalance in the POAS work distribution. Even though POAS aims to distribute evenly the work among devices, this is not always possible. Sometimes, inputs must be divided in non-even distributions to make sure that accelerators receive the optimal input size (we discussed this in Section 5.2.1.3). This division unbalances the distribution because other devices must take the remainder work from the accelerator, or do less work since it is now done by the accelerator. In any case, this second factor has less influence than the prediction error. However, we observe that POAS tends to give excessive work to the CPU in mach2. A small work excess in the CPU has a bigger impact because it is more sensible to execution time variations. This explains why POAS is closer to optimal in mach1.

In any case, POAS finds the optimal distribution in four cases and has a performance loss against optimal distribution smaller than 2% for nine inputs. It performs better in mach1, where perfectly even distributions are the norm. Further tuning in prediction (dynamic scheduling) or work distribution might improve overall performance, but since the limit is very close, the margin to improve performance is quite small.

5.3.4 Performance Analysis

Compared to our previous works like HDNN or ATC, POAS unlocks a new level of performance: Accelerator-Level Parallelism (ALP). POAS itself is not a language, nor a compiler, so it cannot be directly compared our previous approaches. Nonetheless, POAS could be integrated within a single-source language or as a middleware layer in the OS, providing ALP to existent works. If we scale our previous plots to consider performance within an accelerator and to also consider performance when exploiting ALP, POAS would perform like shown in Figure 5.6. Again, we plot POAS with dotted lines to highlight that POAS itself is not a language, but could be integrated with existent approaches to provide that combined productivity, portability and performance. In this figure we consider that POAS is paired with ATC and another state-of-the-art

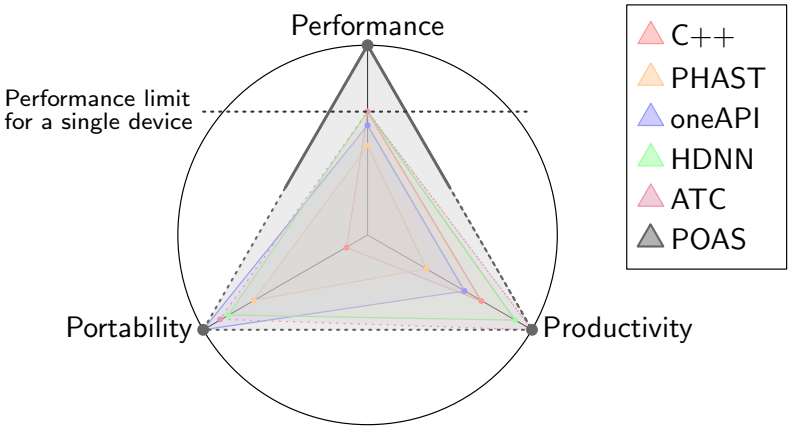


Figure 5.6: P^3 analysis of POAS compared to other approaches.

language. In fact, we believe that POAS is the last piece to complete our P^3 equilateral triangle, providing optimal productivity, portability and performance.

5.4 Related Work

5.4.1 Scheduling

Task scheduling techniques have been proposed for OpenCL kernels in [191], where authors use both code's features as well as runtime ones to predict the speedup of applications in CPU or GPU. Also in OpenCL, non-analytical methods like decision-based trees are used in [190] to schedule OpenCL kernels on CPU/GPU platforms. Co-execution opportunities are studied in [202] on integrated CPU/GPU architectures. They also studied how to determine which compute elements are suitable or not for a given task (in other words, when co-execution is beneficial or not). List scheduling has been applied in both static [206] and dynamic runtime scenarios, where new workloads arrive over time [116]. Profiling and machine learning were combined in [66] to provide scheduling in heterogeneous environments. Integer linear programming (ILP) and linear regression were combined with stream graphs in [133] to efficiently distribute workloads on multi-GPU platforms. Performance modeling has been widely applied in many works [176, 147, 157, 61]. In a DynamIQ heterogeneous multi-core environment, a performance model to estimate the efficient distribution of critical sections was designed [147]. Task scheduling has been often applied to CPU/GPU environments, but there are also other approaches for more heterogeneous environments, like CPU/FPGA [167]. In [200], authors proposed

a scheduling strategy for distributed accelerator-rich environments centered in real-time applications. The predictable execution model (PREM) [157] was proposed to enable time prediction on non-predictable hardware. The approach separates programs into memory and compute phases, which can be independently scheduled. It was proposed for CPU only, but a recent work extended it for CPU/GPU architectures [61]. Many of these works focused primarily on minimizing execution time, while others studied energy consumption. Although the latest is often harder to predict, there are some promising works in this field [60]. Given the heterogeneous nature of today's computing systems, other studies considered both execution time as well as consumption in their scheduling decisions [158, 163].

5.4.2 Domain-Specific Scheduler Proposals

General matrix multiplication is a topic that has been deeply studied over time, mainly due to its high relevance in many computer science applications. Recent works have studied the performance of matrix multiplication in heterogeneous environments [173]. Furthermore, several papers have considered the use of different hardware devices to compute matrix multiplications to exploit heterogeneous systems. One of the first studies [19] already approached the problem from an analytical point of view. The authors analyzed the computational power of each processor in the heterogeneous system and later expressed the workload distribution as an optimization problem. As the concept of heterogeneity has evolved, that work was targeted to distributed systems using MPI, which imposed different issues to be solved. In [24], authors designed a hierarchical approach to be able to distribute parts of the matrix multiplication to different devices. When considering multiple accelerators and a range of n columns to be assigned to each accelerator, the search space becomes too big. Therefore, they proposed a hierarchical way of considering all the possibilities, significantly reducing the search space. A new algorithm based on Strassen's method was presented in [97] for heterogeneous environments. To schedule the work between accelerators, a queue-based system was used, which gives blocks of the matrices to be computed whenever a device is free. Matrix multiplication workload distribution has also been studied in the context of energy efficiency [25], where authors proposed an approach for ARM big.LITTLE processors. One of the centric ideas was to study the performance ratio between the big and the little cores in the SoC and use such criteria to perform the scheduling. Despite the large amount of related work in heterogeneous matrix multiplication, there is no previous work that exploits ALP.

In the field of convolutional networks, [197] proposed a pipeline-based scheduler for heterogeneous CPUs. The inference performance of a hybrid CNN/RNN model is improved in [81] also using a pipeline scheduling scheme, as well as a fine-grained scheduling scheme in CPU-GPU environments. Pipeline scheduling is also used in [188] to improve inference in ARM big.LITTLE scenarios. Lastly, [40] presented an approach to improve the energy efficiency of neural network inference using constraint-based optimization using GPUs and deep learning accelerators.

5.4.3 Multi-Domain Scheduler Proposals

Recently, several works have focused on designing frameworks and systems to co-execute applications without domain-specific information. Many are often targeted to specific frameworks or languages that enable single-source coding on heterogeneous platforms. A language that is gaining influence lately is oneAPI [87]. oneAPI, as well as other heterogeneous languages, typically achieve good performance in relevant applications like DNNs [119, 120]. However, oneAPI does not officially provide a mechanism for scheduling or co-execution. In a recent research [136], authors proposed a new co-execution runtime in oneAPI based on load-balancing algorithms. Another relevant framework in this context is OpenCL, which was also coupled with a co-execution engine in [137]. In [162], authors extend the OmpSs framework to allow co-execution of OpenCL kernels. Lastly, a Python-based heterogeneous scheduler was proposed in [110], which objectives are similar to what POAS pursues. It uses task parallelism and a queue-based approach to schedule programs in multi-GPU environments.

5.5 Conclusions

Heterogeneity is becoming increasingly common in all scopes. Energy-constrained systems benefit from accelerators thanks to their lower consumption while high-performance systems also take advantage of massive performance improvements in compute-intensive workloads. To exploit heterogeneity, Accelerator-Level Parallelism (ALP) is a promising approach. Running concurrently any workload in multiple devices requires dividing the work wisely between the compute elements. As the number of applications in which accelerators are used is growing quickly, we need solutions that allow this process to be performed efficiently.

This chapter has presented POAS, a framework for scheduling workloads among the heterogeneous compute elements available within a node. The framework adapts to the software libraries and hardware, maximizing resource usage. We applied our framework to linear algebra and deep learning fields, showing that POAS can fully exploit ALP to improve overall performance. Specifically, our proposal has a deviation of only 1.3% and 1.4% from optimal performance for GEMM and convolution, respectively. Furthermore, the framework is fully extensible to other applications and domains.

In recent years, the number of accelerators within a chip is growing. Likewise, ALP relevance is also increasing in a similar manner. Then, we believe that systems that exploit ALP as POAS does will be a crucial component of future computing systems, either implemented within the OS, integrated with the compiler, etc.

Chapter 6

Conclusions and Future Ways

6.1 Conclusions

Like multicores did, heterogeneous computing is increasing the complexity of software development and making the architecture of computers more and more complex due to the diverse hardware variety. All computer architecture advances have come with increasing hardware complexity, which we must tame to make computers practical and useful. New architectures, different from the long-lasting CPU, bring unprecedented levels of performance and energy efficiency.

We are constantly looking for better ways of managing heterogeneity. The first problem we have to exploit accelerators is finding a way of making software development practical. Each accelerator needs different programming languages to be used, so developing as many code versions as accelerators is extremely costly, it is not practical. Second, even if we find a convenient way of programming accelerators easily, there is a tremendous code base that is already written. If new code must be developed, new languages can tackle the problem of heterogeneity, but with written code the problem is different. Novel techniques allowing the compilation of code written for CPU to accelerators could increase performance by several orders of magnitude without user intervention. Third and last, we are observing that computers have more and more dedicated accelerators for different tasks. SoCs are populated with a great amount of diverse hardware, and this number is expected to increase in the future. The hard part of having many dedicated accelerators is how to use them efficiently and how to reduce idle time. Accelerator-Level Parallelism (ALP) is said to be

the next computer architecture paradigm, facing this challenge. However, it is not clear how ALP can be exploited to increase the overall performance and/or energy consumption of future systems.

In this thesis, we have focused on solving these three main issues in heterogeneous computing. The main contributions of this thesis are:

- **We have shown that performance portability is possible with single-source languages, as well as a novel DSL for DNNs that achieves excellent performance, productivity and portability in heterogeneous environments (Chapter 3).** Porting the Caffe framework with the PHAST library and some Caffe layers with oneAPI has shown that performance portability is achievable in real-world scenarios. Indeed, languages and frameworks are improving, and new technologies appearing, enhancing the portability and productivity of software development in heterogeneous environments. However, we found that many of these technologies are often hard to use, or hard to achieve good performance portability. Thus, we proposed Heterogeneous Deep Neural Networks (HDNN), a heterogeneous, Domain-Specific Language for DNNs. Compared to others, HDNN programs are shorter. In addition, HDNN is based on MLIR and a novel methodology for lowering the DSL code to the specific accelerators which provide near-optimal performance and close to zero overhead.

- **We have designed a novel methodology for detecting and compiling acceleratable parts of CPU code to specialized hardware accelerators automatically (Chapter 4).** We have proposed Algebra and Tensor Compiler (ATC), a compiler that replaces acceleratable C/C++ code with calls to accelerators APIs, supporting matrix multiplication and convolution programs. First, the pre-trained neural classifier finds the acceleratable functions of the program. Then, the compiler performs an IO-based synthesis where all the candidates are tested against valid APIs to test for behavioral equivalence. Because the combinatorial space is huge when mapping the variables between two complex functions, we have proposed different techniques to make this space tractable. ATC is completely automatic thanks to a novel technique that is able to detect the livein and liveout variables of a function, as well as the dimension of the matrices, even in complex, non-structured codes. Besides, ATC features an SVM-based classifier used to only offload the workload to the accelerator if it is profitable; otherwise, the workload is executed in the CPU-optimized library. ATC is able to detect between 2.6 and 7x more programs than state-of-the-art approaches, resulting in more than an order of magnitude performance improvement.

We also believe that the ATC methodology is very solid and could be easily applied to many other domains apart from GEMM and convolution.

- **We have proposed a framework for exploiting Accelerator-Level Parallelism in heterogeneous environments (Chapter 5).** The hardware for exploiting ALP is already in our phones, and soon will be more common in other devices such as laptops and servers. But the software stack needed to exploit this new generation of hardware is not yet ready. Therefore, we have proposed Predict, Optimize, Adapt and Schedule (POAS), a framework for efficiently exploiting ALP. We assumed an environment where all the devices are idle and do not have any other work to do. POAS is able to run an application in all the available hardware within a system, reducing the execution time and/or the energy consumed by the application. Our framework divides the task of scheduling applications into four different steps. We developed an implementation of POAS that schedules GEMM and convolution kernels in environments with CPU, GPU and XPU (tensor cores). Overall, we demonstrate that the workload distribution found by POAS has a deviation of only 1.3% from the optimal. We believe that POAS is an excellent candidate to reach ALP in future computer systems.

6.2 Thesis Contributions

The following subsections summarize the contributions of the thesis.

6.2.1 Productivity and Performance Portability in the Heterogeneous Era

We explore ways to achieve performance portability with single-source programming languages. First, we study Caffe, a machine learning framework that is implemented in C++/CUDA. We port the whole framework to PHAST, a single-source library for heterogeneous programming that supports CPUs and GPUs. Second, we studied the use of oneAPI, an SYCL-based language for single-source programming made by Intel. Instead of implementing the full framework, we developed only some layers of the Caffe framework. We learned that performance portability with these languages is possible, but it is very hard for the programmer to achieve. Thus, we propose Heterogeneous Deep Neural Network (HDNN), a heterogeneous Domain-Specific Language (DSL) for DNNs. Our language supports CPUs, GPUs and TPUs, the accelerator for deep learning

designed by Google. Unlike PHAST and parts of DPC++, HDNN does not compile the user code into the appropriate accelerator. Instead, it simply replaces the high-level DNN intrinsics in the language with a call to the accelerator API. The HDNN backend is implemented in MLIR and has a negligible performance impact. Since the accelerator APIs often provide optimal performance, HDNN achieves excellent performance on all accelerators with very little programming effort.

