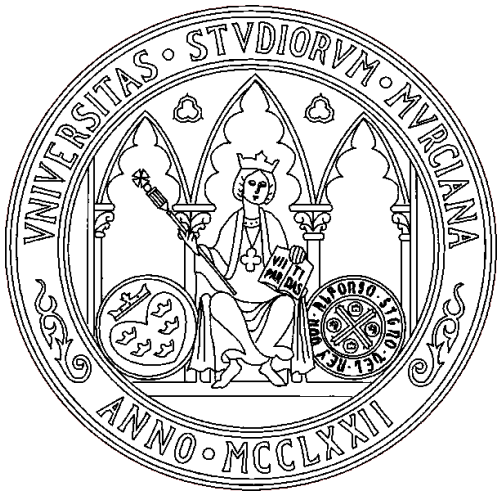


UNIVERSIDAD DE MURCIA
Facultad de Informática

Extending Tensor Core usage for HDNN-MLIR Framework with new data types and operations

TRABAJO DE FIN DE GRADO
Grado en Ingeniería Informática

Presentado por
Alejandro Carmona Martínez

Supervisado por
José Manuel García Carrasco

Murcia, Junio de 2024
Convocatoria de Junio

Universidad de Murcia
Facultad de Informática

Extending Tensor Core usage for HDNN-MLIR Framework with new data types and operations

TRABAJO DE FIN DE GRADO

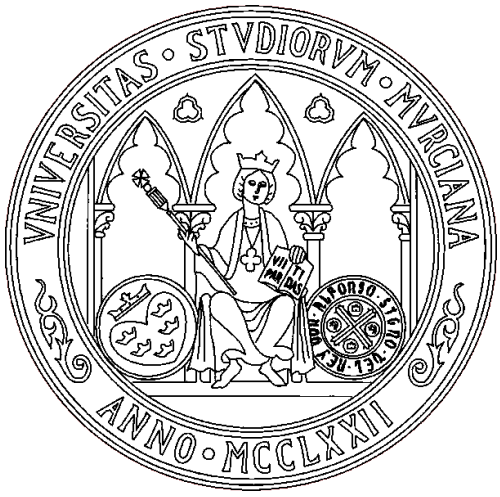
Grado en Ingeniería Informática

Presentado por
Alejandro Carmona Martínez

Supervisado por
José Manuel García Carrasco

Murcia, Junio de 2024

Convocatoria de Junio



UNIVERSITY OF MURCIA
Faculty of Informatics

Extending Tensor Core usage for HDNN-MLIR Framework with new data types and operations

BACHELOR'S THESIS
Grado en Ingeniería Informática

By
Alejandro Carmona Martínez

Advised by
José Manuel García Carrasco

Murcia, June 2024
June call for submissions

University of Murcia
Faculty of Informatics

Extending Tensor Core usage for HDNN-MLIR Framework with new data types and operations

BACHELOR'S THESIS

Grado en Ingeniería Informática

By
Alejandro Carmona Martínez

Advised by
José Manuel García Carrasco

Murcia, June 2024

June call for submissions

Declaration of Authorship

I, Alejandro Carmona Martínez, declare that this bachelor's thesis, titled "*Extending Tensor Core usage for HDNN-MLIR Framework with new data types and operations*", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor's degree at this University.
- Where I have consulted the published work of others to explain or introduce information, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.

Alejandro Carmona Martínez
Murcia, June 2024

Abstract

Moore's Law's decline along with Deep Learning's rise (DL) have led to the development of Domain-Specific Accelerators (DSAs) like GPUs and TPUs, which rely on distinct Domain Specific Languages (DSLs) to optimize their features and hardware such as Nvidia Tensor Cores. Known for maximizing tensor operations performance, these accelerators found in Nvidia GPUs are essential for DL tasks. Heterogeneous Deep Neural Networks (HDNN) convert device-agnostic code to device-specific code, optimizing for CPUs, GPUs, and TPUs using LLVM MLIR's progressive lowering and optimized libraries. However, HDNN underutilized Tensor Cores, missing out on their full potential. This paper presents enhancements which enable HDNN to leverage Tensor Core capabilities effectively. First, user-controlled Tensor Core usage for FP32 data is implemented via FP16 and TF32 downconversion, allowing users to adjust compute modes without modifying the original program. Second, native Tensor Core support for FP16 input/output DL operations is added by including the FP16 MLIR datatype in HDNN. Lastly, a new DL operation, Matrix Multiplication, is integrated, maximizing Tensor Core usage through the previous improvements. Benchmark results show substantial improvements in Tensor Core utilization and ease of use, achieving near-peak performance and often surpassing frameworks like PyTorch.

Resumen extendido

La caducidad de la Ley de Moore, junto con el surgimiento de la Inteligencia Artificial (IA) en su forma actual, ha marcado un punto de inflexión en el campo de la arquitectura de computadores. La Ley de Moore, que postulaba un aumento exponencial en el número de transistores en los circuitos integrados, y por ende, en la capacidad de procesamiento, ha alcanzado su límite práctico. Esto ha llevado a los arquitectos de sistemas informáticos a explorar alternativas para incrementar la potencia de cómputo, más allá de lo que las unidades de procesamiento central (CPU) tradicionales pueden ofrecer.

Al mismo tiempo, la IA y el Deep Learning (DL) han aumentado la demanda de potencia de cómputo para operaciones tensoriales específicas, como la multiplicación de matrices. Esta necesidad ha llevado a la convergencia en la importancia de los Aceleradores Específicos de Dominio (DSAs) como FPGAs, TPUs, extensiones vectoriales para CPUs y el rey de los DSA, la GPU. A pesar de ser el acelerador por defecto en DL, con el objetivo de aumentar el rendimiento en operaciones tensoriales, los Tensor Cores, un circuito diseñado específicamente para ejecutar una multiplicación de matrices con alto rendimiento, fueron introducidos por parte de Nvidia en 2017.

A pesar del rendimiento superior que ofrecen los DSAs, surge un nuevo problema: la programación de estos dispositivos. Cada DSA generalmente requiere su propio Lenguaje Específico de Dominio (DSL), como CUDA de Nvidia, ROCm de AMD o SYCL de Intel. Esto obliga a los programadores a aprender múltiples DSLs para maximizar el rendimiento, creando varios desafíos en términos de productividad, portabilidad y rendimiento, lo que se conoce como el problema *P3* [44]. La productividad se ve afectada porque dominar cada nuevo DSL requiere una considerable inversión de tiempo y esfuerzo, aumentando los tiempos de desarrollo y el riesgo de errores. La portabilidad es problemática, ya que idealmente un código base único debería ejecutarse en diversas plataformas con

mínimas modificaciones, pero en la práctica cada DSA requiere cambios significativos, disparando los costos de desarrollo y mantenimiento. El rendimiento es otra preocupación, ya que los DSAs, aunque son capaces de ofrecer ese alto rendimiento para tareas específicas, requieren una optimización detallada, además de un profundo conocimiento del hardware subyacente, limitando el rendimiento final alcanzable.

Heterogeneous Deep Neural Networks (HDNN) [44], propuesto por Pablo Antonio Martínez Sánchez, busca abordar estos problemas. HDNN es un dialecto basado en MLIR (Multi-Level Intermediate Representation) [13] diseñado específicamente para operaciones propias de redes neuronales profundas (DNN) con soporte para realizar operaciones de convolución y softmax sobre tensores y compatible con una gama de hardware que incluye CPUs, GPUs y TPUs. Los programas HDNN son inherentemente portables, lo cual es facilitado por un ecosistema basado en MLIR que emplea una estrategia de *progressive lowerings* [44], la traducción progresiva en etapas de operaciones de alto nivel y agnósticas del hardware en operaciones específicas de dispositivos a bajo nivel. De esta manera, se asegura que el código pueda ejecutarse eficazmente en diversas plataformas mediante el uso de bibliotecas optimizadas por los fabricantes. De esta manera, HDNN consigue productividad y portabilidad al permitir que un solo programa se despliegue en varios DSAs con mínimas modificaciones, gracias a su arquitectura basada en MLIR, al tiempo que se aprovecha de bibliotecas optimizadas para lograr un alto rendimiento.

Sin embargo, a pesar de las bondades de HDNN para distintos dispositivos, en cuanto a la GPU, los Tensor Cores estaban, como detallaremos a continuación, siendo infrautilizados. Impidiendo alcanzar el máximo rendimiento en programas ejecutados en GPU y negando una de las grandes ventajas de HDNN. La meta de este proyecto es presentar e implementar mejoras a HDNN para aprovechar plenamente las capacidades de los Tensor Cores.

Introducidos por primera vez en la arquitectura Volta, los Tensor Cores ofrecen un rendimiento en operaciones tensoriales de multiplicación y acumulación de matrices, MMA por sus siglas en inglés, significativamente superior a los CUDA cores, lo que los hace ideales para cargas de trabajo de DL [19].

Este espectacular rendimiento se debe a que los Tensor Cores son unidades funcionales diseñadas específicamente para realizar estas operaciones MMA, a diferencia de los CUDA Cores, que están más orientados a procesamiento paralelo y gráfico, habituales en el campo de las GPU. Los Tensor Cores admiten varios formatos de entrada y salida como FP16, TF32, BF16, INT8 e INT4. La menor precisión de estos datatypes respecto a FP32 permite un mayor rendimiento en

ciertas tareas de DL en la que una alta precisión que no es necesaria. TF32, por ejemplo, combina el rango de precisión de FP32 con la velocidad de FP16 [16].

Sin embargo, aunque los Tensor Cores ofrezcan más rendimiento pico [15], siguen utilizando el mismo subsistema de memoria que los CUDA Cores, así que teóricamente solo se obtendrá un mayor rendimiento a partir de un cierto umbral de intensidad aritmética.

Para hacer uso de los Tensor Cores, su programación se realiza principalmente a través de bibliotecas de alto nivel de CUDA. CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela creada por NVIDIA que permite a los desarrolladores crear aplicaciones de alto rendimiento aceleradas con GPU, mejorando significativamente el rendimiento en aplicaciones susceptibles de procesamiento paralelo [17].

En este proyecto, la explotación de los Tensor Cores vendrá de la utilización de las bibliotecas CuBLAS [1] para álgebra lineal de alto rendimiento y CuDNN [6] para redes neuronales profundas (DNN). Estas bibliotecas permiten la encapsulación de la funcionalidad de los Tensor Cores en funciones transparentes al programador.

Con el objetivo de ahondar en las posibles capacidades y rendimiento de HDNN con el uso de los Tensor Cores, se ha implementado las funcionalidades detalladas a continuación:

Uso de Tensor Cores controlado por el usuario para datos FP32. El usuario ha de ser capaz de seleccionar el uso de Tensor Cores o CUDA Cores en programas de HDNN para datos de entrada/salida FP32. Para ello, se implementa la selección entre CUDA Cores o Tensor Cores en línea de comandos al ejecutar el programa HDNN, que en caso de emplear Tensor Cores, también permite la conversión descendente a FP16 y TF32 con el objetivo de priorizar la precisión o el rendimiento:

- **cuda-core-fp32:** Evita el uso de Tensor Cores y se basta de los CUDA Cores para todas las operaciones. Emplea FP32 en los cálculos y acumulaciones.

- **tensor-core-tf32:** Permite operaciones con entrada y salida en FP32 con TF32 como datatype para el cómputo para aprovecharse del uso de los Tensor Cores.

- **tensor-core-fp16:** Convierte entradas, de forma transparente al usuario, de FP32 a FP16 para utilizar Tensor Cores. Sin embargo, los resultados de la convolución también se acumulan en FP32.

Los modos de cómputo ajustables pueden ser fácilmente configurados por el usuario para cada programa sin necesidad de realizar cambios en el código del programa original. Esta flexibilidad garantiza la utilización efectiva de los Tensor Cores, mejorando tanto la usabilidad como el rendimiento o la precisión, según requisitos específicos. Se pueden utilizar tanto para operaciones de convolución como para operaciones de multiplicación de matrices.

Uso nativo de operaciones FP16 para Tensor Cores. Se ha añadido soporte nativo sin conversión FP32 a FP16 para todas las operaciones de HDNN, incorporando el tipo de dato FP16 en el dialecto MLIR de HDNN. Esta mejora permite a HDNN utilizar los Tensor Cores para operaciones de aprendizaje profundo con entrada y salida en FP16, esenciales para lograr un alto rendimiento y eficiencia en las aplicaciones de inteligencia artificial modernas. Al admitir el tipo de dato MLIR FP16, HDNN puede valerse de los Tensor Cores para realizar operaciones tensoriales con un speedUp de entre 3-6x sobre el uso de FP32 en CUDA Cores, lo que conduce a un procesamiento más rápido y eficiente. Este uso de los Tensor Cores se puede emplear tanto con la convolución como en la multiplicación de matrices explicada a continuación.