6.2.2 Compiling Existent Code to Accelerators

The downside of single-source languages is that they force developers to rewrite code which, in large code bases, is sometimes impractical. Therefore, we propose a novel technique based on program synthesis to automatically compile C/C++ programs to hardware accelerators. Our compiler, called Algebra and Tensor Compiler (ATC), can compile matrix multiplication and convolution programs to the tensor cores and TPU, respectively. ATC is in fact the first compiler that can match and replace codes without regard of the code length or complexity. Unlike other proposals, ATC can match GEMM programs as easy as the naive 3-loop structure, and as complex as the Strassen matrix multiplication. We achieve this with a novel methodology called IO synthesis. Rather than relying on code structure to guide detection, ATC uses behavioral equivalence to determine if a section of code is a linear algebra operation. ATC outperforms all previous work in matching programs and overall performance. Furthermore, it also presents an automatic performance predictor that decides at runtime when the accelerator is slower than the CPU code. In that case, the accelerator is not used, improving the program performance for small inputs.

6.2.3 Exploiting Accelerator-Level Parallelism

Once we are able to compile programs to accelerators automatically and achieve excellent performance on them, we seek new ways of improving overall performance. We believe that one promising way is by exploiting Accelerator-Level Parallelism (ALP), which consists of using multiple accelerators concurrently. We present Predict, Optimize, Adapt and Schedule (POAS), a novel framework for exploiting ALP in heterogeneous environments. Our approach consists of analyzing the application performance on a given hardware and defining a mathematical model that faithfully represents that behavior. The model can later be optimized to find the optimal work distribution among different accelerators. The work distribution can pursue two different objectives: to reduce power consumption and/or to reduce execution time. Precisely, we apply POAS to matrix

multiplication and convolution applications using tensor cores. POAS divides the work evenly among the CPU, GPU and GPUs with tensor cores, achieving work distributions almost identical to optimal.

6.3 Publications

The research produced during this thesis is either published in relevant journals and conferences or is currently under review. We present all of these works and briefly outline them, detailing the section to which each corresponds.

6.3.1 Refereed Journals and Conferences

- Our first work [120] is published in the *International Journal of High Performance Computing Applications* journal (Q2). This paper covers the use of PHAST, a hardware-agnostic library to implement Caffe, a machine learning framework. We accomplished this research in collaboration with Biagio Pecerilo and Sandro Bartolini, the authors of PHAST, from the University of Siena. This work is detailed at the beginning of Chapter 3.

- Following a similar approach, we used oneAPI, a single-source language developed by Intel to implement some layers within Caffe [119]. This work is published in the *Concurrency and Computation: Practice and Experience* journal (Q3). We summarize our findings of this paper in Chapter 3.

- Our proposal of a Domain-Specific Language for deep neural networks [118] is published in the *Journal of Supercomputing* (Q2). This work corresponds to the end of Chapter 3.

- In the *32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23)*, we presented ATC [122], our proposal for automatically compiling C/C++ codes to accelerators. This work is the result of a collaboration with Michael O'Boyle during my stay at *The University of Edinburgh*. We detail this research in Chapter 4.

- Our proposal for exploiting Accelerator-Level Parallelism, called POAS [117], is currently under review in the *IEEE Transactions on Parallel and Distributed Systems* (TPDS) journal. This work is detailed in Chapter 5.

6.3.2 Other presentations

- In the *13th International Workshop on Programmability and Architectures for Heterogeneous Multicores*, which was held held in conjunction with the *15th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, we presented our first version of Caffe implemented using the PHAST library [76].
- In the *21st Workshop on Compilers for Parallel Computing (CPC 2021)*, we presented "Towards an Efficient Unified Programming Model for Heterogeneous Computing", a high-level vision of a heterogeneous scheduler for exploiting co-execution opportunities. This work presented the first ideas, which later evolved into the POAS framework [117].
- In the *21th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE)* we presented a work called in a work called "Achieving native performance with a simple heterogeneous programming framework based on LLVM". This presentation was our first approach to heterogeneous compilation using LLVM, which was the base for our latter work in HDNN [118].

6.4 Future Ways

We believe that the solutions proposed in this thesis can be useful for the future of heterogeneous computing, especially for improving its usability and performance. Besides, we envision different lines for improvements and future work. The following are the most relevant:

- **A high-level interface for HDNN.** Despite the good capabilities of HDNN, developers would have to work directly at the MLIR level and interact with the HDNN dialect directly. It is not the ideal scenario since MLIR programming is at a low level of abstraction, which is rare and known by very few developers. Instead, we would like to design a high-level language for HDNN, similar to easy-to-use languages like Python. This high-level language on top of HDNN would facilitate the extensibility and increase the popularity of HDNN, making it easier for the general audience to use and expand it.
- **Extending the ATC compiler for broader domains and languages.** We have proved that behavioral synthesis has great potential for replacing CPU code with calls to optimized APIs automatically. However, the ATC

approach has only been applied to matrix multiplication and convolutions. Long term, we would like to explore the compilation of more complex programs and not limit the compiler to replace only one code. Ideally, we would like to replace all the handmade implementations in a full machine-learning framework like Caffe with optimized APIs, instead of replacing just GEMM and convolution. Further, we wish to extend support for other programming languages, like Python.

- **A machine learning approach to find the right mapping between two functions.** In ATC, a machine learning approach is used to find the acceleratable candidates, and a deterministic algorithm with different heuristics is used to find the mapping between the user code and the accelerator API. However, machine learning could also be used to approach the latter, as recent works show [135]. We have proposed different heuristics and techniques to reduce the search space, but we also believe that a machine learning technique could be useful to find those patterns automatically. In the case of using a neural network, valid mappings could be easily validated using IO tests.

- **Using Large Language Models (LLMs) to detect kernel types in code.** To find the acceleratable parts of the user code, ATC uses neural embeddings to detect if a function belongs to a kernel type (e.g., whether a function is a matrix multiplication or not). In our experience, this methodology works well in conjunction with I/O validation. Rather than I/O testing all the functions within a program, the neural embeddings act as a "filter" to cut down the search space and remove functions that are clearly not the kernel we are looking for. Then, I/O testing is used to find the right function across the selection from neural embeddings. Recent advances in Large Language Models (e.g., GPT-3) have shown amazing performance detecting kernel types. Although I/O testing will probably be used to ensure that the function is actually performing the kernel or not, we believe that LLMs could substantially reduce the search space compared to neural embeddings.

- **Detailed analysis of scheduling techniques for exploiting ALP.** We plan to further extend POAS with more sophisticated scheduling policies. Although the proposed ones provide high levels of hardware utilization in many scenarios, we believe that exploring new approaches can enhance the flexibility of the POAS framework to other domains. Another open topic is how to efficiently schedule the communications between the CPU and accelerators, which can also have a notable impact on overall performance,

especially in shared bus scenarios. Long-term, we wish to extend and explore new domains with POAS. We believe that POAS is an excellent way of exploiting ALP in the present and next-generation computing systems.

- **Design of a new scheduler for running many workloads in ALP.** POAS is a framework that allows running one application in ALP, thus running in different accelerators concurrently. Another interesting line in ALP is to run many applications in ALP. Rather than occupying all the hardware with just one application (our framework POAS), we would seek to distribute all the applications, to either maximize hardware usage or minimize energy consumption. It opens a fascinating challenge: discoverability. In other words, discovering applications and hardware, and especially understanding which accelerators are useful for which applications. This would probably require adding new information at some point about the accelerator capabilities, supported APIs, etc. Fortunately, part of the POAS framework could be reused for this research since predicting performance and or energy would also be needed for assigning workloads to accelerators. We believe that this new approach could be integrated inside the operating system to orchestrate the execution applications, much like current process schedulers, but for accelerators.

Appendix A

Caffe Implementation Details

A.1 The Softmax Layer in Caffe

The softmax layer essentially applies the softmax function. This function takes a vector of N dimensions as input and produces a vector of the same number of dimensions. At each dimension, the output contains all the values in the input normalized in the $[0, 1]$ range. The total sum must be 1 so that they can be interpreted as probabilities. Given a vector z with size i, and σ as the softmax function, it is computed as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

This layer is usually the last one in CNNs since it computes the probabilities of the classifier, that is, the probability of an image belonging to a class. Following the formula, the Caffe code computes softmax in four differentiated steps:

1. The max value of the first row of the cube is subtracted from all the elements. This procedure guarantees that no numerical issues arise.

2. The exponentiation of all the elements is done (e^{z_i}).

3. All the elements for a given classification are accumulated ($\sum_{j=1}^{K} e^{z_j}$). To do so, the cube is iterated by columns.

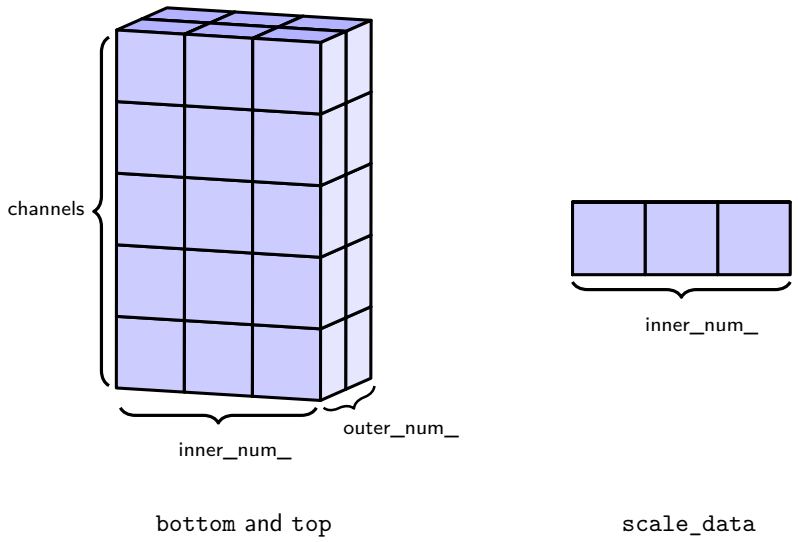


Figure A.1: Data organization in Caffe's original softmax layer.

4. All the elements are divided by the accumulated value (again, this is done by columns).

Figure A.1 shows the data layout that is used by the softmax layer in the original Caffe version.

In addition to `bottom` and `top` cubes, `scale_data` is an additional 1D vector that Caffe uses to store temporal data. Figure A.1 shows `bottom` and `top` as a 3-dimension structure, instead of 4. This happens because the last dimension stores more data, so the full input contains many 3D cubes as shown in the figure. From an implementation standpoint, the main weakness of the Caffe implementation in CPU is the code being sequential, and as such, it is unable to benefit from multicores' parallel hardware.

A.2 The Convolution Layer in Caffe

Caffe implements the convolution using general matrix multiplication (GEMM). This approach is adopted by many deep learning frameworks due to performance reasons. To compute the convolution using matrix multiplication, the first thing to do is to transform the input using a method called "image to columns" (`im2col`). Caffe relies on different optimized libraries to do the convolution: openBLAS, ATLAS or MKL in the CPU and cuBLAS in the GPU. Depending on the selected backend, the performance may vary. Some backends do not even

provide parallelization in the CPU (for example, openBLAS), in which case convolution will run sequentially. If MKL is used instead, parallelization and other low-level optimizations (like vectorization) are applied. Furthermore, Intel made its version of Caffe, which is also based on the same `im2col` and GEMM idea, but it is implemented using OpenMP. Thus, we will compare the oneAPI convolution layer against two different approaches: the isolated convolution layer from BVLC Caffe [1] and the one from Intel Caffe [2]. While the `im2col`+GEMM method is a typical way of implementing the convolution, it is not the only one. Winograd convolution [108] and direct convolution [203] are other efficient competitive approaches. Both of them are available in the oneDNN library, which we adopt to compute the convolution using oneAPI. Another important fact to take into account is that Caffe uses an NCHW layout for the input. During the original Caffe layer exploration, we found three weaknesses that can be exploited to improve the performance. First, Caffe uses its own implementation of `im2col`, which is potentially slower than an optimized one. Second, Caffe expects the selected backend to be parallelized, which is not always true. Third, Caffe implementation has a lot of code that handles manually the `im2col` execution and other aspects of the convolution. This handcrafted code is also potentially slower than an optimized one, like the one in the oneDNN library.

[1] Available at `https://github.com/BVLC/caffe`

[2] Available at `https://github.com/intel/caffe`

Appendix B

PHAST-Caffe Implementation

B.1 Softmax (Feedforward)

In the softmax computation, Caffe works with the data structures `top` and `bottom` and `scale_data_` (see Figure A.1). The `scale_data_` structure stores temporal data for intermediate computations. In the case of the `top` structure, it contains data for multiple classifications. The values for a single softmax classification are stored in the `channel` axis. Since the softmax function operates in a 1D space, the remaining axis (`inner_num` and `outer_num`) store data from different classifications. The softmax layer in the Caffe code takes several steps to produce the output:

1. The max value of the first row of the cube is subtracted from all the elements. This procedure avoids numerical issues.
2. The exponentiation of all the elements is computed.
3. All the elements for a given classification are accumulated. This procedure is done by iterating the cube by columns.
4. All the elements are divided by the accumulated value (column-wise too).