Multiplicación de matrices a través de la biblioteca CUDA CuBLAS. La introducción de una nueva operación en HDNN, la Multiplicación de matrices, viene de la mano de la biblioteca CUDA CuBLAS, nuevamente implementada dentro del dialecto MLIR de HDNN. La multiplicación de matrices es una operación fundamental en muchos algoritmos de aprendizaje profundo, y su implementación eficiente es fundamental para el rendimiento general. Al integrar la biblioteca CuBLAS, HDNN puede aprovechar rutinas de GPU altamente optimizadas para la multiplicación de matrices, garantizando un rendimiento máximo y compatibilidad con futuras operaciones por añadir.

Todas las funcionalidades añadidas son compatibles entre sí, permitiendo el uso de modos de cómputo y FP16 no solo en la convolución, sino también en la multiplicación de matrices. Es de reparo comentar que estas funcionalidades solo se han añadido para la convolución y multiplicación de matrices, puesto que estas son las operaciones que mayor aprovechamiento de Tensor Cores y FP16 pueden hacer, mientras otras operaciones de HDNN como softmax no tienen soporte para el uso de Tensor Cores en su librería correspondiente, CuDNN en este caso [6].

La evaluación de rendimiento de HDNN con estos añadidos, la denominada versión 3 (v3), se ha llevado a cabo utilizando una NVIDIA GeForce RTX 4090, debido a que presenta Tensor Cores de 4a generación con soporte para TF32. Todos los modos de cómputo son evaluados desde `cuda-core-fp32` hasta tensor-

core-tf32 y tensor-core-fp16 además de la computación y acumulación en FP16. Mediante bancos de pruebas para la multiplicación de matrices y capas convolucionales usando CuBLAS, CuDNN y PyTorch, se ha comparado el rendimiento de HDNN con el objetivo de evaluar si se incurre en un *overhead* o sobrecosto por el uso de HDNN en todos sus modos de cómputo frente al uso por separado de las librerías en las que se basa. El uso de PyTorch, por otro lado, nos sirve para comparar el rendimiento de HDNN contra una alternativa común a la hora de describir Redes Neuronales que también hace uso de las mismas librerías del fabricante, CuBLAS para la multiplicación de matrices y CuDNN para la convolución [36].

Los resultados muestran que los modos con Tensor Cores superan a los CUDA Core, siendo la computación y acumulación en FP16 la que alcanza el mayor rendimiento en ambas operaciones, sobre todo en multiplicación de matrices. El rendimiento de HDNN es comparable al de CuBLAS utilizándose en código CUDA compilado con tanto el compilador de Nvidia, nvcc, como con clang, el compilador para el runtime usado por HDNN con soporte para CUDA. El motivo de usar ambos compiladores es que más allá de evaluar un posible *overhead*, para lo cual se realizan las pruebas en clang, también se busca averiguar si hay una diferencia significativa respecto a un programa CUDA con el compilador provisto por el propio fabricante.

Al ejecutar la multiplicación de matrices en HDNN, este supera claramente a PyTorch, especialmente con tamaños de matriz más pequeños, independientemente del modo de cómputo de HDNN usado. Además, como ya se ha explicado, HDNN permite operaciones con acumulación en FP16, una funcionalidad que PyTorch no presenta y que permite aumentar aún más la ventaja de rendimiento. En comparación con el código CUDA compilado con clang, HDNN demuestra un *overhead* insignificante y obtiene resultados prácticamente idénticos. Sin embargo, para tamaños de matriz más grandes y cálculos de precisión mixta (cálculo en FP16, acumulación en FP32), CuBLAS compilado con nvcc mejora los resultados de HDNN ligeramente.

En cuanto a la convolución, para entradas altamente limitadas por memoria, la aceleración via Tensor Cores no siempre proporciona un aumento significativo de velocidad, aunque es posible. En estos casos, dependiendo de las características de la convolución, esta puede favorecer a HDNN o PyTorch.

En convoluciones limitadas por cálculos matemáticos o *compute-bounded* al tener una alta intensidad aritmética, el uso de Tensor Cores se traduce en mejoras de rendimiento notables. A pesar de que PyTorch parece utilizar una implementación diferente del uso de CuDNN, que permite superar a HDNN en

convoluciones de estas características, cuando se utilizan Tensor Cores con datos FP16 tanto para cómputo como para acumulación, HDNN obtiene una ligera ventaja de rendimiento.

El futuro desarrollo de HDNN se enfocará en varias áreas clave. En primer lugar, la expansión del dialecto MLIR para admitir una mayor variedad de operaciones y tipos de datos, como BF16 e INT8, maximizará su compatibilidad con aceleradores de hardware modernos. Actualizar la versión de clang para soportar las últimas versiones de CUDA abrirá la puerta a nuevas optimizaciones y características, mejorando el rendimiento, especialmente en multiplicaciones de matrices de gran escala.

Además, con la integración de la API de Gráficos de NVIDIA en lugar de la API Legacy, se podría conseguir una gestión más eficiente de tareas de GPU, reduciendo la latencia y mejorando el rendimiento, lo que podría acercar HDNN a PyTorch en ciertos casos de uso en los que este le vence actualmente.

De esta manera, HDNN podrá seguir evolucionando como un framework sólido y versátil, capaz de satisfacer las demandas del progreso en el campo del Deep Learning.

Acknowledgments

In this section, I'd like to acknowledge and thank every person who has helped or me contributed to this work in some way or another in my mother tongue, Spanish.

En primer lugar, quiero agradecer a mi tutor, José Manuel García Carrasco, por su apoyo y ayuda durante todo el proceso del Trabajo de Fin de Grado y todas sus etapas. He sido su alumno interno durante 2 años ya, y en ese tiempo me ha visto evolucionar y madurar tanto a nivel académico y personal. También a Daniel Antonio Martínez Sánchez por el mantenimiento del clúster donde se han realizado las pruebas a lo largo del proyecto.

Gracias a mi familia, en especial a mis padres, Mari y José, por darme las herramientas para poder llegar hasta este momento. A mi hermano Guillermo, por enseñarme mucho a pesar de ser mi hermano pequeño.

Gracias a mi pareja, Sophia por todo su apoyo en momentos difíciles y de duda. Por recordarme que la vida no es un examen o un trabajo y siempre estar de mi lado, incluso cuando ni yo mismo lo estaba.

Gracias a mis amigos de la carrera, Bea, Dani, Germán, Gonzalo, Jaime, Javi, Pablo por ser más que compañeros de clase y por el vínculo que nos une sin el cual no hubiera aprendido la mitad de lo aprendido.

Gracias a mis amigos del instituto, Ana Paula, Ángel, Carlos, Celia, Fran Lax, Miriam y Sebas por compartir conmigo la adolescencia y la adultez, haciéndome ver el mundo desde otras perspectivas.

Finalmente, gracias a mi buen amigo y compañero de carrera Fran Cortés por su amistad, por introducirme a la informática y por su apoyo y ayuda en lides académicas y personales durante estos años.

Gracias.

Contents

List of Figures

List of Tables

Chapter 1

Motivation

1.1 AI Demands, Moore's Law and DSAs

Jack of all traits, master of all. Two events have led us to this key moment in time for computer architecture, one death and one birth. The death of Moore's Law caused computer architects to go in new ways to find computing power that could not be extracted from the CPU any more. The birth of Artificial Intelligence (AI) and Deep Learning as we know it today has exacerbated the need for more computing power for a certain set of operations relating tensors, such as Matrix Multiplication. And the both of them have converged into the current importance of Domain Specific Accelerators (DSAs) like FPGAs, TPUs, vector extensions for CPUs and of course, the king of them all, the GPU.

The importance of tensor operations cannot be stressed enough and that is why even with top-notch performance coming from traditional GPUs, Nvidia introduced in 2017 a new specialized DSA for matrix multiplication attached to the GPU itself, the Tensor Cores.

However, this new approach has a new issue: programming all those devices. Usually, each DSA comes with its own Domain Specific Language (DSL), such as Nvidia GPU's CUDA, AMD's ROCm or Intel's SYCL. This creates a situation where in order to write a program that maximizes the performance for each DSA, there is not a program but rather one per accelerator.

1.2 Productivity, Portability and Performance

As vendors delegate to the programmer the need to learn a different DSL to extract the potential of a particular DSA, the challenges of productivity, portability, and performance become increasingly significant.

Productivity is a major concern because each new DSL requires programmers to invest substantial time and effort in learning and mastering it. The learning curve can be steep, especially for DSLs that have unique programming models [44]. This learning process detracts from the time available to develop and optimize the actual application. Furthermore, maintaining proficiency in multiple DSLs becomes a continuous expense, as each language evolves and new versions are released. This complexity reduces overall development speed and increases the potential for errors, as programmers juggle multiple programming environments. For instance, traditional accelerator DSLs like Nvidia's CUDA, AMD's ROCm, or Intel's SYCL cannot run in other vendors devices and are written differently even if they share similar philosophies, although SYCL is compatible with other vendors. An alternative is the use of single-source languages, like SYCL itself, which allow developers to write a program once and execute it on multiple accelerators, as will be detailed later (see chapter 2). However, these languages can be more complex than general-purpose languages like C++, requiring more time to complete a program [44] and may hamper performance relative to the native DSL.

Portability is another critical issue. The portability of a particular language is related to the number of devices and architectures supported [44]. Ideally, a single language codebase should run across various platforms with minimal changes. Unfortunately, the reality is that each DSA often requires significant modifications to the existing code to extract maximum performance. This *fragmentation* means that a program written for one type of hardware in mind might not run well enough or not even run on another without extensive rewriting. Lack of portability hampers the ability to leverage different hardware platforms efficiently, leading to increased development and maintenance costs. It also makes it difficult to adapt applications quickly to new and potentially more powerful DSAs as they become available, locking developers into specific hardware ecosystems. Additionally, the issue of legacy code cannot be overlooked [44]. Many applications rely heavily on extensive codebases written in traditional programming languages, which are not portable. Rewriting these legacy applications to take advantage of modern DSAs can be a daunting and costly task, also leading to a productivity loss.

Performance is the third aspect of interest. While DSAs deliver high performance for specific tasks, achieving this performance may require fine-tuning and optimization within each DSL as previously introduced regarding portability. Moreover, improving performance might cause a loss of portability and viceversa as the specialized nature of DSAs means that the optimal way to exploit their capabilities can differ significantly from one to another. As a result, a programmer must become an expert not only in the DSL, but also in the underlying hardware architecture to fully harness its power. This requirement for deep domain-specific knowledge can limit the performance obtained in the end.

In conclusion, the rise of DSAs has ushered in a new era of potential in computing performance, but it has also introduced significant challenges in terms of productivity, portability, and performance, which are referred to as P3 [44]. The fragmentation of programming efforts across multiple DSLs, the steep learning curves, and the need for continual optimization present substantial barriers to efficient and effective software development. Furthermore, it becomes clear that maximizing one of the three Ps might come at the expense of another one, as they're entangled between themselves. Performance is achieved through native and intensive DSLs use, though it may create a code with very low or even no performance portability, in light of the different qualities and architectures present. Meanwhile, the development time may also increase as the developer familiarize with one or more DSLs. Addressing these issues requires innovative solutions that can unify development efforts and maximize the potential of different DSAs without sacrificing the performance benefits of specialization that these types of DSA provide.

1.3 Problem Context

In this search for P3, **Heterogeneous Deep Neural Network (HDNN)** burst into the scene. Proposed by Universidad de Murcia's Pablo Antonio Martínez Sánchez as one of the components of his Doctoral Thesis [44], HDNN is a proof-of-concept MLIR (Multi-Level Intermediate Representation) [13] dialect and framework specifically designed for deep neural networks, supporting a range of hardware including CPUs, GPUs, and TPUs.

HDNN programs are inherently portable, facilitated by an MLIR-based ecosystem [44] discussed in subsection 2.2.1, it employs a progressive lowering strategy that translates high-level, device-agnostic operations into low-level, device-specific operations, ensuring that code can be effectively executed across different

hardware platforms [44]. To do so, HDNN leverages optimized vendor libraries for performance-critical operations [44]. This strategy not only simplifies the compilation process but also ensures competitive performance by utilizing pre-existing, highly optimized computational routines whenever possible.

By adopting this hybrid approach, HDNN addresses the P3 problem head-on. It enhances productivity by reducing the need for developers to master multiple DSLs, as the same HDNN program can be deployed across various DSAs with minimal modifications. Portability is significantly improved, enabling applications to run efficiently on diverse hardware without extensive rewriting. And, lastly, HDNN's performance takes advantage of the vendor already fine-tuned libraries for a particular DSA.

In summary, HDNN represents a significant step forward in of deep learning and heterogeneous computing. It provides a viable solution to the intertwined challenges of productivity, portability, and performance, offering a more streamlined and efficient pathway for developing high-performance applications in an increasingly diverse hardware landscape.

Yet, HDNN does not extensively use Tensor Cores, which are critical in Nvidia GPU tensor operations in Deep Learning because of their significant speedup against conventional CUDA Cores. Tensor Cores are specialized hardware components within Nvidia GPUs designed to accelerate tensor operations, which are fundamental to deep learning [19]. These cores can deliver a 4-8x overall speedup compared to CUDA Cores when performing matrix multiplications and other tensor-intensive computations, only incurring in a precision loss for FP32 data but not in lower precision formats as FP16, FP8, ... Despite this significant performance boost, the current implementation of HDNN does not extensively leverage Tensor Cores, which presents a substantial opportunity for improvement. Moreover, HDNN does not support FP16 data being used as input/output in any of its operations, which also would hamper Tensor Core performance and use range. Finally, in its operations' repertoire, HDNN does not include a Matrix Multiplication implementation. This operation is key in DL frameworks and as stated before, it benefits greatly from Tensor Core use.

To sum up, HDNN is a very good solution for the P3 problem, but there is room for improvement.

1.4 Goals and scope

Tensor Cores are optimized for matrix multiplications and, though they originally computed in FP16 (16-bit floating-point) other formats like TF32 (32-bit Tensor Float) [16], BF16 or FP8 also were adopted in newer generations. Still, computation may be in a different datatype to accumulation, being able to compute in FP16 or TF32 while storing the output in FP32, for example. To fully harness their capabilities, it is essential to adapt HDNN to utilize these operations where applicable, incorporating Tensor Core support for different operations and datatypes, leveraging their potential to boost performance in DL tasks significantly. And, in order to achieve those objectives, the next goals are proposed:

User-Controlled Compute Modes. By default, when using FP32 input/output for Tensor Cores with CUDA vendor libraries, and currently in HDNN, Tensor Cores usage is enabled. However, they will compute by default in TF32, but even more importantly they are used without consulting the user, which could have negative consequences if more precision is needed rather than maximum performance. To control the usage of Tensor Cores or CUDA Cores for FP32 data in CUDA's libraries, both must be explicitly selected by the user by modifying the code.

As a code-agnostic solution, we propose the introduction of HDNN's **Compute Modes**. Controlled by the user via CLI for each program without modifying the code itself, FP32 Tensor Core computing with FP16 (Mixed Precision) or TF32 downconversion can be enabled or disabled.

Native Support for FP16 Operations. The second goal is to ensure that Tensor Core usage for FP16 deep learning operations is native and always available. This involves adding support for the FP16 datatype within the MLIR framework of HDNN. Half precision data support ensures that deep learning operations can fully utilize the efficiency of FP16 computations on Tensor Cores, achieving the maximum possible performance.

Matrix Multiplication via CuBLAS. Our last goal would be the implementation of a new HDNN operation for matrix multiplication. This operation is to be integrated via the CUDA CuBLAS library within the HDNN MLIR Dialect. CuBLAS is a GPU-accelerated library that provides highly optimized implementations of BLAS (Basic Linear Algebra Subprograms) functions, which are fundamental to many deep learning algorithms.

In summary, these goals aim to optimize HDNN's performance by harnessing the power of Tensor Cores, providing user-friendly control over compute

modes, ensuring native support for essential data types, and implementing core operations with industry-standard libraries.

1.5 Structure of the document

The rest of the document is as follows:

- Chapter 2 explains the essential background to have a better understanding of the work undertaken and the results achieved. It also takes on the current state of the art in heterogeneous programming, studying similar solutions to HDNN.
- Chapter 3 goes through the specific details for each of the aforementioned goals and how they were implemented. It also takes on the methodology and tools used during the development of this project
- Chapter 4 performs an in-depth evaluation of the results achieved with HDNN, comparing it to alternatives and equivalent CUDA programs that use the same libraries as it does.
- Chapter 5 reflects on the whole of the work accomplished and presents future avenues for improvement.

Chapter 2

Background and State of the Art

2.1 GPU Architecture and Tensor Cores

Nowadays, GPUs are massively parallel computing devices and are present in several different fields [44]; most importantly, they are the leading accelerator used in DL. Nvidia GPUs are a set of streaming multiprocessors (SMs) akin to a CPU core composed of different computing cores inside the SM. Parallel-processing-oriented CUDA cores and specialized accelerators like Tensor Cores (TC) are at the same level in the computing hierarchy [40] (see Figure 2.4) and may be seen as functional units inside a core. Tensor Cores are programmable matrix-multiply-and-accumulate (MMA) units present on NVIDIA GPUs [39]. First introduced in Volta, they offer significantly more throughput than CUDA cores (see Figure 2.1), which makes them excellent for DL workloads. Each Tensor Core can perform 64 FMAs per cycle on Turing and Volta and 256 on Ampere [9], or as is commonly represented, they perform MMA operations represented as D = A * B + C, where the operations size is 4×4×4 [45] and 8×4×8 [43] for the respective architectures. Tensor cores initially supported only FP16 for input and FP16 or FP32 for accumulation and output [20] (see Figure 2.2). But now they support a variety of formats for both input and output like TF32, BF16, INT8, and INT4 [39]. In particular, TF32 [16] has the same range as FP32 and the same precision as FP16, but only consists of 19 bits. It balances FP32's higher precision with FP16's faster performance. When using TF32, FP32 inputs are internally converted to TF32, with the accumulation done in FP32 and output as FP32 (see Figure 2.3). Regarding programming, there are different ways to

leverage Tensor Cores, as explored in other works [39]. Still, this project combines CUDA high-level libraries such as CuDNN [6] and CuBLAS [1] with the MLIR ecosystem to maximize performance

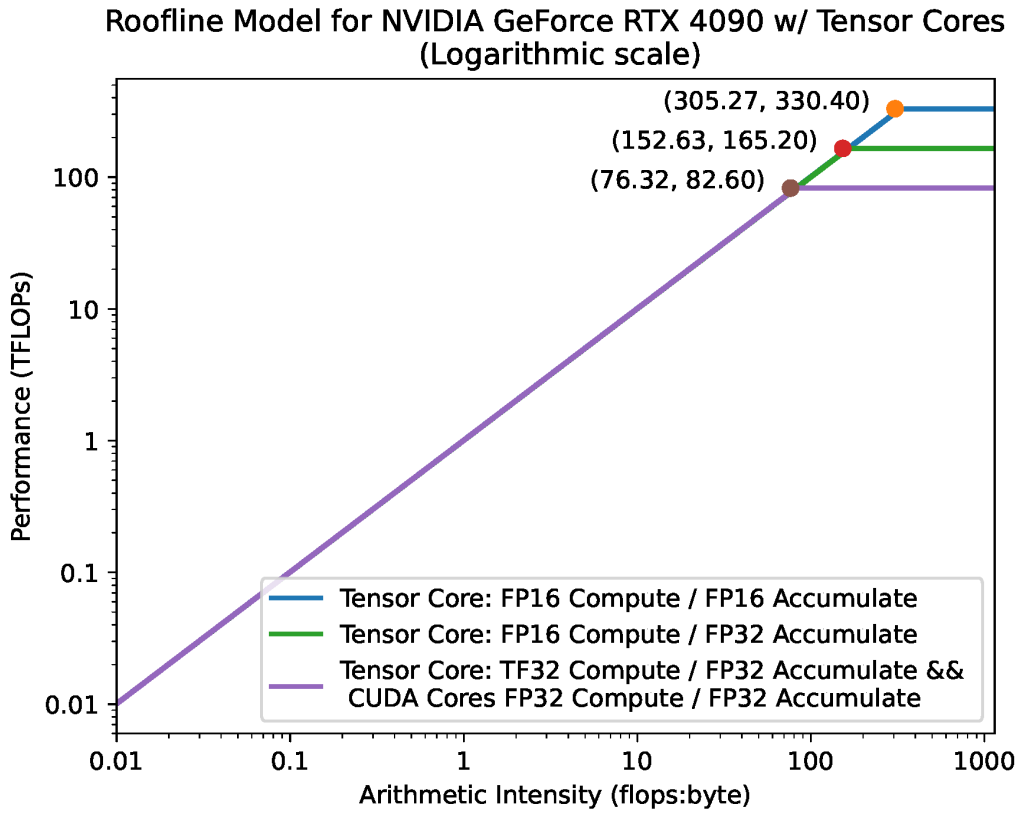


Figure 2.1: Roofline Model in logarithmic scale for NVIDIA GeForce RTX 4090 w/ Compute Modes. Specifications from NVIDIA Ada-Lovelace whitepaper [15]

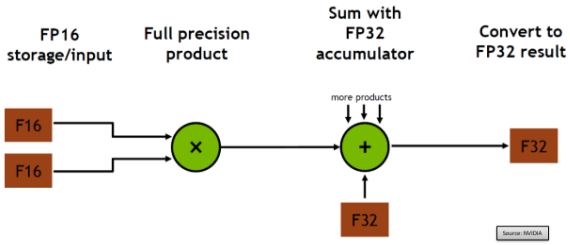


Figure 2.2: Tensor Core FP16 Accumulation [19]

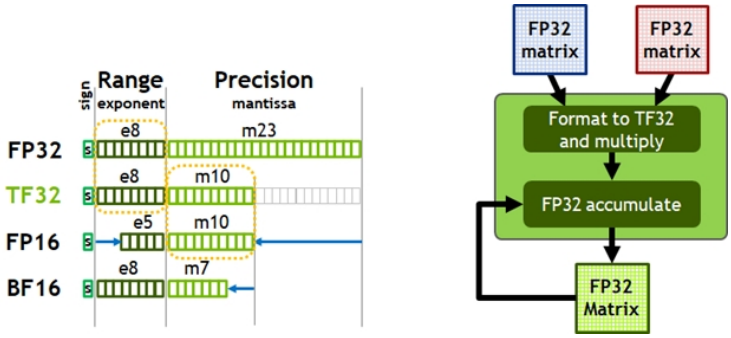


Figure 2.3: *"TensorFloat-32 (TF32) provides the range of FP32 with the precision of FP16 (left). A100 accelerates tensor math with TF32 while supporting FP32 input and output data (right), enabling easy integration into DL and HPC programs and automatic acceleration of DL frameworks."* [29]

Figure 2.4: A100 (GA100) SM architecture [29]

2.1.1 CUDA

"CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA" [33]. It allows developers to utilize NVIDIA GPUs in order to extract higher performance by virtue of GPU-accelerated applications [17] [3], an approach known as GPGPU (General-Purpose computing on Graphics Processing Units) [34].

CUDA is equipped with a wide range of tools, including a hardware driver, an API with its runtime, and optimized mathematical libraries [17]. It supports extensions for various programming languages, enabling a wide range of applications from high-performance computing to deep learning and 3D rendering [3].

In addition to the core components, CUDA also includes domain-specific libraries that provide optimized routines for multiple domains. Notably, cuBLAS is a library for Basic Linear Algebra Subprograms (BLAS) [1], which is essential for high-performance linear algebra computations [2]. Similarly, cuDNN dabbles in deep neural networks, providing highly tuned implementations for deep learning primitives such as convolutions, softmax, ...[4]. Through these libraries' performance, developers are able to leverage the parallel processing power of NVIDIA GPUs, including Tensor Cores as already introduced. CuBLAS works as a more straight-forward typical library, as it can be used with ease to perform Basic Linear Algebra through a few calls. Meanwhile, CuDNN presents a more complex model with two different APIs that may be used, Legacy or Graph. The newer cuDNN Graph API, introduced in cuDNN 8.0, offers a declarative programming model allowing users to build and run computation graphs of tensor operations, enabling more complex operation fusions [23]. In contrast, the legacy API is imperative, easier to use but more restrictive for some tasks [27].