To implement the feedforward stage of the softmax layer, we distinguish two approaches:

- Direct translation (**v1**): For each line of the code in the original Caffe implementation we translate it to the PHAST library.

```
template <>
void Forward_cpu(vector<Blob<float>> bottom, vector<Blob<float>> top) {
  phast::cube<float> bottom_data = bottom[0]->getDataAsCube(...);
  phast::cube<float> top_data = top[0]->getDataAsCube(...);
  phast::vector<float> scale_data = scale_.getDataAsVector(...);
  (...)
  for(auto it = top_data.begin_i(); it != top_data.end_i(); it++) {
    (...)
    // 1. subtraction
    for (int j = 0; j < channels; j++) {
      for (int k = 0; k < inner_num_; k++) {
        scale_data[k] = phast::math::fmax(scale_data[k], this_top[j][k]);
      }
    }

    phast::for_each(top_data.begin_ijk(),
    top_data.begin_ijk(), my_func_substract);
    // 2. exponentiation
    phast::for_each(this_top.begin_ij(),
    this_top.end_ij(), my_func_exp);
    // 3. sum after exp
    this_top.transpose();

    phast::for_each(this_top.begin_i(),
    this_top.end_i(), scale_data.begin(), my_func_sum);

    this_top.transpose();
    // 4. division
    phast::for_each(this_top.begin_i(),
    this_top.end_i(), my_func_div);
  }
}
```

Listing B.1: PHAST softmax layer (`v1`)

- Global translation (**v2**): We study the layer from beginning to end and apply the translation in a more general way. This approach enables global optimizations and modifications to the original algorithm. However, it requires much more development effort.

The straightforward version `v1` is shown in Listing B.1 (some irrelevant parts of the code are omitted to improve readability). The code structure looks the same as the original Caffe. When Caffe calls a function to compute a step of the softmax layer, the PHAST version replaces this call with a PHAST functor. The main inefficiency is caused by the outer loop that iterates sequentially over the faces of the `top_data` cube. This is also what the original Caffe does on the CPU side, which is reasonable due to the limited parallelization opportunities on the CPU. However, on the GPU, this approach is inefficient. By converting that code in our implementation, the `for_each` algorithms inside the loop body

```
template <>
void Forward_cpu(vector<Blob<float>> bottom, vector<Blob<float>> top) {
  phast::cube<float> bottom_data =
    bottom[0]->getDataAsCube(outer_num_, channels, inner_num_);
  phast::cube<float> top_data =
    top[0]->getDataAsCube(outer_num_, channels, inner_num);
  phast::matrix<float> scale_data =
    scale_.getDataAsMatrix(outer_num_, inner_num, false);

  top_data.assign(bottom_data);
  top_data.transpose_ikj();

  phast::for_each(top_data.begin_ij(), top_data.end_ij(),
                  scale_data.begin_ij(), func_softmax<float>());

  top_data.transpose_ikj();
}
```

Listing B.2: PHAST softmax layer (v2)

can only work in parallel on the two remaining dimensions. Another essential downside of the first approach is using the matrix transpose after and before step 3. The transpose is needed because the PHAST `for_each` algorithm works in a row-major fashion, but Caffe iterates this structure in a column-major fashion. Finally, the remaining four operations are partitioned into four different `for_each` algorithms, which is inefficient because the kernel launching on GPU or thread launching on CPU and final synchronizations are paid four times at each iteration of the outer loop.

The v2 softmax is implemented to take advantage of as much parallelism as possible. In this new implementation (Listing B.2), we replace the outer loop with a single `for_each` algorithm that processes in parallel `outer_num`×`inner_num` vectors of `channels` elements. To allow these modifications, we make two changes:

- The `top_data` cube is transposed to swap the two minor dimensions;
- The `scale_data` container is transformed into a matrix with `outer_num` × `inner_num` elements.

Additionally, all the previous calculations are moved into a single `for_each` algorithm that uses a single functor (not shown). Inside it, the vector is manipulated taking advantage of two in-functor `for_each` algorithms that can leverage an additional axis of parallelism on GPUs [153]. By moving every operation into a single functor and eliminating the outer loop, this new version extracts much parallelism. It is also more concise and simple.

B.2 Convolution (Feedforward)

Caffe uses a matrix multiplication approach for computing the convolution, benefiting from high-performance matrix multiplications on both CPU and GPU using BLAS-based libraries. In PHAST, the matrix multiplication takes advantage of cuBLAS on the GPU side, while no special-purpose library has been integrated on the CPU, which leads to worse performance than the original Caffe. Instead of improving the PHAST backend for matrix multiplication, we decide to add a native convolution algorithm to the PHAST library. We believe that this enhanced convolution algorithm can improve the performance of matrix multiplication convolutions. In this case, the re-engineering process is more relevant on the PHAST backend than in the Caffe frontend. Hence, the convolution layer at the Caffe level is very much simple. The main task to accomplish is to prepare the data to pass to the PHAST native convolution. As happened with the softmax layer, two different approaches are considered:

- The PHAST convolution computes one batch at a time (**v1**): Caffe iterates over the batches of the input and calls the native convolution for each of them.
- The PHAST convolution computes all the batches in parallel (**v2**): Caffe calls PHAST native convolution directly with all the batches, which are processed internally in the PHAST library.

In the `v1` version, the PHAST convolution algorithm is parallelized over the number of filters. Therefore, it calls PHAST convolution which runs in parallel. Additionally, the PHAST convolution call also applies the bias, so the Caffe frontend does not have to take care of it. Listing B.3 shows the implementation code (omitted code corresponds to data preparation).

The outer loop iterates over the number of inputs to be convolved. The loop count of the outer loop is usually one, so it is not a big concern for us from the performance perspective. At each iteration of the outer loop we build a PHAST grid containing the convolution data. The grid is then iterated over the number of batches, obtaining a cube at each iteration that is computed using the PHAST convolution algorithm.

In the second approach (`v2`), the PHAST library takes the full input (a PHAST cube) and computes the convolution for all the batches. Thus, the Caffe frontend arranges all the data as a PHAST cube container and calls the PHAST native algorithm. In this approach, the PHAST convolution algorithm parallelizes both the batches and the filters, so this is a coarse-grained parallelization scheme. Listing B.4 shows this implementation.

```
  // Several checks that fails if the convolution to be
  // applied is not supported
  (...)
  phast::vector<float> bias, *bias_ptr = nullptr;
  std::vector<phast::cube<float>> filters;
  (...)
  for (int i = 0; i < bottom.size(); ++i) { // Outer loop
    phast::grid<phast::cube<float>> grid_b(bottom_data, ...);
    phast::grid<phast::cube<float>> grid_t(top_data, ...);

    phast::cube<float> b_cub;
    phast::cube<float> t_cub;

    // Batch loop
    for (; b_it != grid_b.end(); b_it++, t_it++) {
      b_cub.set_dev(b_it.sub_size_i(), ...);
      t_cub.set_dev(t_it.sub_size_i(), ...);

      phast::ai::convolution(b_cub, filters, stride, bias_ptr, t_cub);
    }
  }
}
```

Listing B.3: PHAST convolution layer (v1)

```
template <>
Forward_cpu(vector<Blob<float>*>& bottom, vector<Blob<float>*>& top) {
  // Several checks that fails if the convolution to be
  // applied is not supported
  (...)

  // Prepare data
  phast::vector<float> bias, *bias_ptr = nullptr;
  (...)

  phast::cube<float> filters =
  this->blobs_[0]->getDataAsCube(...);
  for (int i = 0; i < bottom.size(); ++i) {
    (...)

    phast::cube<float> bottom_data = bottom[i]->getDataAsCube(...);
    phast::cube<float> top_data = top[i]->getDataAsCube(...);
    phast::ai::batch_convolution(bottom_data, filters,
                                 num_filters, stride,
                                 bias_ptr, top_data);
  }
}
```

Listing B.4: PHAST convolution layer (v2)

The new code is even more concise than before because the new PHAST primitive (`batch_convolution`) contains more logic than the previous one. In

the re-engineered version of the convolution, the computation workload goes directly to the PHAST library instead of relying on the Caffe front-end. Since convolution is the heaviest layer of the convolutional networks, the performance of the native PHAST convolution is crucial to achieving good performance in the whole network.

B.3 Convolution (Backpropagation)

The backpropagation phase can be divided in three different steps:

1. The bias gradient calculation;
2. The weight gradient calculation;
3. The input data gradient calculation (data stored in `bottom` data structure)

The second and third steps require the calculation of a convolution, whereas the bias gradient does not. Since the backpropagation phase also needs to compute convolutions, it also benefits from the new PHAST convolution primitive. Unlike the previous layers, we only propose one version in the backpropagation. The bias gradient computation is done by accumulating all the matrices with top_x $\times$ top_y elements in the top gradient. They are stored into a temporal matrix transposed and accumulated by rows to get the `conv_out_channels` values, which are as many as the filters, as shown in Listing B.5.

In the case of the weight gradient, the computation is essentially a convolution. The new PHAST native primitive is used, as shown in Listing B.6.

```
// Data preparation
phast::cube<float> top_diff_tmp = top[i]->getDiffAsCube(
  this->num_ * this->conv_out_channels_, top_x, top_y);
phast::grid<phast::cube<float>> top_diff(top_diff_tmp, 1, top_x, top_y);
phast::vector<float> bias_diff =
  this->blobs_[1]->getDiffAsVector(this->num_output_);
phast::matrix<float> tmp_acc(this->num_, this->conv_out_channels_);

// Computation
phast::for_each(top_diff.begin(), top_diff.end(),
  tmp_acc.begin_ij(), func_conv_bp_bias<float>());

tmp_acc.transpose();

phast::for_each(tmp_acc.begin_i(), tmp_acc.end_i(),
  bias_diff.begin(), reduceMatrixVectors<float>());
```

Listing B.5: The PHAST version of the bias gradient calculation (Step 1)

```
// Data preparation
phast::cube<float> top_diff = top[i]->getDiffAsCube(
  t_shp[0]*t_shp[1], t_shp[2], t_shp[3]);
phast::cube<float> weight_diff =
  this->blobs_[0]->getDiffAsCube(d_shp[0] * d_shp[1],  d_shp[2], d_shp[3]);
phast::cube<float> images = bottom[i]->getDataAsCube(
  b_shp[0]*b_shp[1], b_shp[2], b_shp[3]);

// Computation
phast::ai::batch_convolution_channel_major(images,
top_diff, num_filters, stride, bias_dummy, weight_diff);
```

Listing B.6: PHAST version of the weight gradient calculation (Step 2)

```
// Data prepraration
phast::cube<float> bottom_diff = top[i]->getDiffAsCube(
  b_shp[0]*b_shp[1], b_shp[2], b_shp[3]);
phast::cube<float> weight =
  this->blobs_[0]->getDataAsCube(d_shp[0] * d_shp[1], d_shp[2], d_shp[3]);
phast::cube<float> weights_t(
  weights.size_i(), weights.size_j(), weights.size_k());

// Computation
phast::for_each(weights_t.begin_i(), weights_t.end_i(),
        transposer<float>(weights, d_shp[0], d_shp[1]));

phast::ai::rotate_and_pad(weights_t,0,0,rotated_weights);
phast::ai::pad(top_diff, pad_h, pad_w, padded_top_);

phast::ai::batch_convolution(padded_top_,
    rotated_weights_, d_shp[1], stride, bias_dummy,
    bottom_diff);
```

Listing B.7: PHAST version of the input data gradient calculation (Step 3)

There is an essential detail in this step: Caffe performs the convolution transposing the input data, telling the underlying library to transpose the matrix after the matrix multiplication. Under the hood, this transpose is implemented as an implicit operation instead of transposing the data. Thanks to that, the transpose operation is very efficient. We decide to do the same inside the PHAST library, adding a new version of the convolution called `batch_convolution_channel_major` that implements the data transposition as previously explained. Lastly, regarding the input gradient calculation, the PHAST library has to be extended with two new primitives, `phast::ai::pad` and `phast::ai::rotate_and_pad`. The former is used to pad its argument, while the latter also rotates the cube faces, which is necessary to re-use the same convolution algorithm starting from the upper-left corner of the input instead of the bottom-right, as required in the input-gradient calculation. Both operations can be translated to

classic `phast::for_each` algorithms, but a native port allows to achieve higher performance. The PHAST code for this phase is given in the Listing B.7.

B.4 Adam Solver

The solver algorithm is divided into four computations:

- $m = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
- $v = \beta_2 \cdot m_{t-1} + (1 - \beta_2) \cdot {g_t}^2$
- $t = m/sqrt(v) + eps\_hat$
- $np\_diff = t * (local\_rate \cdot correction)$

where `m`, `v`, `t` and `np_diff` are vector variables related to the Adam solver. Those structures can be retrieved from Caffe directly.

Since `np_diff` is a vector, the `phast::for_each` algorithm iterates over its elements and applies the functor to all of elements in parallel. The other vectors (`val_m`, `val_v`, and `val_t`) are passed to the functor's constructor and accessed by index inside its body. The PHAST port of Adam solver is shown in Listings B.8 and B.9.

```
phast::vector<float> val_m = (...)
phast::vector<float> val_v = (...)
phast::vector<float> val_t = (...)
phast::vector<float> np_diff = (...)

adam_solver<float> solver(val_m, val_v, val_t, ... );
phast::for_each(np_diff.begin(), np_diff.end(), solver);
```

Listing B.8: The PHAST Adam solver

```
_PHAST_METHOD void operator()(
  phast::functor::scalar<T>& np_diff) {

  int i = this->get_index();

  val_m_[i] = (1 - beta1_)*np_diff + beta1_*val_m_[i];
  val_v_[i] = (1 - beta2_)*np_diff*np_diff + beta2_*val_v_[i];
  val_t_[i] = val_m_[i] / (phast::math::sqrt(val_v_[i]) + a_);
  np_diff = val_t_[i] * scale_;
}
```

Listing B.9: The PHAST Adam solver functor

B.5 Extended Evaluation

For the measurement of the isolated layers we moved the relevant source files to a different project with their dependencies. This way, a layer is completely isolated from the rest of the framework. Besides, we added a custom C++ main file to make it suitable for standalone execution, providing an easy interface to test the layer with any input. To choose the input sizes, we look at standard datasets (MNIST and CIFAR-10) at each isolated layer and choose a similar input size. The elapsed times of each execution are shown in milliseconds. The timers' recording starts after a warm-up phase to avoid any transitory effect due to high cache miss rates and power-saving policies in CPU and GPU.