Other specialized CUDA libraries include cuFFT for fast Fourier transforms, cuSPARSE for sparse matrix operations, cuSOLVER for dense and sparse direct solvers, and cuRAND for random number generation [5].

2.1.2 DL operations in GPU: Matmul and Convolution

Since GPUs are designed to handle a high level of parallelism, they excel at tasks with high arithmetic intensity, like matrix multiplication and convolution layers. This is because GPUs can perform a multitude of operations simultaneously, making them well-suited for the parallel nature of operations that can be computed in parallel [37].

Matrix multiplications [8], particularly **General Matrix Multiplications (GEMMs)**, are fundamental to many operations in neural networks such as fully-connected, recurrent, and convolutional layers. GEMMs involve multiplying matrices A and B, with dimensions M x K and K x N respectively, to produce a result matrix C of dimensions M x N. This process requires a significant number of floating-point operations and is influenced by factors such as arithmetic intensity and memory bandwidth.

In the context of deep learning, **a convolution** is a mathematical operation used in convolutional neural networks (CNNs) to process data with a grid-like topology, such as images [21]. It involves sliding a filter or kernel over the input data and computing the dot product of the filter with the input at each position. This process transforms the input into a feature map, highlighting certain features in the input, such as edges or textures, which are crucial for tasks like image recognition [21].

The performance of both operations in deep learning is also closely related to the concept of **arithmetic intensity** (AI), which is the ratio of the number of floating-point operations (FLOPs) to the number of bytes accessed from memory [7]. High arithmetic intensity implies that the computation is more likely to be compute-bound as the roofline model's ridge point is already reached, with performance not limited by bandwidth any more. This aspect will be a very important factor when Tensor Cores are used rather than CUDA Cores, as up until the arithmetic intensity surpasses the roofline ridge's point, Tensor Cores usage might not produce a speedUp, and they may hamper results' precision without any net gains. However, the roofline model is not a perfect approximation, as practical programs cannot usually reach the peak performance for its AI. More on chapter 4.

2.2 HDNN

2.2.1 LLVM-MLIR

"LLVM is a suite of modular compiler and toolchain technologies" [12]. It allows for an increase in flexibility and optimization during the compilation process. This is achieved by use of a compiler framework that translates code to and from an intermediate representation (IR) between source code and the actual compiled output. MLIR (Multilevel Intermediate Representation) [13] [41], an extension of the LLVM project, aims to enhance the efficiency of compiler infrastructure by offering a flexible and extensible IR. Unlike LLVM's single-level IR, MLIR introduces the concept of a multilevel IR, allowing the representation to begin at a high level and be progressively lowered to a lower-level IR through each compiler pass while achieving various optimization goals [44]. A key feature of MLIR is the concept of dialects [14], which define specific sets of operations and types pertinent for each abstraction level or domain, such as Tensor operations or

GPU usage [22]. These dialects can be used within the IR at any stage, offering extensibility and flexibility.

2.2.2 HDNN and MLIR

HDNN is a proof-of-concept MLIR dialect designed for deep neural networks, supporting convolution and softmax layers, as well as basic I/O functionality [44]. It leverages MLIR to ensure device-agnostic source code that can be executed on supported devices. The compilation process in HDNN involves three progressive lowerings or passes, utilizing optimizations from existing dialects and vendor libraries to achieve portability, productivity, and native performance simultaneously. This optimized libraries utilization distinguishes HDNN from other MLIR projects focused on GPU performance [40].

The first pass replaces device-agnostic operations with device-specific counterparts (e.g., `hdnn.conv` to `hdnn.conv.gpu`). It lowers high-level data types (tensors) to lower-level ones (memref). By the end of this pass, HDNN's dialect is only dynamically legal, indicating that device-specific operations have replaced the agnostic ones. In the second pass, calls to the HDNN runtime are inserted, utilizing optimized libraries for layer execution, with only the `mlir-llvm` dialect remaining legal by the pass's end. The compiler handles data management between CPU and GPU using the MLIR GPU dialect, generating GPU dialect operations that undergo transitive lowering to other lower-level operations rather than being translated in a later pass. The third pass automatically converts `mlir-llvm` into LLVM which employs vendor-optimized libraries, thereby circumventing portability issues through a device specific runtime. Libraries such as oneDNN for CPU and cuDNN [6] for GPU are managed by these runtimes and will be embedded in what is to be called the HDNN runtime, facilitating communication between dialect and cuDNN operations for instance [44].

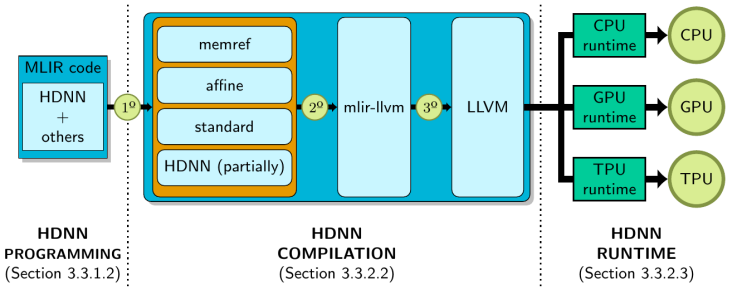


Figure 2.5: HDNN compilation flow [44]

2.3 State of the Art: HDNN Alternatives

Though HDNN is a novel approach to the P3 problem due to its use of MLIR and vendor optimized libraries, there are similar purposed alternatives that try and tackle the aforementioned issues in heterogeneous programming.

These solutions would encompass both device-agnostic Deep Learning Domain Specific Languages (DL DSLs) and more general heterogeneous programming solutions. Designed to abstract away the complexities of underlying hardware, developers are able to write code that can run on various devices without modification, following the P3 principles. These frameworks provide high-level APIs and constructs that automatically optimize and translate code for execution on different types of hardware, such as CPUs, GPUs, and TPUs, which enables developers to focus on writing code rather than dealing with the intricacies of hardware-specific optimizations.

2.3.1 PyTorch

One prominent example with a comparable functionality to HDNN of a device-agnostic DSL Front End for Deep Learning is PyTorch. PyTorch is an open-source machine learning Python package that provides a rich set of tools and libraries for deep learning [36]. It is known for its dynamic computation graph, which allows for flexibility in model architecture and is particularly suited for research and prototyping [31] [36]. PyTorch abstracts away the hardware details, enabling models to be run on different devices like CPUs, GPUs, even with Tensor Core use, and TPU seamlessly. It achieves this through its backend, which includes support for various hardware accelerators and optimized libraries like the aforementioned CuDNN and CuBLAS for NVIDIA GPUs and its Intel equivalents for CPUs, similarly to HDNN (see Figure 2.6).

In contrast to HDNN, which is a MLIR dialect specifically designed for deep neural networks, PyTorch offers a broader ecosystem, a large community and comprehensive documentation. It is a well-established framework in the deep learning community that through its popular Python package acts as a Front End for DL programming.

2.3.2 HPVM

Being that PyTorch is a Front End for device-agnostic programming, Heterogeneous Parallel Virtual Machine (HPVM) is a complete compiler framework

Figure 2.6: PyTorch User Workflow

designed to facilitate hardware-agnostic programming on heterogeneous compute platforms, supporting CPUs, GPUs, FPGAs, and domain-specific accelerators [38] [24], thus it can be considered more similar in concept to HDNN.

"HPVM introduces a hardware-agnostic parallel Intermediate Representation (IR) with constructs for hierarchical task, data, and pipeline parallelism, including dataflow parallelism, and supports multiple front-end languages" [38]. HPVM's IR employs hierarchical graph representation and uses LLVM's IR to represent scalar and vector computations.

In that way, HPVM allows significant performance and energy improvements while maintaining object code portability [38], tackling the same P3 problem with another perspective.

HPVM's front ends support general-purpose parallel code through HeteroC++, a parallel dialect of C/C++, and deep learning frameworks such as the previously mentioned PyTorch. This way, HPVM can be used from PyTorch code to run convolution layers or matrix multiplication operations.

HPVM's back ends generate code for various target devices using a bottom-up traversal of a dataflow graph. For instance, the GPU back end converts the graph's leaves nodes into OpenCL kernels. It also supports tensor operations from high-performance libraries like cuDNN and MKL-DNN through PyTorch Tensor library, sharing HDNN's philosophy and ideas for vendor-optimized library use.

Overall, HPVM provides a robust framework for hardware-agnostic programming of heterogeneous systems, leveraging a different approach to HDNN.

2.3.3 SYCL

While previous heterogeneous alternatives were Deep Learning oriented, SYCL is a C++-based programming model that provides an abstraction layer for writing single-source, device-agnostic code, enabling parallel programming across various hardware platforms like CPUs, GPUs, and FPGAs [11] [32]. It simplifies development by abstracting hardware details and reducing the need for device-specific optimizations [18]. SYCL can be very well-suited for high-expertise

experts in C++ due to both its programming model and fine-grained control over hardware execution [10].

2.3.4 Kokkos

The Kokkos C++ library aims for HPC oriented code to achieve performance portability across a range of devices [35]. Its ensures that user code can be compiled for various devices and perform similarly to device-specific code, making performance portability the primary focus, particularly within high-performance computing (HPC) in mind as it is able to abstract a modern HPC environment as a network of compute nodes, each containing one or more devices [35]. As of current day, HDNN does not offer a multidevice approach to execute one code in multiple nodes even though, it could be a future possibility to enhance its current HPC capabilities.

The Kokkos C++ Performance Portability Ecosystem is divided into three components [25], the Kokkos Core Programming Model, the Kokkos Kernels Math Libraries and the Kokkos Profiling and Debugging Tools. However, only the Kokkos Core Programming Model will be of interest for us. Kokkos Core offers a C++ programming model aimed at crafting applications with performance portability across all principal HPC platforms [26]. It delivers abstractions that facilitate parallel code execution and efficient data handling. Engineered for intricate node structures featuring multi-tiered memory hierarchies and diverse execution resources, Kokkos currently supports backends like CUDA, SYCL, OpenMP, and C++ threads.

Overall, Kokkos' abstraction level provides a crucial extension point for managing increasingly complex device architectures, ensuring that codes are future-proof and capable of adapting to new hardware capabilities. This design is vital in HPC applications and highlights the current and future importance of Heterogeneous Programming in HPC.

2.3.5 Direct performance comparison to HDNN

When evaluating HDNN performance (see chapter 4), although other alternatives might come closer to HDNN ideas, PyTorch will be our benchmark as it is designed for DL operations, and it has a simpler programming model similar to HDNN that is both commonly utilized in DL and easy to use. In addition, when benchmarking to study a possible HDNN overhead, CUDA code employing the vendor optimized libraries will be relied upon.

Chapter 3

Extending HDNN Tensor Core usage

3.1 HDNN: A detailed view

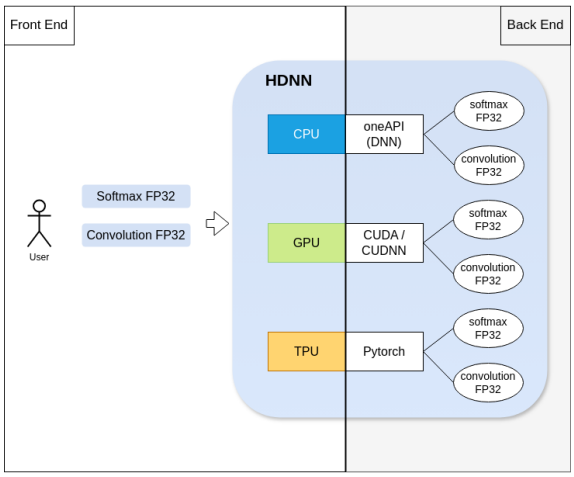


Figure 3.1: Current Vanilla HDNN workflow diagram

Before explaining the modifications intended to be implemented, a deeper understanding of HDNN's current state for GPU usage and the processes taking part is needed. Figure 3.1 and Figure 3.2 reflect both the current HDNN state from the user's point of view (see Figure 3.1) and a compilation standpoint (see Figure 3.2). As previously discussed, HDNN only supports the convolution and softmax layers for FP32 data, with no explicit Tensor Core control.

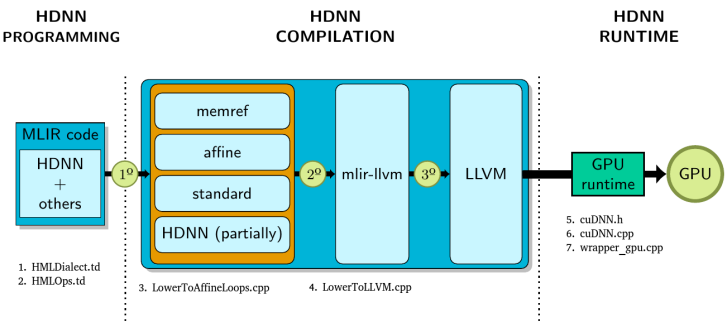


Figure 3.2: HDNN compilation flow with program files for GPU Execution [44]

HDNN employs a clearly divided execution process. For the programming side of HDNN (see Figure 3.3), it relies on the MLIR framework through a dedicated MLIR Dialect. The Dialect is Declared using two .td Tablegen files, one for the Dialect metadata, **HMLDialect.td**, and one for the operations of the Dialect itself, **HMLOps.td**. Particularly, HMLOps.td declares all device-independent and device-specific operations of the Dialect and specifies the arguments' datatypes.

```
1  func @main() -> i32 {
2    hml.launch {dev = "DEVICE_TYPE"} {
3      //images tensor (#batch, in_channels, height, width)
4      %imgs = hml.random() : tensor <1x3x512x512xf32>
5       //filters tensor (out_ch, in_ch, filter_h, filter_w),
6      %weig = hml.random() : tensor <512x3x64x64xf32>
7      //bias 1D bias tensor (out_ch)
8      %bias = hml.random() : tensor <512xf32>
9      //HDNN dialect (hml) convolution operation
10     %out = "hml.conv"(%imgs, %weig, %bias) {iters = 1} : (
          tensor <1x3x512x512xf32>, tensor <512x3x64x64xf32>,
          tensor <512xf32>) -> tensor <1x512x449x449xf32>
11   }
12   hml.return
13 }
```

Figure 3.3: HDNN Convolution program example

When a HDNN program is compiled, its own *hml-opt* compiler is called to execute the three progressive lowerings explained in detail in [44]. The first two lowerings are described in **LowerToAffineLoops.cpp** and **LowerToLLVM.cpp**,

respectively, while the third is completely handled by the MLIR framework itself and does not need of a program file. In these files, each Dialect operation is transformed according to the needs of the pass. In LowerToAffineLoops.cpp, the agnostic HDNN operations are replaced by device specific and the Tensor dialect is converted to memref while the Affine Dialect is used to optimize control sequences in the code.

LowerToLLVM.cpp describes the changes needed to convert all MLIR dialects other than HDNN to the mlir-llvm dialect that allows for direct compilation to LLVM, hence making it possible for the code produced with *hml-opt* to be later compiled with clang-13. This is of utmost significance and the key that opens the door to use vendor libraries as LowerToLLVM.cpp also replaces the previously untouched HDNN device-specific operations with the calls to the GPU Runtime which is written in C++ using CUDA and CuDNN as C++ libraries, which clang does support unlike other C++ compilers.

The calls are described in the file **cuDNN.h** in a format so they can be called passing their LLVM arguments. Meanwhile, they are defined in **cuDNN.cpp** where all modifications needed to call the C++ function using the vendor library are made.

Finally, in **wrapper_gpu.cpp**, a set of functions using the vendor library, in this case CuDNN, are available as a front-end to the library. And as a result, implementation changes are not an issue.

From the user perspective, a Makefile takes a certain .mlir extension HDNN program performs the compilation of said .mlir code using *hml-opt* and clang along with all the necessary compilation flags and middleware. This .mlir is indicated to a newly introduced in this development shell script called *maker.sh* that makes necessary adjustments transparent to the user, such as engaging the correct wrapper to compile from a specific device using a CLI command to invoke maker.sh: `./maker.sh HDNN-PROGRAM] DEVICE-TYPE`.

3.2 Compute Modes and Tensor Core implementation

Firstly, it is worth noting that Tensor Cores are only available from Volta and Turing architecture and onwards. Hence, its inclusion on CuDNN and CuBLAS is relatively recent, and HDNN does not offer Tensor Core direct control. However, it could use Tensor Cores with TF32 (by default without user control) if the convolution algorithm is best suited for a particular program that supports

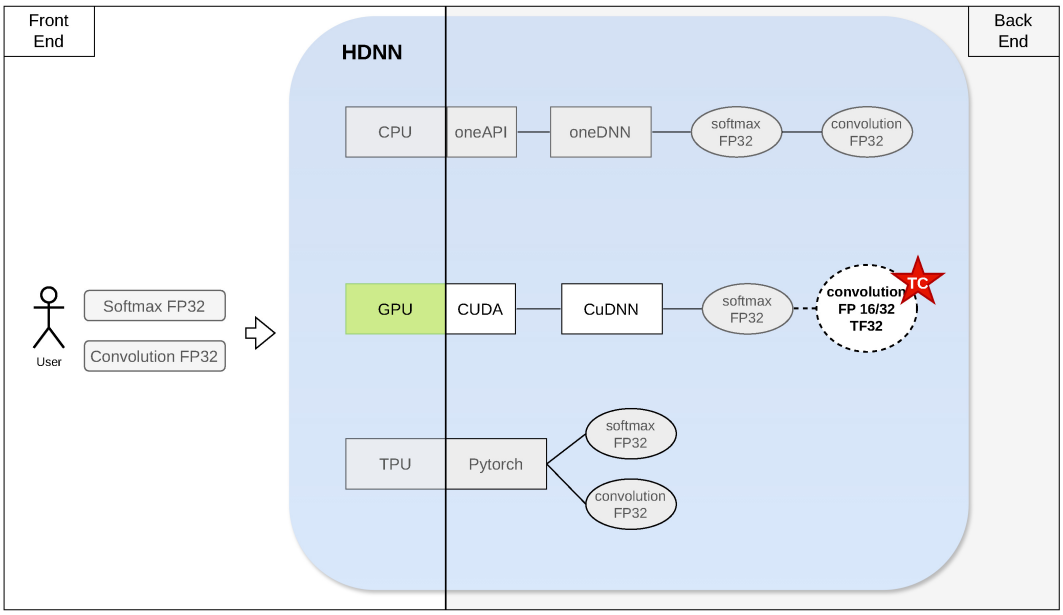


Figure 3.4: Extended HDNN workflow diagram w/ Tensor Core Usage and Compute Modes

Tensor Cores and the hardware is Ampere or superior. But, even if the hardware supported Tensor Cores, it did not allow for an FP16 computation with FP32 accumulation, computing the data as FP16 through conversion but saving the result as FP32, nor did it support the native use of FP16 data.

To efficiently deal with the above situation, three compute modes, **cuda-core-fp32, tensor-core-fp16, and tensor-core-tf32**, are to be implemented into the HDNN Dialect and Runtime for higher precision, performance, and a compromise between the previous two with a caveat, as TF32 is supported since Ampere, and it is also only supported from CUDA 11 onwards. This represents no issue for HDNN as it utilizes the LLVM clang 13 compiler, which is compatible with CUDA 11.2 and can execute in compatibility mode for newer CUDA versions. Still, TF32 only allows for a peak performance gain compared to CUDA core usage with fp32 in specific high-end GPUs like the A100 or H100, though it can offer some performance gains in less powered GPUs. In order to do so, modifications to the GPU Runtime will be made, in particular to wrapper_gpu.cpp to implement the compute mode selection and its Tensor Core use and no MLIR changes will be necesary.

Tensor Cores through cuDNN can be controlled using a macro to establish

a specific math mode before running the convolution [6]. Depending on the computing mode previously selected by the user, a macro would be chosen to allow the HDNN GPU runtime to use CUDA cores or Tensor Cores on the precision selected while keeping the same source code and the Dialect operations the same. In this way, the end user can compile its source code to use a specific accelerator via a simple CLI command:

- `CUDNN_FMA_MATH` avoids Tensor Core for CUDA Cores operations and utilizes FP32 in computation and accumulation. To use it, the `cuda-core-fp32` must be selected.
- `CUDNN_DEFAULT_MATH` permits TF32 Tensor Core operations, but for the convolution layer case, if the algorithm selected does not support Tensor Core usage, Tensor Cores will not be taken advantage of. This macro is linked to the mode `tensor-core-tf32`.
- `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` transforms FP32 inputs into FP16 to use Tensor Cores. However, results are stored as FP32. It is used when the compute mode `tensor-core-fp16` is selected.

Although Tensor Cores, effectiveness may be affected by other factors such as the dimensions of the inputs being a multiple of 4,8 or 16 depending on the datatype used, wave/tile quantization [7] depending on the compiler and hardware used or if the arithmetic intensity is sufficiently high to not be a memory bounded operation. Tensor Cores are engaged through the use of previous macros in CUDA code, CuDNN for this case. In turn, the part susceptible of being modified ought to be HDNN's runtime, in particular, the file **wrapper_gpu.cpp** as it describes the use of the CuDNN library operations.

The macros for Tensor Core use can be set at all times before the actual convolution, but in order to select the best algorithm taking Tensor Cores use into consideration, they must be set before algorithm selection which takes place in wrapper_gpu's `cudnn_find_conv_fwd_algo` method implementation which acts as front to use CuDNN's `cudnnFindConvolutionForwardAlgorithm` to pick the best performing algorithm dynamically for the convolution in place.

Once modified the wrapper, when *maker.sh* is called, a new optional argument can be used to express which compute mode to use: `./maker.sh HDNN-PROGRAM DEVICE-TYPE [COMPUTE-MODE]`. Said selection will be stored in gpu_modes.h variables, which will be updated before compilation starts and consulted when wrapper_gpu is compiled by clang. In this way, Tensor Core use will be dictated by the user from CLI without further changes needed (see Figure 3.5).

In case, the hardware does not support Tensor Cores, even if a compute mode using Tensor Core is selected, the macros will not have effect and CUDA Cores will be used. If Tensor Cores are supported, but GPU architecture is previous to Ampere, TF32 data support wouldn't be available and while `tensor-core-fp16` would work as previously explained, exectuting on `tensor-core-tf32` would be equivalent to `cuda-core-fp32`.

```
extern "C"
cudnnStatus_t cudnn_find_conv_fwd_algo(...){
  cudnnStatus_t cudnnerr = CUDNN_STATUS_SUCCESS;
  /*COMPUTE MODES MODIFICATION BEGIN*/
  //Variables already set up from the maker in gpu-modes.h
  if(cuda_core_fp32 == 1) {
    cudnnSetConvolutionMathType(*conv_desc, CUDNN_FMA_MATH);
  }
  else if(tensor_core_fp16 == 1) {
    cudnnSetConvolutionMathType(*conv_desc,
       CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION);
  }
  else if(tensor_core_tf32 == 1) {
    cudnnSetConvolutionMathType(*conv_desc, CUDNN_DEFAULT_MATH
       ); //Post-Ampere, TF32 is default
  }
  else{
    cudnnSetConvolutionMathType(*conv_desc, CUDNN_FMA_MATH);
  }
  /*COMPUTE MODES MODIFICATION END*/
  /*Algorithm Selection takes place*/
}
```

Figure 3.5: wrapper_gpu method to find the best algorithm with compute mode selection

3.3 Datatypes

Running a convolution layer in HDNN is quite simple, as the code for any accelerator and the newly added compute modes can be reused. Compute modes allow users to use Tensor Cores if needed on FP32 data. On the other hand, Tensor Cores are not running natively in FP16, which could slow the performance

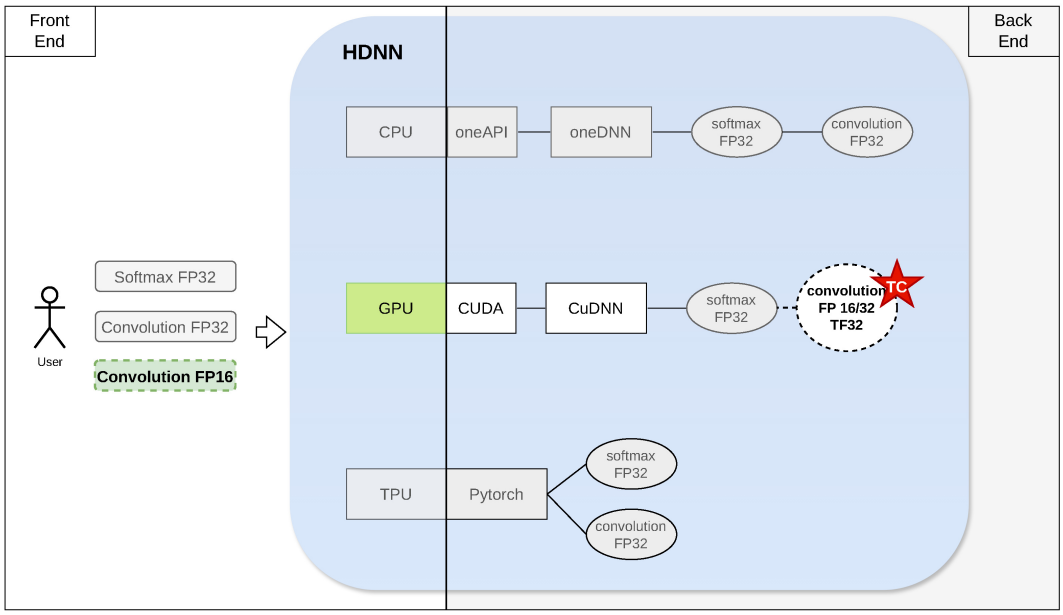


Figure 3.6: Extended HDNN workflow diagram w/ Datatypes (FP16 support)

down. As this development aims to implement Tensor Core usage in HDNN to its full extent, the addition of FP16 data support for the operations that support it, convolution in CuDNN and GEMM in CuBLAS, as explained in the next section, is a must. This Tensor Core usage for fp16, not a compute mode since it is always in use for FP16 data, will be referred to as `native-tensor-core-fp16` not to be confused with `tensor-core-fp16`.