B.5.1 Correctness

To check the Caffe PHAST version's correctness, we run the unit tests provided from the Caffe framework, and we obtain the results shown in Table B.1.

The Caffe framework supports many kinds of convolutions, like dilated convolutions or grouped convolutions. In the PHAST version, these convolutions fail since they are not supported. However, all the tests which evaluate basics convolutions are successful. Our intention is not to support all types of convolutions but just those needed to run the networks we want to evaluate. Therefore, these results are what we expected. As for the remaining layers, they are working, except for accuracy. Failed accuracy tests do not affect our benchmarked networks either, since the functionality which fails in the Caffe tests does not affect LeNet networks. Henceforth, we can conclude that the PHAST version works successfully for CPU and GPU.

Table B.1: Caffe test results for the preliminary PHAST implementation.

Block	Passed	Not Passed	Total	%Passed
Convolution	5	10	15	33%
Pooling	11	0	11	100%
InnerProduct	9	0	9	100%
Softmax	4	0	4	100%
Softmax Loss	4	0	4	100%
Accuracy	9	3	12	75%

B.5.2 Softmax

The extended softmax input includes 9 different inputs, which are detailed in Table B.2. Execution times are shown in Table B.3.

As anticipated, `v1` shows much worse results than `v2`. The big performance difference is due to the fact that `v1` is dominated by a sequential for loop with 4 `phast::for_each` and two matrix-transpose inside its body, while `v2` needs only one `phast::for_each` and two cube-transpose invocations. The difference is more evident on the GPU than the CPU because in `v1` the calculations are performed on sequentially iterated matrix-shaped slices of the cube, instead of the whole cube in parallel, with fewer opportunities for parallelism. This leads to low utilization of the GPU. Moreover, the higher startup and synchronization costs for the CPU play an important role.

On the CPU, PHAST `v2` performs better than original Caffe in 6 tests out of 9. The most significant difference in the elapsed times is observed in input 7, where the PHAST version is 1.5x times faster. PHAST achieves the worst result in input 4, with a speedup of 2.60x in favor of the original implementation. Overall, the PHAST version seems to scale better with the input size, showing better results when the input grows.

On the GPU, PHAST `v2` shows a significant improvement concerning our first version, but it is consistently less efficient than the original version. The original Caffe version shows little sensitivity to the input size, with small running time increases with the input. PHAST performance scores, conversely, are more affected by the parameter size, with the performance gap that grows accordingly. This effect is due only to the two cube-transposes, which are absent in the original Caffe code and present a higher sensitivity to the workload size.

Table B.2: Extended inputs for the isolated softmax layer.

Input	*H*x*W*	*C*	*N*
1	32x32	32	1
2	32x32	32	3
3	32x32	32	10
4	64x64	64	1
5	64x64	64	3
6	64x64	64	10
7	128x128	128	1
8	128x128	128	3
9	128x128	128	10

Table B.3: Extended execution times for the isolated softmax layer on the CPU and GPU.

Input	CPU			GPU		
	Original	PHAST `v1`	PHAST `v2`	Original	PHAST `v1`	PHAST `v2`
1	1.231	73.254	2.741	0.116	515.891	0.120
2	3.019	76.437	6.228	0.118	1490.160	0.356
3	15.161	96.476	11.782	0.188	5057.560	0.359
4	8.891	151.635	23.093	0.147	4095.280	0.501
5	25.895	141.634	19.012	0.260	11977.7	0.840
6	53.485	180.858	36.303	0.272	39667.5	1.216
7	71.150	248.915	47.461	0.273	33366.3	3.831
8	71.552	255.402	67.365	0.210	33125.1	3.876
9	65.998	267.166	54.301	0.267	33125.1	3.906

B.5.3 Convolution (Feedforward)

We run 18 different tests, which are shown in Table B.4. Each test consists of the execution of a workload characterized by five parameters. They have been selected carefully to measure the performance of our solution in a variety of cases. In all the tests, a value of 100 and 32, respectively, has been assigned to the number of batches and number of filters.

Also in this case, there are significant improvements in `v2` with respect to `v1`, especially for small inputs. This difference is due to the new algorithm's ability to take advantage of the parallelism offered by multiple batches.

On the CPU, PHAST `v2` performs better than the original implementation in 12 tests out of 18. These are mainly concentrated in the part of the table with small/medium input sizes. The maximum speedup is achieved in input 10, where performance is 2.57x that achieved in the original version. Conversely, the test where PHAST scores the worst performance for the original version is input 12, where the speedup is 1.27x in favor of the latter. As for the best performing parameters, the number of threads varies between 8 and 16 in the small and medium tests and sets to 16 for the biggest ones. On the GPU, the block-size is always 32 on the significant axis except for inputs 16 and 17, where it is 64, while the best scheduling strategy is `SATURATE` in the majority of cases. Overall, the PHAST revised implementation proves to be competitive on the CPU, and thus the approach described in [203] could be regarded as a valid alternative to the usual approach based on matrix multiplication.

On the GPU, the `v2` implementation performs better than the original in

Table B.4: Extended inputs for isolated convolution layer (for both feedforward and backpropagation).

Input	Batches	Image Size	Number of filters	Filters Size
1	100	28x28x1	32	5x5
2	100	28x28x3	32	5x5
3	100	28x28x10	32	5x5
4	100	32x32x1	32	5x5
5	100	32x32x3	32	5x5
6	100	32x32x10	32	5x5
7	100	32x32x1	32	16x16
8	100	32x32x3	32	16x16
9	100	32x32x10	32	16x16
10	100	64x64x1	32	5x5
11	100	64x64x3	32	5x5
12	100	64x64x10	32	5x5
13	100	64x64x1	32	16x16
14	100	64x64x3	32	16x16
15	100	64x64x10	32	16x16
16	100	64x64x1	32	32x32
17	100	64x64x3	32	32x32
18	100	64x64x10	32	32x32

8 tests out of 18, which are those with the smallest sizes. This proves that the approach described by [27] which inspired the PHAST implementation of the convolution, is competitive for small images and small filters but loses its advantage with increasing sizes. This is evident by comparing the achieved performance in the smallest tests (inputs 1, 2 and 3) against the biggest ones (inputs 16, 17 and 18). The achieved performance goes from a speedup of 18.3x over the original Caffe implementation in input 1 to the 28.1x speedup of the original implementation over PHAST in input 18.

It is worth noting that big filters as those in inputs 16, 17 and 18, where PHAST library performs poorly, are uncommon in practical DNNs. In six popular DNNs reported by [183], filter sizes are significantly smaller, being 1×1, 3×3, 5×5, 7×7, and 11×11. Considered this, our PHAST native convolution proves to be a better solution for practical DNNs.

Table B.5: Extended execution times for the isolated convolution layer in feedforward phase on the CPU and GPU.

Input	CPU			GPU		
	Original	PHAST v1	PHAST v2	Original	PHAST v1	PHAST v2
1	17.562	101.772	16.216	3.745	3.013	0.204
2	38.218	110.576	30.358	3.981	3.466	0.450
3	83.515	149.402	51.854	3.665	4.951	1.280
4	22.507	99.813	23.042	3.750	3.692	0.220
5	47.931	114.776	28.226	3.998	4.315	0.523
6	99.555	174.990	64.423	3.759	7.547	1.495
7	59.693	123.567	37.889	5.105	31.193	0.544
8	108.643	191.292	62.651	6.420	32.842	1.599
9	227.076	375.883	207.972	8.088	38.298	6.299
10	81.601	127.020	31.722	4.541	12.275	1.269
11	116.201	194.849	78.562	5.842	15.919	3.366
12	222.373	409.520	282.716	6.117	30.994	9.452
13	165.880	328.035	155.146	6.298	194.490	4.558
14	375.109	666.406	451.024	8.924	204.713	13.586
15	1349.185	1688.884	1458.930	56.993	246.431	54.157
16	259.051	473.004	269.747	13.963	1319.007	94.311
17	724.254	1070.027	786.370	20.838	1334.103	283.211
18	2708.486	2976.198	2603.698	46.735	1403.413	1316.586

B.5.4 Convolution (Backpropagation)

On the CPU, the PHAST implementation proves better than the original in 11 tests out of 18. The higher speedup is achieved in the case of input 12, where it reaches a speedup of 1.68x. On the other side, the higher speedup achieved by the original implementation over PHAST is 1.33x, in the case of input 1. Since the performance of each run is the effect of seven different algorithms, the analysis of the parameters shows no clear trend in this case, except the evidence that higher number of threads are generally selected in most expensive algorithms. Overall, the backpropagation phase confirms the trend observed in the feedforward case, with PHAST performing better than the original version in the majority of cases and losing performance especially with big inputs.

On the GPU, the achieved performance shows the same as in the feedforward case: the PHAST implementation of the convolution performs better than the original only with small inputs, being consistently faster in the first 8 tests and

Table B.6: Extended execution times for the isolated convolution layer in backpropagation on the CPU and GPU.

Input	CPU		GPU	
	Original	PHAST	Original	PHAST
1	32.986	43.958	7.767	0.986
2	61.449	70.567	7.812	1.748
3	170.250	121.933	8.712	4.907
4	42.873	50.985	7.932	1.366
5	81.283	79.649	8.040	2.654
6	233.521	154.198	9.360	7.639
7	98.756	121.867	18.820	2.858
8	291.692	281.210	19.250	7.736
9	947.073	886.190	21.050	24.620
10	115.156	97.016	10.758	12.514
11	318.930	230.114	12.432	34.193
12	903.935	537.887	16.052	81.434
13	597.130	466.535	26.137	35.175
14	1722.433	1291.191	38.996	80.487
15	5932.176	3964.532	80.841	267.251
16	1109.800	1376.846	69.909	365.565
17	3405.712	3761.880	83.782	1098.258
18	9644.456	12418.025	194.401	3636.622

slower in the remaining 11. The test where PHAST achieves the best results is input 1, with a speedup of 7.88x. On the other side of the spectrum, there is input 18, where the original implementation is 18.71x faster than the PHAST implementation. As for the parameters, the same considerations can be done on the GPU side.

Also in this case, the performance achieved in the backpropagation shows the same characteristics observed in the feedforward, with the PHAST implementation being faster with small inputs and slower with big inputs compared to the original Caffe implementation. Overall, also in the backpropagation case, PHAST loses performance on CPU and GPU with big filters, which are uncommon in DNNs employed in practical cases.

B.5.5 Adam solver

If we look at the input sizes for MNIST in the Adam solver, we find that it runs the solver for 8 different sizes of vectors that range from 5000 to 400000 elements. Thus, we execute 4 tests with sizes in the same range (inputs 1-4) and another couple with higher values to see if the performance trend continues. Adam inputs are shown in Table B.7.

Table B.7: Input sizes for isolated Adam solver.

Input	Vector Size
1	50x100
2	100x200
3	200x400
4	500x800
5	1000x1600
6	2000x3200

We find that Adam in PHAST works well out of the box on the CPU for the inputs we tried, but it pays a higher overhead than the original solution. We think that the latter may take advantage of thread pooling techniques, while the former does not. Besides, the performance gap between the two implementations is reduced by increasing the vector size. The performance achieved by both versions is almost the same (around 1.2% difference) in input 5. PHAST becomes faster than the original version in input 6, with a 1.82x speedup. Even with a higher startup cost, PHAST makes better use of the threads, which is crucial for big inputs that require all the cores to be used efficiently. The best

Table B.8: Execution times for isolated Adam solver on the CPU and GPU.

Input	CPU		GPU	
	Original	PHAST	Original	PHAST
1	0.132	21.914	220.125	214.694
2	7.007	22.962	220.913	217.499
3	8.586	26.639	220.567	217.104
4	17.246	33.971	222.024	218.645
5	47.862	48.443	222.435	218.530
6	135.164	74.157	222.221	222.519

performing number of threads is 2 in the first three tests, 8 in input 4 and 16 in input 5 and 6, growing with increasing size as expected.

On the GPU, both implementations are equally valid, as they can achieve about the same performance. The embarrassingly parallel nature of the Adam solver makes complete utilization of the GPU in both cases. Furthermore, we also have a very low sensitivity to the parallelization parameters: neither registers nor the shared memory utilization is limiting factors; thus, various configurations are managed equally well by the hardware scheduler, leading to similar results.

Appendix C

oneAPI-Caffe Implementation

To design the oneAPI version of a given layer, we first need to isolate a layer from the Caffe framework. It allows us to run just the layer itself, filling it with arbitrary data and content of various sizes. This means that both the input and output of the layer are 4D tensors in the NCHW layout. In the case of feedforward, Caffe represents the input as `bottom` and the output as `top`.

C.1 Softmax

The source code to compute the softmax layer in oneAPI is divided into four parts for clarity:

1. *Definition of the buffers that store the data to be computed by the softmax layer.* The definition code is shown in Listing C.1. SYCL buffers represent data that can exist on the host and/or any other device. This means that if the code is running on the host, no data copy is needed from the Caffe data (first argument of buffer constructor) to the buffer. If the device is not the host (e.g., a GPU with its memory), a data copy is done implicitly. Work items and work groups are defined in this section of the code. SYCL provides the possibility to express the kernel in the SIMT execution model [30]. Therefore, we have to control certain parameters in the execution of the kernel. This is the same idea of how a kernel works in CUDA. In the context of SYCL, (as happens in CUDA) these two values control the execution of SYCL kernels. Thus, in CUDA terminology work groups are equivalent to blocks and work items are equivalent to threads. In the

case of softmax, we set the size of work items to the minimum of 32 or the total size of the input. We adjust the work groups so that all work items will be working with a single data. Experienced programmers in CUDA will find this approach straightforward since oneAPI and CUDA share the SIMT execution model.

```
template <>
void SoftmaxLayer<float>::Forward_cpu(
    vector<Blob<float>*>& bottom,
    vector<Blob<float>*>& top,
    sycl::queue queue) {
  ...
  int work_items = min(32, t_size);
  int work_groups = (t_size + work_items - 1) / work_items;

  sycl::buffer<float,1> buf_t(top_ptr, sycl::range<1>(t_size));
  sycl::buffer<float,1> buf_b(bot_ptr, sycl::range<1>(b_size));
  sycl::buffer<float,1> buf_s(scale_ptr, sycl::range<1>(s_size));
```

Listing C.1: Init code for softmax layer

2. *Exponentials computation.* In SYCL, the work is described as a command group. In Listing C.2, a command group handler (variable `cgh`) is passed to the kernel. A command group encapsulates a kernel with its dependencies, and it is processed as a single entity atomically by the SYCL runtime once it is submitted to the queue. Inside the exponential kernel, we request access to write in the top buffer (`buf_t`, the output) and read from the bottom buffer (`buf_b`, the input). After that, the kernel is submitted. Inside the kernel, each thread will compute the exponential of one element of the input and store it in the same position on the output.

```
// 1. EXPONENTIAL
queue.submit([&] (sycl::handler& cgh) {
  auto t = buf_t.get_access<sycl::access::mode::discard_write>(cgh);
  auto b = buf_b.get_access<sycl::access::mode::read>(cgh);

  cgh.parallel_for<class SoftmaxExp>(sycl::nd_range<1>(work_groups *
      work_items, work_items), [=] (sycl::nd_item<1> item) {
    size_t local_id = item.get_local_linear_id();
    size_t global_id = item.get_global_linear_id();

    t[global_id] = sycl::exp(b[global_id]);
  });
});

queue.wait();
```

Listing C.2: Exponential computation

3. *Accumulation of the values previously computed with the exponential.* Due to the data layout adopted by Caffe, data access is more complex in this kernel,

shown in Listing C.3. Fortunately, because SYCL kernels can be expressed as SIMT, we copied the code to compute this part directly from the source code of Caffe in GPU (source file `softmax_layer.cu`[1]). Memory is another concept in SYCL that shares similarities with CUDA. In SYCL, there is also a difference between local and global memory. In the kernel shown in Listing C.1, all the accesses were performed in global memory. There was no other choice since we need to modify the entire output structure. However, in the case of this kernel, we can do almost all the computation in local memory. While this will likely have no impact on the CPU, GPUs and other accelerators will probably benefit from this optimization. To access global memory, SYCL provides a mechanism to allocate a chunk of local memory. In this kernel, however, we do not make use of such a mechanism. Instead, we use local memory implicitly. Local variables are stored in local memory, so `sum` accumulation is done in local memory although read access to `t` variable has to be done from global memory. After `sum` is computed, we store it in the `scale` global memory structure (see Figure A.1). We have to reassign the work item and work group variables (see the first two lines in listing C.3) because the length of the data to work with is different from the previous kernel.

```
work_items = min(32, s_size);
work_groups = (s_size + work_items - 1) / work_items;
...

// 2. SCALE
queue.submit([&] (sycl::handler& cgh) {
  auto t = buf_t.get_access<sycl::access::mode::read>(cgh);
  auto sss = buf_s.get_access<sycl::access::mode::write>(cgh);

  cgh.parallel_for<class SoftmaxScale>(sycl::nd_range<1>(work_groups
      * work_items, work_items), [=] (sycl::nd_item<1> item) {
    size_t local_id = item.get_local_linear_id();
    size_t global_id = item.get_global_linear_id();
    size_t spatial_dim = height * width;
    size_t n = global_id / spatial_dim;
    size_t s = global_id % spatial_dim;

    float sum = 0.0f;
    for (int c = 0; c < channels; c++) {
      sum += t[(n * channels + c) * spatial_dim + s];
    }

    sss[global_id] = sum;
  });
});
```

[1]Available at `https://github.com/BVLC/caffe/blob/master/src/caffe/layers/softmax_layer.cu`

```
queue.wait();
```

Listing C.3: Accumulate

4. *The division of the exponentiated values*. Values are stored in `top` data structure, represented by the `t` variable in Listing C.4. This structure has to be divided by the sum of the exponentiated values (stored in `sss`). Again, the code can be easily adapted from the CUDA Caffe version, and we also have to reassign the work item and work group variables.

```
// 3. DIVISION
work_items = min(32, t_size);
work_groups = (t_size + work_items - 1) / work_items;

queue.submit([&] (sycl::handler& cgh) {
  auto t = buf_t.get_access<sycl::access::mode::write>(cgh);
  auto sss = buf_s.get_access<sycl::access::mode::read>(cgh);

  cgh.parallel_for<class SoftmaxDivide>(sycl::nd_range<1>(work_groups
      * work_items, work_items), [=] (sycl::nd_item<1> item) {
    // variable position computations ...

    t[global_id] = t[global_id] / sss[n * spatial_dim + s];
  });
});

queue.wait();
}
```

Listing C.4: Division

However, the implementation of this kernel has a little room for improvement, but SYCL and DPC++ do not provide an easy approach to accomplish that. The problem is the usage of local memory. In the `SoftmaxScale` kernel, the variable `sum` contains the value in local memory, which has to be divided. However, we had to store it in global memory to be able to read it in the `SoftmaxDivide` kernel. The best approach would have been to compute the sum in local memory and then compute the division using local memory too, but there is no way to communicate this data between different kernels (`sum` is unavailable from `SoftmaxDivide` kernel). A tricky solution could be to calculate the division in the same kernel (the `SoftmaxScale` kernel). This is not a good idea due to the different grains of parallelism of the two kernels: `SoftmaxScale` kernel needs a small number of work groups, while `SoftmaxDivide` kernel is capable of running a much bigger amount of work groups.

C.2 Convolution

The initialization of oneDNN is similar to the one explained in Section 3.2.2.1. The only difference is that, instead of a device queue, the layer receives the engine kind to be used. oneDNN engines are an abstraction of a computational device (CPU, GPU, etc). With the engine kind, we create an engine that will be used to compute the convolution. The engine creation is shown in Listing C.5.

```
template <>
void ConvolutionLayer<float>::Forward_cpu(
    vector<Blob<float>*>& bottom,
    vector<Blob<float>*>& top,
    engine::kind engine_kind) {
  dnnl::engine engine(engine_kind, 0);
  dnnl::stream engine_stream(engine);
```

Listing C.5: Engine initialization in convolution layer

The goal of the initialization code is translating from the Caffe idiom to oneDNN. After the engine creation, we populate the variables (Listing C.6) to be used in oneDNN calls using Caffe data. With `bottom[0]->shape()` and `top[0]->shape()`, we retrieve the shape of the tensor that represents the input (`bottom`) and the output (`top`), which are stored in a 4D vector. With that information, we can initialize the convolution values (batch size, number of weights, size of the weights, etc).

```
std::vector<int> input_shape = bottom[0]->shape();
std::vector<int> output_shape = top[0]->shape();
...

const memory::dim N  = input_shape[0], // batch size
const memory::dim IC = input_shape[1], // input channels
...

const memory::dim OC = output_shape[1], // output channels
const memory::dim KH = weights_shape[2], // weights height
const memory::dim KW = weights_shape[3], // weights width
...
```

Listing C.6: Convolution data size initialization

To interact with memory, oneDNN has two abstractions: the memory description and the memory object itself. In Listing C.7, we show the creation of the memory descriptor of the input data, represented by `conv_src_md`.

```
memory::dims src_dims = {N, IC, IH, IW};
...
auto conv_src_md = memory::desc(src_dims, dt::f32, tag::any);
```

Listing C.7: Input memory description creation

After the creation of the memory description, we create the memory object (see Listing C.8), which describes not only the data layout (NCHW in this case) but also the engine to be used and the dimensions of the tensor. Because the memory object also contains the data to be processed, we must copy the contents of the input to the memory object. The input (given by the Caffe layer) is stored in `blob` structure. If the code is running in a CPU, a simple `memcpy` is performed. In the case of a GPU or other device, the same idea is applied using the appropriate functions. The data copy is wrapped inside the function `write_to_dnnl_memory`.

```
auto conv_src_mem = memory({src_dims, dt::f32, tag::nchw}, engine);
write_to_dnnl_memory((void *) bottom[0]->cpu_data(), conv_src_md);
```

Listing C.8: Input memory creation

After creating the memory object and descriptors for the output, bias and weights, we can create the forward convolution description (listing C.9), which is needed to run the convolution. In this description, we specify which convolution algorithm we want. In this case, we are using direct convolution (`algorithm::convolution_direct`), but the Winograd convolution is available too in oneDNN (see `https://oneapi-src.github.io/oneDNN/group__dnnl__api__attributes.html#ga00377dd4982333e42e8ae1d09a309640`). With the forward convolution description, we can create the oneDNN primitive, which we named `conv_prim`.

```
auto conv_desc = convolution_forward::desc(prop_kind::forward_training,
                                           algorithm::convolution_direct,
                                           conv_src_md,
                                           conv_weights_md,
                                           user_bias_md,
                                           conv_dst_md,
                                           strides_dims,
                                           padding_dims_l,
                                           padding_dims_r);
// Create primitive descriptor.
auto conv_pd  = convolution_forward::primitive_desc(conv_desc, engine);
...
// Create the primitive.
auto conv_prim = convolution_forward(conv_pd);
```

Listing C.9: Fordward convolution description creation

To link the memory descriptor with its memory objects, we use the convolution arguments, whose creation is shown in Listing C.10.

```
std::unordered_map<int, memory> conv_args;
conv_args.insert({DNNL_ARG_SRC, conv_src_mem});
conv_args.insert({DNNL_ARG_WEIGHTS, conv_weights_mem});
conv_args.insert({DNNL_ARG_BIAS, user_bias_mem});
conv_args.insert({DNNL_ARG_DST, conv_dst_mem});
```

Listing C.10: Convolution arguments creation

To actually run the convolution, we need the oneDNN primitive, the oneDNN engine and the convolution arguments.

```
conv_prim.execute(engine_stream, conv_args);
```

Listing C.11: Convolution execution

After the convolution is computed, we have to read the data from the appropriate memory object and copy it back to the Caffe structure. We encapsulate the data copy inside the function `read_from_dnnl_memory`.

```
read_from_dnnl_memory(top[0]->mutable_cpu_data(), user_dst_mem);
```

Listing C.12: Data copy from oneDNN memory to Caffe structures

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association. 2.1, 2.3, 3.5

[2] S. Adve and R. Bodik. I-USHER: Interfaces to Unlock the Specialized Hardware Revolution. Information Science and Technology (ISAT), 2019. `http://rsim.cs.illinois.edu/Talks/I-USHER.pdf`. 3.5

[3] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, and S. Kamil. Automatically Translating Image Processing Libraries to Halide. *ACM Transactions on Graphics*, 38:1–13, Nov. 2019. doi: 10.1145/3355089.3356549. 4.4

[4] U. Ahmed, J. C.-W. Lin, G. Srivastava, and M. Aleem. A load balance multi-scheduling model for OpenCL kernel tasks in an integrated cluster. *Soft Computing*, 25(1):407–420, Jan. 2021. doi: 10.1007/s00500-020-05152-8. 2.7

[5] Aksel Alpay. hipSYCL - an implementation of SYCL over NVIDIA CUDA/AMD HIP, 2019. `https://github.com/illuhad/hipSYCL`. 3.5

[6] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *the 2015 10th Joint Meeting*. ACM Press, Aug. 2015. doi: 10.1145/2786805.2786849. 4.4

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3:1–29, Jan. 2019. doi: 10.1145/3290353. 4.4

[8] M. S. B. Altaf and D. A. Wood. LogCA: A High-Level Performance Model for Hardware Accelerators. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 375–388, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3079856.3080216. 4.4

[9] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, et al. Programming languages for data-Intensive HPC applications: A systematic mapping study. *Parallel Computing*, 91:102584, 2020. doi: 10.1016/j.parco.2019.102584. 3.5

[10] AMD Corporation. New AMD ROCm Information Portal - ROCm v4.5 and Above, Nov. 2022. `https://rocmdocs.amd.com/en/latest`. 1.3

[11] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy. 2.9TOPS/W Reconfigurable Dense/Sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14NM Tri-Gate CMOS. In *2018 IEEE Symposium on VLSI Circuits*, pages 39–40, 2018. doi: 10.1109/VLSIC.2018.8502333. 2.5

[12] M. Anderson, B. Chen, S. Deng, J. Fix, M. Gschwind, A. Kalaiah, C. Kim, J. Lee, J. Liang, H. Lui, et al. First-Generation Inference Accelerator Deployment at Facebook. 2021. doi: 10.48550/arXiv.2107.04140. 4.1.1, 4.4

[13] J. M. Andión. *Compilation techniques for automatic extraction of parallelism and locality in heterogeneous architectures*. Ph.D. Thesis, Universidade Da Coruña, 2015. `http://hdl.handle.net/2183/15854`. 4.4

[14] K. Angstadt, J.-B. Jeannin, and W. Weimer. Accelerating Legacy String Kernels via Bounded Automata Learning. ACM, Mar. 2020. doi: 10.1145/3373376.3378503. 4.4

[15] Arm. Arm Ethos-U55: microNPU, 2020. `https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u55`. 4.1.1, 4.4

[16] J. Armengol-Estapé and M. F. P. O'Boyle. Learning C to x86 Translation: An Experiment in Neural Compilation. 2021. doi: 10.48550/arXiv.2108.07639. 2.6

[17] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program Synthesis with Large Language Models. 2021. doi: 10.48550/arXiv.2108.07732. 2.6

[18] S. H. S. Basha, M. Farazuddin, V. Pulabaigari, S. R. Dubey, and S. Mukherjee. Deep Model Compression based on the Training History. 2021. doi: 10.48550/arXiv.2102.00160. 5.2.2.5

[19] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001. doi: 10.1109/71.963416. 5.4.2

[20] T. Ben-Nun and T. Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Comput. Surv.*, 52(4), Aug. 2019. doi: 10.1145/3320060. 3.3.2.3, 5.2.2.5