Currently, the user must indicate in the HDNN program (see Figure 3.3) the desired dimensions of the inputs and outputs using the tensor MLIR dialect, but it is restricted to using FP32 data. Any other datatype would trigger an error, as the convolution device-agnostic operation available in HDNN only takes in FP32 tensors. So, the HDNN MLIR dialect must be prepared to handle FP16 tensors through all transformation passes and new CuDNN operations with half-precision arguments and operations are needed to support half-precision in the runtime. To preserve the code reusability and ease of use of the HDNN dialect, HDNN convolution and all other operations used in a program with FP16 data must accept both FP32 and FP16 data as inputs, which would need to be adjusted in the declaration of the dialect operations in a .td file using MLIR tablegen to do a declarative specification (see Figure 3.7).

This dual declaration, in turn, delegates in the Dialect operations that support

```
class HML_ConvolutionT[...]{
  let arguments = (ins
    AnyTypeOf<[F16Tensor, F16MemRef, F32Tensor, F32MemRef]>:
       $imgs,
    AnyTypeOf<[F16Tensor, F16MemRef, F32Tensor, F32MemRef]>:
       $weights,
    AnyTypeOf<[F16Tensor, F16MemRef, F32Tensor, F32MemRef]>:
       $bias
  );
  [...]
}
```

Figure 3.7: MLIR tablegen declaration example for a dialect operation for both FP32 and FP16 tensors and memory references

half and single precision to be able to work with both datatypes; in our case, the Dialect operations are agnostic to the selected datatype and could use any MLIR datatype with the runtime examining the datatypes in compilation time and delegating in the runtime custom methods for single or half precision accordingly. In that way, the user can reuse the same program for an HDNN convolution with FP32 or FP16, with Tensor Cores always being used for FP16 due to its better performance and no precision loss.

So the modifications to be made will leapfrog both progressive lowering files since they are agnostic to the datatypes used and once the operations in HMLOps.td support FP16 data for all operations in a program, they will insert the same calls as for FP32. These calls defined in **cudnn.cpp** will, however, be tasked with making the distinction between FP16 and FP32 data. To do so, in **wrapper_gpu.cpp** there will be two functions to front for CuDNN functionality for each previous function. One will be for FP32 data exactly as before, and the new one will carry out the same functionality for FP16 data.

Implementation will be carried as such:

1. New half methods will be made, as HDNN convolution operation is decomposed into different CuDNN methods, adapting the original float functions to use half data and code accordingly.

2. In cudnn.cpp, new calls to the wrapper for these functions will be used accordingly if the data with which the function is called is either FP16 or FP32.

Starting from wrapper_gpu.cpp, firstly, the function to create the input tensor descriptor needed for a tensor to be used in CuDNN `cudnn_create_tensor` will have a sister function for half descriptor as these need to use the macro `CUDNN_DATA_HALF` instead of `CUDNN_DATA_FLOAT` (see Figure 3.8).

```
1 extern "C"
2 cudnnStatus_t cudnn_create_tensor_half(int ndims, int* dims,
     cudnnTensorDescriptor_t* tensor) {
3 [...]
4   if((cudnnerr = cudnnSetTensor4dDescriptor(*tensor,
       CUDNN_TENSOR_NCHW, CUDNN_DATA_HALF/*CUDNN_DATA_FLOAT*/,
       newd[0], newd[1], newd[2], newd[3])) !=
       CUDNN_STATUS_SUCCESS) {
5     printf("%s: %s\n", __func__, cudnnGetErrorString(cudnnerr
        ));
6     return cudnnerr;
7   }
8 [...]
9 }
```

Figure 3.8: wrapper_gpu method modified to create a FP16 Tensor Descriptor

Similar changes are also made to the functions to create filter and biases descriptors along with the descriptor for the convolution itself.

Then to run the convolution layer with the data from the HDNN program `cudnn_convolution_forward_half` is similarly modelled after `cudnn_convolution_forward` with the main difference being the data types used (see Figure 3.9).

Finally, once there is support for CuDNN functionality in Half Precision, we can use it depending on the data type for a specfic convolution by performing a dynamic data type check (see Figure 3.10).

Of course, we must also add these changes for every CuDNN function that performs a call to a wrapper_gpu.cpp method with half and float versions. However, as the dynamic check is in cudnn.cpp, a call to cudnngen_convolution_forward from LowerToLLVM.cpp can remain the same whether data is half or single precision as intended.

```
cudnnStatus_t cudnn_convolution_forward(cudnnHandle_t* handle,
    cudnnTensorDescriptor_t* imgsT, cudnnFilterDescriptor_t*
   weighD, cudnnTensorDescriptor_t* biasT,
   cudnnTensorDescriptor_t* outpT, float* imgs, float* weights
   , float* bias, float* output, cudnnConvolutionDescriptor_t*
    conv_desc, cudnnConvolutionFwdAlgo_t* algo)

cudnnStatus_t cudnn_convolution_forward_half(cudnnHandle_t*
   handle, cudnnTensorDescriptor_t* imgsT,
   cudnnFilterDescriptor_t* weighD, cudnnTensorDescriptor_t*
   biasT, cudnnTensorDescriptor_t* outpT, __half* imgs, __half
   * weights, __half* bias, __half* output,
   cudnnConvolutionDescriptor_t* conv_desc,
   cudnnConvolutionFwdAlgo_t* algo)
```

Figure 3.9: wrapper_gpu cudnn_convolution_forward method's headers for FP32 and FP16 data

```
Value cudnngen_convolution_forward(Operation* op, [...], Value
    handle, Value imgs,[...]) {
[...]
  /*ConvolutionHalf MOD BEGIN*/
  std::string func_name = "cudnn_convolution_forward";
  auto operand_it = op->operand_type_begin();
  auto memRefTypeTensorImg = (*operand_it).cast<MemRefType>();
  if(memRefTypeTensorImg.getElementType().isa<Float16Type>())
  {
    func_name = "cudnn_convolution_forward_half";
  }
  auto cudnn_convolution_ref = get_cudnn_fun(rewriter,
     parentModule, func_name);
  /*ConvolutionHalf MOD END*/
/*Operation Rewriting is completed*/
}
```

Figure 3.10: cudnn.cpp method modified to make a call to a FP16 or FP32 method

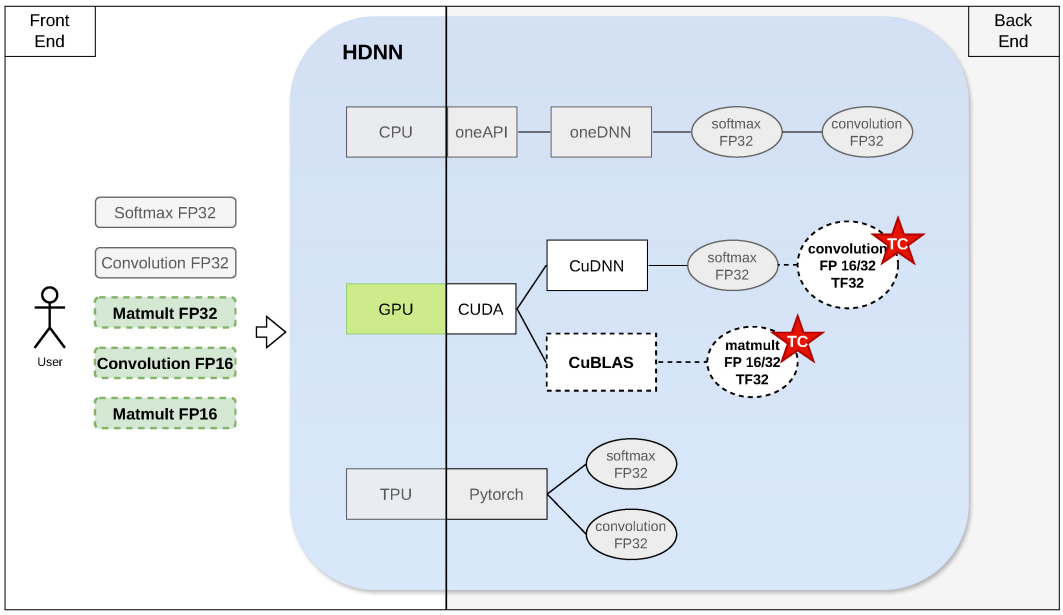


Figure 3.11: Final extended HDNN workflow diagram w/ Operations (CuBLAS Matmul)

3.4 Operations

The additions of more control for Tensor Core use and FP16 native data improve the usability of this accelerator in HDNN. However, Tensor Cores are also primarily used in GEMM operations, which were lacking in HDNN repertoire at release. Since NVIDIA's CuBLAS offers a Tensor Core-supported GEMM operation [8], the integration of this library and its GEMM operation bring flexibility and variety to the current toolkit of operations, but, moreover, it maximizes Tensor Core usage through HDNN. Similarly to the previous section, adding an operation in the dialect means it must be declared in the tablegen file and lowered through the different passes that Dialect operations must complete. It must also declare the datatypes it will use (see Figure 3.7).

After declaring the operation, there must be lowering operations for the first and second passes, transforming the operation into affine and LLVM. In the second pass, the call to the runtime method using CuBLAS is inserted with the input of the matmul function as the arguments to said method, which, in turn, performs the actual GEMM operation along with the compute mode support

previously discussed that now also includes the macros needed by CuBLAS [1] in addition to CuDNN's to use tensor cores if FP32 is in use.

These functions will be called from the dialect using the Runtime calls declared. Though different macros are used in CuBLAS [1] to control Tensor Cores, the compute modes remain the same as in the previous section. User control from the HDNN source code is also available, and the option to run both convolution and matmul using tensor cores for both operations with the selected datatype (FP32 or FP16) is also available.

In conclusion, for the matmul operation to be implemented, it is needed to build from scratch all the steps needed to fulfil all compilation stages (see Figure 3.2).

However, to make use of CuBLAS itself, it needs to be installed and available for use. As previously discussed, CuBLAS is a vendor optimized and provided library. Unlike CuDNN which requires an independent installation process as a library, CuBLAS is included in many CUDA installation methods, such as CUDA-toolkit or docker images for CUDA tagged as *devel*, but its installation is not mandatory.

HDNN's initial version and docker image was built with an ubuntu base image in which all necessary packages were installed, but only had a CUDA and CuDNN installation through a .deb package and did not support CuBLAS operations, however in order to both add CuBLAS support and update CUDA's version for future developments, docker image *nvidia/cuda:12.3.1-devel-ubuntu22.04* [30] was used to rebuild HDNN using it as a base image instead of said ubuntu image.

Once CuBLAS is installed, the first order of business will be to create a function that can be used to perform a matrix multiplication in wrapper_gpu.cpp (see Figure 3.12). To perform the GEMM operation, cublasSgemm takes the dimensions of the tensors besides the tensor themselves and writes the results to an output Tensor [1]. If Half Precision were to be used, __half would've been used instead of float, but moreover, cublasSgemm must be replaced with cublasHgemm in order to perform the same operation for Half data.