[21] S. G. Bhaskaracharya, J. Demouth, and V. Grover. Automatic Kernel Generation for Volta Tensor Cores. 2020. doi: 10.48550/arXiv.2006.12645. 4.1.1, 4.4

[22] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002. doi: 10.1145/567806.567807. 2.3

[23] G. H. Blindell. *Universal Instruction Selection*. Ph.D. Thesis, KTH Royal Institute of Technology, 2018. 4.4

[24] J. Cámara, J. Cuenca, and D. Giménez. Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. *The Journal of Supercomputing*, 76(12):9922–9941, Dec. 2020. doi: 10.1007/s11227-020-03235-9. 5.4.2

[25] S. Catalán, F. D. Igual, R. Mayo, L. Piñuel, E. S. Quintana-Ortí, and R. Rodríguez-Sánchez. Performance and Energy Optimization of Matrix Multiplication on Asymmetric big.LITTLE Processors. 2015. doi: 10.48550/arXiv.1507.05129. 5.4.2

[26] L. Chelini, A. Drebes, O. Zinenko, A. Cohen, N. Vasilache, T. Grosser, and H. Corporaal. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26, 2021. doi: 10.1109/CGO51591.2021.9370332. 4.4

[27] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka. A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, page 81,

New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3295500.3356162. 3.2.1.6, B.5.3

[28] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association. 3.5

[29] J. Choquette, O. Giroux, and D. Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2):42–52, 2018. doi: 10.1109/MM.2018.022071134. 2.5, 4.1.1

[30] Codeplay. SYCL for CUDA developers - Execution Model, 2020. `https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/execution-model`. 1

[31] J. Coiffier. *Fundamentals of Numerical Weather Prediction*. Cambridge University Press, 2011. doi: 10.1017/CBO9780511734458. 4.1.1

[32] B. Collie. *Practical Synthesis from Real-World Oracles*. Ph.D. Thesis, The University of Edinburgh, June 2022. doi: 10.7488/era/2334. 4.1.2

[33] B. Collie, P. Ginsbach, and M. F. O'Boyle. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 55–67, 2019. doi: 10.1109/PACT.2019.00013. 4.4

[34] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, and M. F. P. O'Boyle. M3: Semantic API Migrations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 90–102, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3324884.3416618. 4.4

[35] D.-A. Constantinescu, A. Navarro, F. Corbera, J.-A. Fernández-Madrigal, and R. Asenjo. Efficiency and productivity for decision making on low-power heterogeneous CPU+GPU SoCs. *The Journal of Supercomputing*, 77(1):44–65, Mar. 2020. doi: 10.1007/s11227-020-03257-3. 3.5

[36] M. Costanzo, E. Rucci, C. García-Sánchez, M. Naiouf, and M. Prieto-Matías. Migrating CUDA to oneAPI: A Smith-Waterman Case Study. In I. Rojas, O. Valenzuela, F. Rojas, L. J. Herrera, and F. Ortuño, editors, *Bioinformatics and Biomedical Engineering*, pages 103–116, Cham, 2022. Springer International Publishing. doi: 10.1007/978-3-031-07802-6_9. 3.5

[37] M. Costanzo, E. Rucci, C. G. Sánchez, M. Naiouf, and M. Prieto-Matías. Assessing Opportunities of SYCL and Intel oneAPI for Biological Sequence Alignment. 2022. doi: 10.48550/arXiv.2211.10769. 3.4.3.2, 3.5

[38] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze. Automatic Generation of High-Performance Quantized Machine Learning Kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 305–316, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3368826.3377912. 4.4

[39] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR, 18–24 Jul 2021. 4.1.1, 4.1.2, 4.2.1

[40] I. Dagli, A. Cieslewicz, J. McClurg, and M. E. Belviranli. AxoNN: Energy-Aware Execution of Neural Network Inference on Multi-Accelerator Heterogeneous SoCs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 1069–1074, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3489517.3530572. 5.4.2

[41] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming, and Jiang. GitHub Copilot AI pair programmer: Asset or Liability? 2022. doi: 10.48550/arXiv.2206.15331. 2.6

[42] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient Embedded Computing. *Computer*, 41(7):27–32, 2008. doi: 10.1109/MC.2008.224. 1.2

[43] W. J. Dally, S. W. Keckler, and D. B. Kirk. Evolution of the Graphics Processing Unit (GPU). *IEEE Micro*, 41(6):42–51, 2021. doi: 10.1109/MM.2021.3113475. 2.5

[44] W. J. Dally, Y. Turakhia, and S. Han. Domain-Specific Hardware Accelerators. *Commun. ACM*, 63(7):48–57, June 2020. doi: 10.1145/3361682. 1, 1.2, 2.5, 4.1.1

[45] D. Das. An Introduction to AMD Optimizing C/C++ Compiler. In *European LLVM Developers Meeting*, 2018. https://llvm.org/devmtg/2018-04/slides/Das-An Introduction to AMD Optimizing Compiler.pdf. 2.1

[46] J. a. P. L. De Carvalho, B. Kuzma, I. Korostelev, J. N. Amaral, C. Barton, J. Moreira, and G. Araujo. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *ACM Trans. Archit. Code Optim.*, 18(3), June 2021. doi: 10.1145/3459010. 1.3, 4.1.1, 4.1.2, 4.3.2.1, 4.3.4.1, 4.3.4.3, 4.4

[47] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *the 2018 26th ACM Joint Meeting*. ACM Press, Nov. 2018. doi: 10.1145/3236024.3236059. 4.4

[48] M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020. doi: 10.1017/9781108679930. 4.1.1

[49] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi: 10.1109/JSSC.1974.1050511. 1.2

[50] B. Di Martino and G. Iannello. PAP Recognizer: a tool for automatic recognition of parallelizable patterns. In *WPC '96. 4th Workshop on Program Comprehension*, pages 164–174, 1996. doi: 10.1109/WPC.1996.501131. 4.4

[51] J. Domke, E. Vatai, A. Drozd, P. ChenT, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. WahibT, et al. Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws? In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1056–1065, Los Alamitos, CA, USA, May 2021. IEEE Computer Society. doi: 10.1109/IPDPS49936.2021.00114. 4.1.1

[52] M. Dukhan. The Indirect Convolution Algorithm. In *Efficient Deep Learning for Compute Vision (ECV) workshop*, page 10, 2019. doi: 10.48550/arXiv.1907.02129. 3.2.1.6

[53] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A proposal for programming heterogeneous multicore architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. doi: 10.1142/S0129626411000151. 3.5

[54] A. Dutta, S. Gupta, B. Khaleghi, R. Chandrasekaran, W. Xu, and T. Rosing. HDnn-PIM: Efficient in Memory Design of Hyperdimensional Computing

with Feature Extraction. In *Proceedings of the Great Lakes Symposium on VLSI 2022*, GLSVLSI '22, page 281–286, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3526241.3530331. 4.4

[55] H. C. Edwards and C. R. Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24, Aug. 2013. doi: 10.1109/XSW.2013.7. 1.3, 1, 3.5, 4.1.1

[56] A. Ejjeh, A. Councilman, A. Kothari, M. Kotsifakou, L. Medvinsky, A. R. Noor, H. Sharif, Y. Zhao, S. Adve, S. Misailovic, and V. Adve. HPVM: Hardware-Agnostic Programming for Heterogeneous Parallel Systems. *IEEE Micro*, 42(5):108–117, 2022. doi: 10.1109/MM.2022.3186547. 2.1, 3.5

[57] A. C. Elster and T. A. Haugdahl. Nvidia Hopper GPU and Grace CPU Highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022. doi: 10.1109/MCSE.2022.3163817. 1.2

[58] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011. doi: 10.1145/2000064.2000108. 1.2

[59] J. Flores-Contreras, H. A. Duran-Limon, A. Chavoya, and S. H. Almanza-Ruiz. Performance prediction of parallel applications: a systematic literature review. *The Journal of Supercomputing*, 77(4):4014–4055, Apr. 2021. doi: 10.1007/s11227-020-03417-5. 2.7

[60] B. W. Ford and Z. Zong. A cost effective framework for analyzing cross-platform software energy efficiency. *Sustainable Computing: Informatics and Systems*, 35:100661, 2022. doi: 10.1016/j.suscom.2022.100661. 5.4.1

[61] B. Forsberg, L. Benini, and A. Marongiu. HePREM: A Predictable Execution Model for GPU-based Heterogeneous SoCs. *IEEE Transactions on Computers*, 70(1):17–29, 2021. doi: 10.1109/TC.2020.2980520. 5.4.1

[62] J. Fowers, K. Ovtcharov, M. K. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al. Inside Project Brainwave's Cloud-Scale, Real-Time AI Processor. *IEEE Micro*, 39(3):20–28, 2019. doi: 10.1109/MM.2019.2910506. 4.1.1, 4.4

[63] Y. Fu, E. Bolotin, N. Chatterjee, D. Nellans, and S. W. Keckler. GPU Domain Specialization via Composable On-Package Architecture. *ACM Trans. Archit. Code Optim.*, 19(1), Dec. 2021. doi: 10.1145/3484505. 1.2

[64] G. Führ, S. H. Hamurcu, D. Pala, T. Grass, R. Leupers, G. Ascheid, and J. F. Eusse. Automatic Energy-Minimized HW/SW Partitioning for FPGA-Accelerated MPSoCs. *IEEE Embedded Systems Letters*, 11(3):93–96, 2019. doi: 10.1109/LES.2019.2901224. 4.4

[65] gcc documentation. 26.2 Match and Simplify: The Language, 2022. `https://gcc.gnu.org/onlinedocs/gccint/The-Language.html`. 4.4

[66] T. Geng, M. Amaris, S. Zuckerman, A. Goldman, G. R. Gao, and J.-L. Gaudiot. A Profile-Based AI-Assisted Dynamic Scheduling Approach for Heterogeneous Architectures. *International Journal of Parallel Programming*, 50(1):115–151, Feb. 2022. doi: 10.1007/s10766-021-00721-2. 5.4.1

[67] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018. doi: 10.1109/SC.2018.00069. 3.2.1.6

[68] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. *SIGPLAN Not.*, 53(2):139–153, Mar. 2018. doi: 10.1145/3296957.3173182. 1.3, 2.1, 4.1.1, 4.1.2, 4.3.2.1, 4.3.4.1, 4.4

[69] GitHub. Your AI pair programmer, 2022. `https://github.com/features/copilot`. 2.6

[70] M. Goli, K. Narasimhan, R. Reyes, B. Tracy, D. Soutar, S. Georgiev, E. M. Fomenko, and E. Chereshnev. Towards Cross-Platform Performance Portability of DNN Models using SYCL. pages 25–35. 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2020. doi: 10.1109/P3HPC51967.2020.00008. a, 3.5

[71] K. Goto and R. A. v. d. Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.*, 34(3), May 2008. doi: 10.1145/1356052.1356053. 4.3.2

[72] A. Green et al. libffi - A Portable Foreign Function Interface Library, 2022. `http://sourceware.org/libffi/`. 4.2.2.1

[73] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011. 2.1, 4.1.1, 4.1.2, 4.3.2.1

[74] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. 2.6, 4.1.2

[75] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser. Domain-Specific Multi-Level IR Rewriting for GPU. 2020. doi: 10.48550/arXiv.2005.13014. 2.1, 2, 3.5

[76] E. J. Gómez-Hernández, P. A. Martínez, B. Peccerillo, S. Bartolini, J. M. García, and G. Bernabé. Using PHAST to port Caffe library: First experiences and lessons learned. In *13th International Workshop on Programmability and Architectures for Heterogeneous Multicores*, page 11, 2020. doi: 10.48550/arXiv.2005.13076. 1, 3.2.1.1, 6.3.2

[77] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. 1

[78] Hennessy, John L. and Patterson, David A. A New Golden Age for Computer Architecture. *Commun. ACM*, 62(2):48–60, Jan. 2019. doi: 10.1145/3282307. 1

[79] M. Hill and V. Janapa Reddi. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330, 2019. doi: 10.1109/HPCA.2019.00047. 2.7

[80] M. D. Hill and V. J. Reddi. Accelerator-Level Parallelism. *Commun. ACM*, 64(12):36–38, Nov. 2021. doi: 10.1145/3460970. 1.1, 5.1.1

[81] H.-R. Huang, D.-Y. Hong, J.-J. Wu, P. Liu, and W.-C. Hsu. Efficient Video Captioning on Heterogeneous System Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1035–1045, 2021. doi: 10.1109/IPDPS49936.2021.00112. 5.4.2

[82] IBM. IBM ILOG CPLEX Optimizer, 2022. `https://www.ibm.com/analytics/cplex-optimizer`. 5.2.2.2

[83] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool. AI Benchmark: Running Deep Neural Networks on Android Smartphones. 2018. doi: 10.48550/arXiv.1810.01109. 1.2

[84] Intel. oneAPI, 2021. `https://github.com/intel/llvm`. 3.1.1

[85] Intel. Optimizing software for x86 Hybrid Archiecture. *Intel White Paper*, 2021. 5.1.2

[86] Intel. AI Hardware, 2022. `https://www.intel.com/content/www/us/en/artificial-intelligence/hardware.html`. 4.1.1, 4.4

[87] Intel. oneAPI Specification, Sept. 2022. `https://spec.oneapi.com/versions/latest/oneAPI-spec.pdf`. 1.3, 2.1, 3.1.1, 3.5, 4.1.1, 5.4.3

[88] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. 2014. doi: 10.48550/arXiv.1408.5093. 2.3, 1, 4.3.2

[89] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. 2019. doi: 10.48550/arXiv.1903.07486. 2.5

[90] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. 2018. doi: 10.48550/arXiv.1804.06826. 2.5

[91] H. Jiang. Intel's Ponte Vecchio GPU: Architecture, System and Software. In *IEEE Hot Chips 34 Symposium (HCS)*, pages 1–29, 2022. doi: 10.1109/HCS55958.2022.9895631. 3.4.3.2