In addition to the needed parameters, to perform the actual matmul, a cublasHandle_t object is needed. This object is necessary to perform any CuBLAS operation and must be initialized beforehand. Reason why, wrapper_gpu is host to cublas_create (see Figure 3.13) which will be used when initializing the GPU in the lowering process as is to be explained.

Next, the call to the wrapper_gpu function needs to be made. A new pair of program files equivalent to cudnn.h and cudnn.cpp will be made available in the

```
1  extern "C"
2  cublasStatus_t cublas_matmul(cublasHandle_t* handle, int m,
       int n, int k, float* tensorA, float* tensorB, float* output
       ) {
3    if(cuda_core_fp32 == 1){ cublasSetMathMode(*handle,
       CUBLAS_DEFAULT_MATH); }
4    else if(tensor_core_fp16 == 1) { cublasSetMathMode(*handle,
       CUBLAS_TENSOR_OP_MATH); }
5    else if(tensor_core_tf32 == 1) { cublasSetMathMode(*handle,
       CUBLAS_TF32_TENSOR_OP_MATH); }
6    else{
7      cublasSetMathMode(*handle, CUBLAS_DEFAULT_MATH);
8    }
9    cublasSgemm(*handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, &
       alpha, tensorA, m, tensorB, k, &beta, output, m));
10 }
```

Figure 3.12: wrapper_gpu.cpp cublas_matmul implementation

```
1  extern "C"
2  cublasStatus_t cublas_create(cublasHandle_t* handle, CUstream
       stream) {
3    cublasStatus_t cublaserr = CUBLAS_STATUS_SUCCESS;
4    if((cublaserr = cublasCreate(handle)) !=
       CUBLAS_STATUS_SUCCESS) {
5      printf("ERROR cublasCreate\n");
6      return cublaserr;
7    }
8    if((cublaserr = cublasSetStream(*handle, stream)) !=
       CUBLAS_STATUS_SUCCESS) {
9      printf("ERROR cublasSetStream\n");
10   }
11   return cublaserr;
12 }
```

Figure 3.13: wrapper_gpu.cpp cublas_create implementation

form of cublas.h and cublas.cpp. This files will be in charge of declaring the calls that will be made from LowerToLLVM and implementing them respectively.

As CuBLAS does not use descriptors, it is possible to run a GEMM with just

initializing the handle and passing the arguments directly to wrapper_gpu's cublas_matmul. Figure 3.14 allows for clearly seeing how the call is performed by converting the LLVM arguments to the C++ arguments previously declared (see Figure 3.12) as the Integer and Float values are passed. Note that no code to distinguish Half and Single Precision data is included in order to not clutter the implementation details, but could very well be included as already shown (see Figure 3.10).

```
1  Value cublasgen_matmul(Operation* op, ConversionPattern-
2  Rewriter &rewriter, Value cublas_handle, int m, int n, int k,
      Value tensorA, Value tensorB, Value output) {
3    ModuleOp parentModule = op->getParentOfType<ModuleOp>();
4    OpBuilder &builder = rewriter;
5    auto context = builder.getContext();
6    auto loc = op->getLoc();
7    auto cublas_matmul_ref = get_cublas_fun(rewriter,
       parentModule, "cublas_matmul");
8    // cublasStatus_t cublas_matmul(cublasHandle_t* handle, int
       m, int n, int k, float* tensorA, float* tensorB, float*
       output)
9    ArrayRef<Value> args({cublas_handle, builder.create<LLVM::
       ConstantOp>(loc, IntegerType::get(builder.getContext(),
       32), builder.getIntegerAttr(builder.getIndexType(), m)),
       builder.create<LLVM::ConstantOp>(loc, IntegerType::get(
       builder.getContext(), 32), builder.getIntegerAttr(builder
       .getIndexType(), n)), builder.create<LLVM::ConstantOp>(
       loc, IntegerType::get(builder.getContext(), 32), builder.
       getIntegerAttr(builder.getIndexType(), k)), tensorA,
       tensorB, output});
10   TypeRange results(IntegerType::get(context, 32));
11   mlir::CallOp callop = rewriter.create<mlir::CallOp>(loc,
       cublas_matmul_ref, results, args);
12   Value status = callop.getResult(0);
13   return status;
14 }
```

Figure 3.14: cublas.cpp cublasgen_matmul implementation

As the runtime methods are ready to be used, the next steps involve declaring the operation itself in the dialect, very similarly to what was previously done (see Figure 3.7). It also means to include the operation in the lowering processes that

are implemented in LowerToAffineLoops.cpp. But, since the process remain the same for all operations declared in which they are lowered from device-agnostic to specific and tensor is replaced by memref along with the introuduction of affine, it is not to be discussed in this document.

LowerToLLVM.cpp is a different story, as the calls to the runtime in cublas.cpp need to be made. Firstly, the cublas_handle needs to be initialized along with the CUDA stream and the cudnn_handle that HDNN uses to perform previous operations in the GPU (see Figure 3.15). This will be achieved with the previously implemented calls in cublas.cpp and the CuBLAS functionality in wrapper_gpu.cpp.

```
1  struct InitGPUOpLowering : public ConversionPattern {
2    InitGPUOpLowering(MLIRContext *ctx) : ConversionPattern(hml
       ::InitGPUOp::getOperationName(), 1, ctx) {}
3    LogicalResult matchAndRewrite(Operation *op, ArrayRef<Value>
       operands,
4    [...]
5      // 1. Init cuda stream [...]
6      // 2. Init cudnn handle  [...]
7      //3. Init cublas handle
8      Value dummy_cublas_handle;
9      status_create = cublasgen_create(op, rewriter, &
         dummy_cublas_handle, currentToken->asyncToken());
10     cublas_handle = new Value(dummy_cublas_handle);
11     check_cuda_status(op, rewriter, status_create);
12     rewriter.eraseOp(op);
13     return success();
14   }
15 };
```

Figure 3.15: LowerToLLVM.cpp InitGPUOpLowering implementation

Regarding the Matmul operation itself, as there are explicit lowerings for the GPU and other devices for each operation, there needs to be a Lowering for Matmul, too. Given that Matmul is only to be run on GPU, the only lowering to developed is the GPU one (see Figure 3.16).

In this lowering, the dimensions of the tensor are taken from the HDNN arguments while the tensors must be initialized from the arguments to make the call to cublas.cpp. Once the arguments are ready, the call takes place and the HDNN Matmul operation is replaced by the cublas.cpp method, which in

turn uses wrapper_gpu's cublas_matmul implementation. Although these are the most significant changes, many have been omitted due to brevity when presenting code snnipets and implementation quirks, but many adaptations have been made to include the matmul operation and FP16 datatype additions when it comes to MLIR by use of its documentation [14].

```
1  struct MatmulGPUOpLowering : public ConversionPattern {
2    MatmulGPUOpLowering(MLIRContext *ctx) : ConversionPattern(
        hml::MatmulGPUOp::getOperationName(), 1, ctx) {}
3    LogicalResult matchAndRewrite(Operation *op, ArrayRef<Value>
        operands, ConversionPatternRewriter &rewriter) const
       final {
4      // 0. Get memref shapes of TensorA, TensorB and output
5      auto memRefTypeOut = (*op->result_type_begin()).cast<
          MemRefType>();
6      auto memRefShapeOut = memRefTypeOut.getShape();
7      int ndims_output = memRefShapeOut.size();
8      int *dims_output = (int *) malloc(sizeof(int) *
          ndims_output);
9      for(int i=0; i < ndims_output; i++) {
10       dims_output[i] = memRefShapeOut[i];
11     }
12     [...]
13     Value tensorA = builder.create<LLVM::DialectCastOp>(loc,
          memRefTypeTensorA, operands[0]);
14     Value tensorB = builder.create<LLVM::DialectCastOp>(loc,
          memRefTypeTensorB, operands[1]);
15     // 1. Run matmul
16       for(int i=0; i < iters; i++) {
17         status_matmul = cublasgen_matmul(op, rewriter, *
             cublas_handle, dims_tensorA[0], dims_tensorB[1],
             dims_tensorA[1], tensorA, tensorB, output);
18         check_cuda_status(op, rewriter, status_matmul);
19     }
```

Figure 3.16: LowerToLLVM.cpp MatmulGPUOpLowering implementation

3.5 Methodology

This subsection will work as a mirror in which to briefly reflect the procedure followed to implement all previous changes. Although I, personally, would not go as far as to say Agile methodology or other development philosophies have been wilfully used, some of their core concepts have applied to this development. Regarding the management and deployment technologies, these tools relate more to their specific use during the three different stages of this project's development as there was a learning curve and implementation for every tool, but all are used throughout each stage.

Additionally, I would like to express that the knowledge from the subjects Administración Avanzada De Sistemas Operativos, Administración De Sistemas Operativos y Redes and Administración Avanzada De Redes introduced me to some of the tecniques used in this methodology and gave me an extensive Docker knowledge insight.

3.5.1 Onboarding stage

Firstly, there was a clear preparatory stage where work was focused in gaining an understanding of all subjacent and relevant HDNN components such as LLVM, MLIR, CUDA, CuDNN and even Docker as the HDNN framework is encapsulated in a Docker image with all necessary dependencies and tools with their required versions.

During these stages, my advisor, Jose Manuel, oriented me with small tasks and feature requests to implement in MLIR and CUDA, although most of the time was passed studying some of the bibliography included to gain the requiered knowledge both from a hardware and software perspective. This was a key aspect of this stage and the following ones, as knowing the hardware used has been as important as the software development in an often interleaved cycle, with both parts affecting my understanding of each other.

3.5.2 Setup stage

Once I rose to have a solid base of knowledge, I was given access to the GACOPs Cluster to perform my benchmarks and tests in GPUs with the required Tensor Core Hardware. This stage was not as lengthy as either the previous or next one, but it was the backbone to allow for an environment that was needed for the development, testing and deployment of new functionality.

As multiple hosts were needed to test the performance in different hardware and some hosts of interest dealt with heavier workloads, a portable workspace was needed. Work started on building the repository to test the development versions of HDNN. To do so, a Git repository in GitHub will be deployed containing all dependencies and the dockerfile itself so the user, can download and build the image to the latest specification. This system ensures portability from host to host, with only the image build process required to run HDNN.

3.5.2.1 Git and Docker

HDNN and its source code repository are bundled into a docker image based of Ubuntu. The construction process is written into a dockerfile that had a set of sizeable dependencies needed to build the LLVM MLIR workspace [14] and install all vendor optimized libraries such as CuDNN.

The bigger sized dependencies were an issue for two motives, size restrictions in GitHub and missing updates in some dependencies which could work with newer versions.

The result was a new dockerfile that downloaded the specific or latest version of a dependency from its repository, achieving a higher portability and ease of use. To better integrate the installation of CUDA, the base Ubuntu image was replaced by the CUDA 12.3 devel image [30], too. This image already contained CUDA and CuBLAS, simplifying the installation process of many dependencies and allowing for future straight-forward CUDA version changes.

In summary, the workspace allowed for a shared code repository that could be updated from any host in the cluster or not, allowing for work in two different hosts to be synchronized which allows to easily switch between hosts depending on the needs of testing and the current workloads.

3.5.2.2 GACOP Cluster

During this project, I first used a cluster to perform benchmarks but also develop as the Git/Docker setup allowed it.

There was a new familiarization process to learn the in and outs of cluster utilization and sharing with other users. GACOP's cluster employs Slurm Workload Manager as queuing system to ensure that the jobs launched by each user are not in conflict concerning the hardware resources. To make the most out of this system, Slurm shell scripts have been used to execute different programs inside a Docker container from the new HDNN image. Slurm scripts allow for easy portability between hosts by setting the queue to which send the workload.

To allow for interactivity for the development/testing in real time, srun was also available. This allowed for more flexibility in changes that could be pushed to the Git repository, while Slurm allowed for very big workloads repeated for different configurations and containers, too. The latter part was of much use when other containers were used to benchmark in both CUDA with nvcc and python as will be explained in the next chapter (see chapter 4).

3.5.3 NVIDIA Nsight Systems

Finally, to better analyze GPU behaviour and Tensor Core usage, NVIDIA Nsight Systems were included in the docker build process and used to obtain several key information to evaluate the results beyond the time performance obtained. NVIDIA Nsight Systems gives a full picture of app performance [28], but it was specially important to evaluate if Tensor Core were actually being used and how much they were being used, besides which data transformation occurred when used.

3.5.4 Development and Testing Stage

This final stage encapsulates the development of the previously explained new HDNN functionalities.

During this stage, José Manuel participated as both the Software client asking for some functionality and advisor, exploring ways to fulfil the functionality and understanding the meaning of the results obtained.