[92] H. Jin, C. Liu, H. Liu, R. Luo, J. Xu, F. Mao, and X. Liao. ReHy: A ReRAM-Based Digital/Analog Hybrid PIM Architecture for Accelerating CNN Training. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2872–2884, 2022. doi: 10.1109/TPDS.2021.3138087. 4.4

[93] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, et al. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, Los Alamitos, CA, USA, June 2021. IEEE Computer Society. doi: 10.1109/ISCA52012.2021.00010. 3.4.4, 4.1.1, 4.4

[94] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM*, 63(7):67–78, June 2020. doi: 10.1145/3360307. 3.4.4

[95] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3079856.3080246. 1.2, 4.1.1

[96] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 92–101, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2597073.2597074. 4.1.1

[97] H. Kang, H. C. Kwon, and D. Kim. HPMaX: heterogeneous parallel matrix multiplication using CPUs and GPUs. *Computing*, 102(12):2607–2631, Dec. 2020. doi: 10.1007/s00607-020-00846-1. 5.1.2, 5.4.2

[98] D. Kasperek, M. Podpora, and A. Kawala-Sterniuk. Comparison of the Usability of Apple M1 Processors for Various Machine Learning Tasks. *Sensors*, 22(20), 2022. doi: 10.3390/s22208005. 1.2

[99] Khronos OpenCL Working Group. SYCL Provisional Specification, version 1.2.1, Nov. 2019. `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`. 2.2.2, 3.5

[100] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, page 15, 2015. doi: 10.48550/arXiv.1412.6980. 3.2.1.5

[101] J. Klainongsuang, Y. S. Nugroho, H. Hata, B. Manaskasemsak, A. Rungsawang, P. Leelaprute, and K. Matsumoto. Identifying Algorithm Names in Code Comments. 2019. doi: 10.48550/arXiv.1907.04557. 4.4

[102] K. Komisarczyk, L. Chelini, K. Vadivel, R. Jordans, and H. Corporaal. PET-to-MLIR: A polyhedral front-end for MLIR. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 551–556, 2020. doi: 10.1109/DSD51259.2020.00091. 2.1, 2, 3.5

[103] T. Kosar, S. Bohra, and M. Mernik. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, 71:77–91, 2016. doi: 10.1016/j.infsof.2015.11.001. 3.1.1

[104] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve. HPVM: Heterogeneous Parallel Virtual Machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 68–80, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3178487.3178493. 2.1, 3.5

[105] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665. 2.1, 4.2.2.1

[106] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv, 2020. doi: 10.48550/arXiv.2002.11054. 2.1, 3.5

[107] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308. 2.1, 3.5

[108] A. Lavin and S. Gray. Fast Algorithms for Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016. doi: 10.1109/CVPR.2016.435. A.2

[109] C. Leary and T. Wang. XLA: TensorFlow, compiled, 2017. `https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html`. 2.1

[110] H. Lee, W. Ruys, I. Henriksen, A. Peters, Y. Yan, S. Stephens, B. You, H. Fingler, M. Burtscher, M. Gligoric, et al. Parla: A Python Orchestration System for Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '22, 2022. 5.4.3

[111] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966. 4.2.3.2

[112] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. 1, 2.6

[113] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness. *Communications of the ACM*, 58:44–46, Jan. 2015. doi: 10.1145/2644805. 4.1.1

[114] A. Lopes, F. Pratas, L. Sousa, and A. Ilic. Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 259–268, 2017. doi: 10.1109/ISPASS.2017.7975297. 2.7

[115] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. 2021. doi: 10.48550/arXiv.2102.04664. 4.4

[116] J. Mack, S. E. Arda, U. Y. Ogras, and A. Akoglu. Performant, Multi-Objective Scheduling of Highly Interleaved Task Graphs on Heterogeneous System on Chip Devices. *IEEE Transactions on Parallel and Distributed Systems*, 33(09):2148–2162, Sept. 2022. doi: 10.1109/TPDS.2021.3135876. 2.7, 5.4.1

[117] P. A. Martínez, G. Bernabé, and J. M. García. POAS: A framework for exploiting Accelerator-Level Parallelism in heterogeneous environments. *Under review*. 6.3.1, 6.3.2

[118] P. A. Martínez, G. Bernabé, and J. M. García. HDNN: a cross-platform MLIR dialect for deep neural networks. *The Journal of Supercomputing*, 78(11):13814–13830, July 2022. doi: 10.1007/s11227-022-04417-3. 3.5, 5.1.1, 6.3.1, 6.3.2

[119] P. A. Martínez, B. Peccerillo, S. Bartolini, J. M. García, and G. Bernabé. Applying Intel's oneAPI to a machine learning case study. *Concurrency and Computation: Practice and Experience*, 34(13):e6917, 2022. doi: 10.1002/cpe.6917. 5.1.1, 5.4.3, 6.3.1

[120] P. A. Martínez, B. Peccerillo, S. Bartolini, J. M. García, and G. Bernabé. Performance portability in a real world application: PHAST applied to Caffe. *The International Journal of High Performance Computing Applications*, 36(3):419–439, 2022. doi: 10.1177/10943420221077107. 3.2.1, 4.1.1, 5.1.1, 5.4.3, 6.3.1

[121] P. A. Martínez, J. Woodruff, J. Armengol-Estapé, G. Bernabé, J. M. García, and M. F. P. O'Boyle. Matching Linear Algebra and Tensor Code to Specialized Hardware Accelerators, Jan. 2023. Zenodo. doi: 10.5281/zenodo.7533561. 4.1.2

[122] P. A. Martínez, J. Woodruff, J. Armengol-Estapé, G. Bernabé, J. M. García, and M. F. P. O'Boyle. Matching Linear Algebra and Tensor Code to Specialized Hardware Accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 85–97, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3578360.3580262. 5.1.1, 6.3.1

[123] A. McCaskey and T. Nguyen. A MLIR Dialect for Quantum Assembly Languages. 2021. doi: 10.48550/arXiv.2101.11365. 2.1, 3.5

[124] C. Mendis, J. Bosboom, K. Wu, S. Kamil, J. Ragan-Kelley, S. Paris, Q. Zhao, and S. Amarasinghe. Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. ACM Press, June 2015. doi: 10.1145/2737924.2737974. 4.4

[125] W. Michael Brown, J.-M. Y. Carrillo, N. Gavhane, F. M. Thakkar, and S. J. Plimpton. Optimizing legacy molecular dynamics software with directive-based offload. *Computer Physics Communications*, 195:95–101, 2015. doi: 10.1016/j.cpc.2015.05.004. 2.7

[126] A. Mishra, A. M. Malik, and B. Chapman. Using Machine Learning for OpenMP GPU Offloading in LLVM. In *SC*, 2020. 4.4

[127] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965. 1.2

[128] G. E. Moore. Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13. Washington, DC, 1975. 1.2, 5.1.1

[129] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293. AAAI Press, 2016. 4.2.1

[130] A. C. Murray. *Customising Compilers for Customisable Processors*. Ph.D. Thesis, The University of Edinburgh, Nov. 2012. `http://hdl.handle.net/1842/8028`. 4.4

[131] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. doi: 10.1109/TCAD.2015.2513673. 3.1.1

[132] J. R. Neely. DOE Centers of Excellence Performance Portability Meeting. Apr. 2016. doi: 10.2172/1332474. 1.3, 2.4

[133] D. Nguyen and J. Lee. Communication-Aware Mapping of Stream Graphs for Multi-GPU Platforms. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 94–104, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2854038.2854055. 5.4.1

[134] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen. Exploring API Embedding for API Usages and Applications. IEEE, May 2017. doi: 10.1109/icse.2017.47. 4.4

[135] A. Ni, D. Ramos, A. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues. SOAR: A Synthesis Approach for Data Science API Refactoring. *ICSE*, 2021. doi: 10.48550/arXiv.2102.06726. 4.4, 6.4

[136] R. Nozal and J. L. Bosque. Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime. *Electronics*, 10(19), 2021. doi: 10.3390/electronics10192386. 5.4.3

[137] R. Nozal, J. L. Bosque, and R. Beivide. EngineCL: Usability and Performance in Heterogeneous Computing. *Future Generation Computer Systems*, 107:522–537, 2020. doi: 10.1016/j.future.2020.02.016. 5.4.3

[138] S. Numata, N. Yoshida, E. Choi, and K. Inoue. On the Effectiveness of Vector-Based Approach for Supporting Simultaneous Editing of Software Clones. *Product-Focused Software Process Improvement*, pages 560–567, Nov. 2016. doi: 10.1007/978-3-319-49094-6_41. 4.4

[139] NVIDIA. CUDA C Programming Guide, Jan. 2021. `docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. 1.3, 3.1.1

[140] NVIDIA. CUDA Toolkit Documentation (cuBLAS): Tensor Core Usage, 2022. `https://docs.nvidia.com/cuda/cublas/index.html#tensorop-restrictions`. 5.2.1.1, 5.2.2.3

[141] NVIDIA. Guidelines For Good Performance On Tensor Cores, 2022. https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#tensor-ops-guidelines-for-dl-compiler. 5.2.2.5

[142] NVIDIA. NVDLA, 2022. nvdla.org. 4.4

[143] NVIDIA. NVIDIA Grace Hopper Superchip Architecture. *NVIDIA Whitepaper*, 2022. 1.2

[144] NVIDIA. Tensor Layouts In Memory: NCHW vs NHWC, 2022. https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html#tensor-layout. 5.2.2.5, 5.3.1

[145] H. Ootomo and R. Yokota. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. 2022. doi: 10.48550/arXiv.2203.03341. 5.2.2.6

[146] OpenAI. ChatGPT: Optimizing Language Models for Dialogue, 2022. https://openai.com/blog/chatgpt/. 2.6

[147] X. Ouyang and Y. Zhu. Core-aware combining: Accelerating critical section execution on heterogeneous multi-core systems via combining synchronization. *Journal of Parallel and Distributed Computing*, 162:27–43, 2022. doi: 10.1016/j.jpdc.2022.01.001. 5.4.1

[148] Oyama, Yosuke and Ben-Nun, Tal and Hoefler, Torsten and Matsuoka, Satoshi. Accelerating Deep Learning Frameworks with Micro-Batches. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 402–412, 2018. doi: 10.1109/CLUSTER.2018.00058. 5.2.2.5

[149] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, 2018. doi: 10.1109/HPCA.2018.00067. 4.4

[150] D.-H. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, et al. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits*, 55(4):933–944, 2020. doi: 10.1109/JSSC.2019.2960480. 2.5

[151] V. R. Pascuzzi and M. Goli. Benchmarking a Proof-of-Concept Performance Portable SYCL-Based Fast Fourier Transformation Library. In *International Workshop on OpenCL*, IWOCL'22, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3529538.3529996. 3.5

[152] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 2.3

[153] B. Peccerillo and S. Bartolini. PHAST - A Portable High-Level Modern C++ Programming Library for GPUs and Multi-Cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):174–189, 2019. doi: 10.1109/TPDS.2018.2855182. 1.3, 2.2.1, 1, 3.2.1.3, 3.5, 4.1.1, B.1

[154] B. Peccerillo and S. Bartolini. Task-DAG Support in Single-Source PHAST Library: Enabling Flexible Assignment of Tasks to CPUs and GPUs in Heterogeneous Architectures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2019, Washington, DC, USA, February 17, 2019*, pages 91–100, 2019. doi: 10.1145/3303084.3309496. 2.2.1, 1

[155] B. Peccerillo and S. Bartolini. Flexible task-DAG management in PHAST library: Data-parallel tasks and orchestration support for heterogeneous systems. *Concurrency and Computation: Practice and Experience*, 2020. doi: 10.1002/cpe.5842. 2.2.1, 1

[156] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:102561, 2022. doi: 10.1016/j.sysarc.2022.102561. 5.1.1

[157] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011. doi: 10.1109/RTAS.2011.33. 5.4.1

[158] J. Peng, K. Li, J. Chen, and K. Li. HEA-PAS: A hybrid energy allocation strategy for parallel applications scheduling on heterogeneous computing systems. *Journal of Systems Architecture*, 122:102329, 2022. doi: 10.1016/j.sysarc.2021.102329. 5.4.1

[159] S. Pennycook, J. Sewall, and V. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019. doi: 10.1016/j.future.2017.08.007. 2.4, 3.4.2.1.1

[160] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith. Navigating Performance, Portability, and Productivity. *Computing in Science Engineering*, 23(5):28–38, 2021. doi: 10.1109/MCSE.2021.3097276. 1.3, 2.4, 3.1.1

[161] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen. Statistical Migration of API Usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017. doi: 10.1109/icse-c.2017.17. 4.4

[162] B. Pérez, E. Stafford, J. Bosque, R. Beivide, S. Mateo, X. Teruel, X. Martorell, and E. Ayguadé. Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 125:45–57, 2019. doi: 10.1016/j.jpdc.2018.11.001. 5.4.3

[163] V. Raca, S. W. Umboh, E. Mehofer, and B. Scholz. Runtime and energy constrained work scheduling for heterogeneous systems. *The Journal of Supercomputing*, 78(15):17150–17177, Oct. 2022. doi: 10.1007/s11227-022-04556-7. 5.1.1, 5.4.1

[164] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. *ACM SIGPLAN Notices*, 44:352–362, May 2009. doi: 10.1145/1543135.1542516. 4.2.2.2

[165] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. Survey of Machine Learning Accelerators. *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept. 2020. doi: 10.1109/hpec43674.2020.9286149. 4.4

[166] R. Rigamonti, B. Delporte, A. Convers, and A. Dassatti. Transparent Live Code Offloading on FPGA. 2016. doi: 10.48550/arXiv.1609.00130. 4.4

[167] A. Rodríguez, A. Navarro, K. Nikov, J. Nunez-Yanez, R. Gran, D. Suárez Gracia, and R. Asenjo. Lightweight asynchronous scheduling in heterogeneous reconfigurable systems. *Journal of Systems Architecture*, 124:102398, 2022. doi: 10.1016/j.sysarc.2022.102398. 5.4.1

[168] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and

A. Yasin. Intel Alder Lake CPU Architectures. *IEEE Micro*, 42(3):13–19, 2022. doi: 10.1109/MM.2022.3164338. 1.2

[169] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, et al. Glow: Graph Lowering Compiler Techniques for Neural Networks. 2018. doi: 10.48550/arXiv.1805.00907. 2.1, 3.5

[170] M. Samak, D. Kim, and M. C. Rinard. Synthesizing replacement classes. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371120. 4.4

[171] C. Schlaak, T.-H. Juang, and C. Dubach. Optimizing data reshaping operations in functional irs for high-level synthesis. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, page 61–72, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3519941.3535069. 4.4

[172] C. Software. ComputeCpp, 2020. `https://github.com/codeplaysoftware/computecpp-sdk`. 2.2.2, 3.5

[173] A. Sorokin, S. Malkovsky, and G. Tsoy. Comparing the performance of general matrix multiplication routine on heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 160:39–48, 2022. doi: 10.1016/j.jpdc.2021.10.002. 5.4.2

[174] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. page 205–217, 2015. doi: 10.1145/2784731.2784754. 4.4

[175] M. Steuwer, T. Remmelg, and C. Dubach. Matrix multiplication beyond auto-tuning: Rewrite-based GPU code generation. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10, 2016. doi: 10.1145/2968455.2968521. 4.4

[176] J. D. Stevens and A. Klöckner. A mechanism for balancing accuracy and scope in cross-machine black-box GPU performance modeling. *The International Journal of High Performance Computing Applications*, 34(6):589–614, 2020. doi: 10.1177/1094342020921340. 5.4.1

[177] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, 2010. doi: 10.1109/MCSE.2010.69. 1.3, 2.1, 1, 4.1.1

[178] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, Aug. 1969. doi: 10.1007/BF02165411. 4.3.2

[179] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127:27, 2012. 4.1.1

[180] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 702–714, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2950290.2950321. 4.4

[181] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numerical Behaviors. 2022. doi: 10.48550/arXiv.2206.02874. 2.5

[182] W. Sun, S. Sioutas, S. Stuijk, A. Nelson, and H. Corporaal. Efficient Tensor Cores support in TVM for Low-Latency Deep learning. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 120–123, 2021. doi: 10.23919/DATE51398.2021.9473984. 4.4

[183] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. doi: 10.1109/JPROC.2017.2761740. 2.3, 3.1.2, 3.2.2, 3.4.3.1, 3.4.4.1, 4.1.2, 4.3.5.2, 5.3.1, B.5.3

[184] Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti. RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, DATE '22, page 1099–1102, Leuven, BEL, 2022. European Design and Automation Association. 4.4

[185] Y. Turakhia, G. Bejerano, and W. J. Dally. Darwin: A Genomics Co-Processor Provides up to 15,000X Acceleration on Long Read Assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 199–213, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3173162.3173193. 1.2

[186] R. Uhrie. *Automatic Computational Domain Detection*. Ph.D. Thesis, Arizona State University, Aug. 2021. https://hdl.handle.net/2286/R.2.N.161894. 4.4

[187] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. 2.6

[188] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2020. doi: 10.1109/TCAD.2019.2944584. 5.4.2

[189] Z. Wang, D. Grewe, and M. F. P. O'Boyle. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM Trans. Archit. Code Optim.*, 11(4), 12 2014. doi: 10.1145/2677036. 4.4

[190] Y. Wen and M. F. O'Boyle. Merge or Separate? Multi-Job Scheduling for OpenCL Kernels on CPU/GPU Platforms. In *Proceedings of the General Purpose GPUs*, GPGPU-10, page 22–31, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3038228.3038235. 2.7, 4.4, 5.4.1

[191] Y. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014. doi: 10.1109/HiPC.2014.7116910. 2.7, 5.4.1

[192] J. Weng, A. Jain, J. Wang, L. Wang, Y. Wang, and T. Nowatzki. UNIT: Unifying Tensorized Instruction Compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 77–89, 2021. doi: 10.1109/CGO51591.2021.9370330. 4.4

[193] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009. doi: 10.1145/1498765.1498785. 2.7

[194] S. Winograd. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980. doi: 10.1137/1.9781611970364. 4.3.2

[195] M. Wolfe. Performant, Portable, and Productive Parallel Programming With Standard Languages. *Computing in Science Engineering*, 23(5):39–45, 2021. doi: 10.1109/MCSE.2021.3097167. 1.3, 4.1.1

[196] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle. Bind the Gap: Compiling Real Software to Hardware FFT Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 687–702, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3519939.3523439. 1.3, 2.6, 4.1.1, 4.1.1, 4.1.2, 4.2.3.3, 4.3.2.1, 4.4

[197] H.-I. Wu, D.-Y. Guo, H.-H. Chin, and R.-S. Tsay. A Pipeline-Based Scheduler for Optimizing Latency of Convolution Neural Network Inference over Heterogeneous Multicore Systems. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 46–49, 2020. doi: 10.1109/AICAS48895.2020.9073977. 5.4.2

[198] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583, 2021. doi: 10.1109/HPCA51647.2021.00055. 4.4

[199] Xilinx. triSYCL, 2021. `https://github.com/triSYCL/triSYCL`. 2.2.2, 3.1.1

[200] S. Yesil and O. Ozturk. Scheduling for heterogeneous systems in accelerator-rich environments. *The Journal of Supercomputing*, 78(1):200–221, Jan. 2022. doi: 10.1007/s11227-021-03883-5. 5.4.1

[201] G. Yuan, S. Palkar, D. Narayanan, and M. Zaharia. Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 293–306. USENIX Association, July 2020. 4.4

[202] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):905–918, 2017. doi: 10.1109/TPDS.2016.2586074. 5.4.1

[203] J. Zhang, F. Franchetti, and T. M. Low. High Performance Zero-Memory Overhead Direct Convolutions. In J. Dy and A. Krause, editors, *Proceedings*

of the 35th International Conference on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 5776–5785, Stockholmsmässan, Stockholm Sweden, July 2018. PMLR. 3.2.1.6, A.2, B.5.3

[204] T. Zhao, X. Huang, and Y. Cao. DeepDSL: A Compilation-based Domain-Specific Language for Deep Learning. 2017. doi: 10.48550/arXiv.1701.02284. 3.4.4.2, 3.5

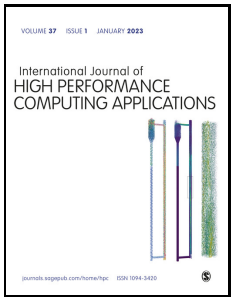
[205] H. Zhou, J. Dong, J. Cheng, W. Dong, C. Huang, Y. Shen, Q. Zhang, M. Gu, C. Qian, H. Chen, Z. Ruan, and X. Zhang. Photonic matrix multiplication lights up photonic accelerator and beyond. *Light: Science & Applications*, 11(1):30, Feb. 2022. doi: 10.1038/s41377-022-00717-8. 4.4

[206] N. Zhou, X. Liao, F. Li, Y. Feng, and L. Liu. List Scheduling Algorithm Based on Virtual Scheduling Length Table in Heterogeneous Computing System. *Wireless Communications and Mobile Computing*, 2021:9529022, Dec. 2021. doi: 10.1155/2021/9529022. 5.4.1

Publications Composing the Thesis

1

Performance portability in a real world application: PHAST applied to Caffe



Title:	Performance portability in a real world application: PHAST applied to Caffe
Authors:	Pablo Antonio Martínez, Biagio Peccerillo, Sandro Bartolini, José Manuel García, Gregorio Bernabé
Journal:	The International Journal of High Performance Computing Applications
JIF:	2.820 Q2 (2021)
Publisher:	SAGE Publications
Volume:	36
Pages:	419–439
Year:	2022
Month:	March
DOI:	10.1177/10943420221077107
Status:	Published

Abstract

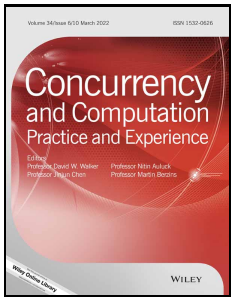
This work covers the PHAST Library's employment, a hardware-agnostic programming library, to a real-world application like the Caffe framework. The original implementation of Caffe consists of two different versions of the source code: one to run on CPU platforms and another one to run on the GPU side. With PHAST, we aim to develop a single-source code implementation capable of running efficiently on CPU and GPU. In this paper, we start by carrying out a basic Caffe implementation performance analysis using PHAST. Then, we detail possible performance upgrades. We find that the overall performance is dominated by few 'heavy' layers. In refining the inefficient parts of this version, we find two different approaches: improvements to the Caffe source code and improvements to the PHAST Library itself, which ultimately translates into improved performance in the PHAST version of Caffe. We demonstrate that our PHAST implementation achieves performance portability on CPUs and GPUs. With a single source, the PHAST version of Caffe provides the same or even better performance than the original version of Caffe built from two different codebases. For the MNIST database, the PHAST implementation takes an equivalent amount of time as native code in CPU and GPU. Furthermore, PHAST achieves a speedup of 51% and a 49% with the CIFAR-10 database against native code in CPU and GPU, respectively. These results provide a new horizon for software development in the upcoming heterogeneous computing era.

Keywords

High-performance computing · Performance portability · Heterogeneous computing · Machine learning.

2

Applying Intel's oneAPI to a machine learning case study



Title:	Applying Intel's oneAPI to a machine learning case study
Authors:	Pablo Antonio Martínez, Biagio Peccerillo, Sandro Bartolini, José Manuel García, Gregorio Bernabé
Journal:	Concurrency and Computation: Practice and Experience
JIF:	1.831 Q3 (2021)
Publisher:	Wiley
Volume:	34
Pages:	1–15
Year:	2022
Month:	June
DOI:	10.1002/cpe.6917
Status:	Published

Abstract

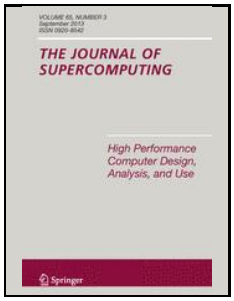
Different technologies and approaches exist to work around the performance portability problem. Companies and academia work together to find a way to preserve performance across heterogeneous hardware using a unified language, one language to rule them all. Intel's oneAPI appears with this idea in mind. In this article, we try the new Intel solution to approach heterogeneous programming, choosing machine learning as our case study. More precisely, we choose Caffe, a machine learning framework that was created six years ago. Nevertheless, how would it be to make Caffe again from the beginning, using a fresh new technology like oneAPI? In terms of not only the ease of programming-because only one source code would be needed to deploy Caffe to CPUs, GPUs, FPGAs, and accelerators (platforms that oneAPI currently supports)-but also performance, where oneAPI may be capable of taking advantage of specific hardware automatically. Is Intel's oneAPI ready to take the leap?

Keywords

Heterogeneous computing · High performance computing · Performance portability · Machine learning.

3

HDNN: a cross-platform MLIR dialect for deep neural networks



Title:	HDNN: a cross-platform MLIR dialect for deep neural networks
Authors:	Pablo Antonio Martínez, Gregorio Bernabé, José Manuel García
Journal:	The Journal of Supercomputing
JIF:	2.557 Q2 (2021)
Publisher:	Springer
Volume:	78
Pages:	13814 – 13830
Year:	2022
Month:	July
DOI:	10.1007/s11227-022-04417-3
Status:	Published

Abstract

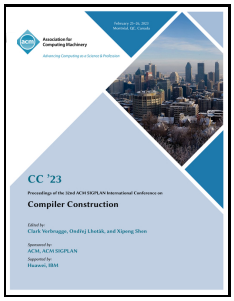
This paper presents HDNN, a proof-of-concept MLIR dialect for cross-platform computing specialized in deep neural networks. As target devices, HDNN supports CPUs, GPUs and TPUs. In this paper, we provide a comprehensive description of the HDNN dialect, outlining how this novel approach aims to solve the P^3 problem of parallel programming (portability, productivity, and performance). An HDNN program is device-agnostic, i.e., only the device specifier has to be changed to run a given workload in one device or another. Moreover, HDNN has been designed to be a domain-specific language, which ultimately helps programming productivity. Finally, HDNN relies on optimized libraries for heavy, performance-critical workloads. HDNN has been evaluated against other state-of-the-art machine learning frameworks on all the hardware platforms achieving excellent performance. We conclude that the ideas and concepts used in HDNN can be crucial for designing future generation compilers and programming languages to overcome the challenges of the forthcoming heterogeneous computing era.

Keywords

High-performance computing · LLVM · MLIR · Heterogeneous computing · Domain-specifc languages · Deep neural networks

4

Matching Linear Algebra and Tensor Code to Specialized Hardware Accelerators



Title:	Matching Linear Algebra and Tensor Code to Specialized Hardware Accelerators
Authors:	Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, Michael F. P. O'Boyle
Conference:	32nd ACM SIGPLAN International Conference on Compiler Construction (CC)
Ranking:	B (CORE2021)
Publisher:	ACM
Volume:	20
Pages:	85 – 97
Year:	2023
Month:	February
DOI:	10.1145/3578360.3580262
Status:	Published

Abstract

Dedicated tensor accelerators demonstrate the importance of linear algebra in modern applications. Such accelerators have the potential for impressive performance gains, but require programmers to rewrite code using vendor APIs - a barrier to wider scale adoption. Recent work overcomes this by matching and replacing patterns within code, but such approaches are fragile and fail to cope with the diversity of real-world codes. We develop ATC, a compiler that uses program synthesis to map regions of code to specific APIs. The mapping space that ATC explores is combinatorially large, requiring the development of program classification, dynamic analysis, variable constraint generation and lexical distance matching techniques to make it tractable. We apply ATC to real-world tensor and linear algebra codes and evaluate them against four state-of-the-art approaches. We accelerate between 2.6x and 7x more programs, leading to over an order of magnitude performance improvement.

Keywords

Program synthesis · GEMM · LLVM · Offloading.