Although, the functionalities implemented are independent, the development followed was carried out taking into account that all functionalities must work in conjunction when finished. When a functionality was made available and compatible with all other features, a new git commit was performed, allowing to keep a version history if some modification resulted in an unforeseen error or behaviour.

To conclude, this approach and the work undertaken in every previous stage to the development and testing of the functionalities enabled me to work efficiently, maintain high-quality standards, and swiftly adapt to changes in either the functionality, code or workspace host machine, ultimately ensuring the project's successful completion.

Chapter 4

Performance evaluation

4.1 Testbed

To benchmark this version of HDNN with the enhancements included, codenamed v3, an NVIDIA GeForce RTX 4090 from GACOP's cluster was used due to its 4th generation Tensor Core with TF32 support, allowing all compute modes available and native Tensor Core use on HDNN to be tested. The following subsection presents HDNN performance for all compute modes to observe the different performances from the compute and accumulation types as implemented in section 3. Then, different matrix multiplication and convolution inputs will be used to compare these HDNN operations to an equivalent implementation of CuBLAS and CuDNN for each compute mode and PyTorch. With these benchmarks, the main objectives are to determine if tensor cores are being used to their full potential, if there is a significant performance loss through the use of HDNN, and how HDNN compares to other DSLs with a similar purpose.

For CuBLAS, CUDA 12.3 with its nvcc compiler is used, while CuDNN's benchmark runs on CuDNN v9.1 and uses the same version of CUDA and nvcc as its CuBLAS counterpart. PyTorch version is (v2.3.0+cu121).

Finally, as previously stated, HDNN uses LLVM-MLIR and is compiled with clang 13 as of this version. Clang 13 supports up to CUDA 11.2, so even though HDNN v3's CuBLAS and CuDNN versions are 12.3 and 9.1.1, it runs in a compatibility mode. That's why, to evaluate the difference between compilers and the overhead induced by HDNN, both CuDNN and CuBLAS benchmarks will be run with nvcc and clang 13.

4.2 Compute Mode Performance

Two benchmarks have been prepared to test the different compute mode implementations with a sufficient arithmetic intensity (AI) to exploit all compute modes and datatypes. First is a benchmark testing a matmul operation (M=N=K=4096) in all compute modes; second, a convolution using input 3 (see Table 4.3).

Table 4.1: Combined performance metrics for Matmul and Convolution operations.

Matmul (M=N=K=4096) Performance (TFLOPS)			
native-tensor-core-fp16	**cuda-core-fp32**	**tensor-core-fp16**	**tensor-core-tf32**
282.80	51.18	158.16	81.83
Convolution (Input 3) Performance (Time in ms)			
native-tensor-core-fp16	**cuda-core-fp32**	**tensor-core-fp16**	**tensor-core-tf32**
200.709	510.363	294.417	293.312

As the table suggests, there is a clear performance difference for each compute mode, with tensor-core modes always faster than CUDA cores and `native-tensor-core-fp16` (FP16 compute and accumulation) as the fastest mode, followed by tensor-core-fp16 and tensor-core-tf32.

For Matmul, native-tensor-core-fp16 achieves the highest performance at 282.80 TFLOPS, leveraging the tensor cores' full potential for FP16 precision with efficient accumulation and close to its peak performance (see Figure 2.1). This mode's performance is significantly higher compared to the tensor-core-fp16 at 158.16 TFLOPS and tensor-core-tf32 at 81.83 TFLOPS, the latter two also benefiting from tensor cores but limited by their respective accumulation precisions. In contrast, the cuda-core-fp32 mode reaches only 51.18 TFLOPS. According to the roofline graph (see Figure 2.1), CUDA Cores' peak performance matches TF32's, but it seems that CUDA Cores are not as optimized as Tensor Cores for matmul operations, hence the difference, despite having the same theoretical peak performance. So apparently, Tensor Core use may even entice a speedup over CUDA Cores, despite not being straight up more powerful.

Similarly, for convolution operations with Input 1, native-tensor-core-fp16 excels with the shortest execution time of 200.709 ms. The tensor-core-fp16 and tensor-core-tf32 modes follow with longer execution times of 294.417 ms and 239.312 ms, respectively, while the cuda-core-fp32 mode is the slowest at 510.363 ms, reflecting the computational overhead of CUDA cores. More on the similar times for tensor-core-fp16 and tensor-core-tf32 in section 4.4.

4.3 Matmul

Table 4.2: Combined Matmul benchmark for different Compute Modes

Mode	M=N=K	TFLOPS				SpeedUp to cublas-nvcc		
		cublas nvcc	cublas clang	hdnn	pytorch	cublas clang	hdnn	pytorch
native tensor core fp16	512	38.498	54.050	48.806	11.425*	1.403	1.268	0.297*
	4096	292.785	291.145	282.796	145.737*	0.994	0.966	0.498*
	16384	316.837	296.356	288.056	161.316*	0.935	0.909	0.509*
cuda core fp32	512	20.236	22.684	22.370	11.589	1.120	1.105	0.573
	4096	59.823	49.447	51.178	35.355	0.826	0.855	0.591
	16384	64.456	52.622	53.241	34.279	0.816	0.826	0.532
tensor core fp16	512	28.816	34.045	33.554	11.425	1.182	1.164	0.397
	4096	154.288	158.453	158.158	145.737	1.027	1.025	0.945
	16384	158.823	156.514	155.869	161.316	0.985	0.981	1.016
tensor core tf32	512	22.803	26.214	25.565	12.482	1.150	1.121	0.548
	4096	81.894	81.880	81.833	50.457	1.000	0.999	0.616
	16384	83.135	87.216	87.122	56.480	1.049	1.048	0.679

Three different test cases have been chosen to evaluate Tensor Core performance in matmul. M=N=K=512 presents an Arithmetic Intensity (AI) bigger than the ridge point of cuda-core-fp32 and tensor-core-tf32, but smaller than any of the other three modes. This could help visualize the difference between Tensor Core performance and CUDA Core performance, while also displaying what happens when Tensor Cores are used though, the problem's arithmetic intensity is not enough for them to reach peak performance. M=N=K=4096 has a more than sufficient AI for all operations to be compute bounded. Then M=N=K=16384 represents a very high AI matmul, so even in a practical case, close to peak performance is hoped for. In addition, these large GEMMs are quite typical in DL workloads, so a good DL performance assessment in this scenario can be achieved.

As shown in Table 4.2, when using CUDA cores, the difference between HDNN and CuBLAS-clang is almost negligible, trading slightly better performance between the two. Conversely, there seems to be a difference between CuBLAS-nvcc and CuBLAS-clang / HDNN. The table states that HDNN performs even better than the CuBLAS code compiled with nvcc for the smaller dimensions matrix multiplications. In contrast, for larger sizes, particularly for native tensor core and tensor-core-fp16, nvcc comes on top. These differences are, however, comparatively small and do not reflect a large enough amount

to state that HDNN is hampering performance. Instead, they confirm that the compiler used plays a part in achieving performance. With regard to Tensor Core usage, whichever is the datatype benchmarked, HDNN's highest performances relative to CuBLAS keep appearing for M=N=K=512 with overall close performance throughout all compute modes and datatypes thus confirming that HDNN overhead for matrix multiplication is not a concern even in somewhat small matrices.

In contrast, when compared to PyTorch, HDNN performs exceptionally well, especially for sizes smaller than M=N=K=16384 (see Table 4.2). It also allows for the computation in FP16 and the accumulation without these issues, which PyTorch does not, forcing the user to accumulate in FP32 to avoid convergence issues in training. However, the inference could benefit from FP16 accumulation with little to no loss of accuracy [42], so in reality, the end user cannot take full advantage of FP16 performance even if the GPU allows it, hence not being able to achieve the performance of HDNN (see Table 4.2), which provides for computation in FP16 and FP32 accumulation, too.

Regarding this configuration (see Table 4.2), performance will be levelled between the two DSLs, save for smaller dimensions where HDNN comfortably outperforms PyTorch. Overall, for FP16 compute and FP32 accumulation, PyTorch's performance is similar to HDNN. Nevertheless, this situation is reverted when Tensor Cores are used for TF32, with PyTorch underperforming when compared to HDNN again.

In conclusion, HDNN performs with negligible overhead to its vendor library donor, though it can have slightly worse or better performance in some cases depending on the compiler. Regarding its face-off against PyTorch, it displays more flexibility, allowing for FP16 computation and accumulation and better overall performance for this benchmark. However, PyTorch start to catch up for bigger M, N, and K, specially when mixed precision (FP16 computation and FP32 accumulation) is used.

4.4 Convolution

To bench different situations for the convolution, three inputs are selected (see Table 4.3). Inputs 1 and 2 were sourced from the MNIST and Imagenet datasets [44] while Input 3 is based on Nvidia's Convolutional Layers User's Guide's ideas [7]. Input 1 is memory-bounded with a shorter execution time. Input 2 is compute-bounded for native FP16 and memory-bounded for the rest, with a longer execu-

Table 4.3: Convolutional parameters for different inputs

Input	batches	image dim	in channels	out channels	kernel dim
1	100	28x28	20	1	5x5
2	100	227x227	96	3	11x11
3	1	1024x1024	3	512	64x64

Table 4.4: Combined Convolution benchmark for different Compute Modes

Mode	Input	Time (ms)				SpeedUp to cudnn-nvcc		
		cudnn nvcc	cudnn clang	HDNN	PyTorch	cudnn clang	HDNN	PyTorch
native tensor core fp16	1	0.034	0.034	0.054	0.222*	1.003	0.630	0.153*
	2	24.029	23.952	27.150	24.311*	1.003	0.885	0.989*
	3	193.123	191.703	200.709	210.563*	1.007	0.962	0.917*
cuda core fp32	1	0.037	0.0374	0.504	0.2235	0.995	0.074	0.167
	2	16.551	16.503	18.262	76.301	1.003	0.906	0.217
	3	483.614	482.375	510.363	298.335	1.003	0.947	1.621
tensor core fp16	1	0.036	0.036	0.286	0.222	1.000	0.126	0.162
	2	16.557	16.519	18.259	24.311	1.002	0.907	0.681
	3	284.067	282.414	294.417	210.563	1.006	0.965	1.349
tensor core tf32	1	0.036	0.036	0.426	0.223	1.000	0.085	0.161
	2	16.548	16.521	18.259	74.106	1.002	0.906	0.223
	3	284.192	282.497	293.312	204.952	1.006	0.969	1.386

tion time. Input 3 involves a large convolution with high arithmetic intensity to trigger GEMM-based algorithms tailored to tensor core usage. [7].

Regarding Table 4.4, generally, there is no significant difference between the CuDNN nvcc and CuDNN clang compilers, suggesting parity in compilation efficiency, which means if HDNN is slower, it is not due to the compiler selection.

Algorithm selection is a crucial step of the convolution layer execution, as both FP16 and TF32 show similar precision accumulation in CuDNN and HDNN for all three inputs, except for Input 1 with HDNN. In this specific case, the tensor-core-fp16 algorithm selected `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING` while tensor-core-fp32 selected `CUDNN_CONVOLUTION_FWD_ALGO_FFT`. Interestingly, for native-tensor-core-fp16, results for Input 2 appear to be slower than for all other compute modes, including cuda-core-fp32, despite native-tensor-core-fp16 being the only compute mode not math-bounded. This is due to the use of `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` instead of `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING`, which is used in the other modes.

It's important to note that while CuDNN handles algorithm selection, it may not always choose the best algorithm.

As for tensor core acceleration, Inputs 1 and 2 for CuDNN show no speedup as expected for this highly memory-bounded input. At the same time, though PyTorch is not affected, HDNN seems to experiment with a speedup in Input 1 due to the tensor core usage hiding some of the overhead that appears when comparing HDNN to CuDNN. This overhead only appears for Input 1 due to its shorter execution time, making it prominent. Nevertheless, it isn't relevant with longer times in matmul and other convolution layer inputs, as shown in Input 2. Also, HDNN outperforms PyTorch in all modes while achieving the best absolute time except for native, hampered in this ocasion by the previous algorithm selection.

However, Input 3 is math-limited, which may easily translate into actual speedup from tensor core use (see Table 4.4). Though there is an improvement going from CUDA Cores to Tensor and from FP32 to FP16 accumulation, there does not seem to be a difference between TF32 and FP16. In this case, the algorithm for both modes is the same, but the lack of difference could be attributed to CuDNN's algorithm implementation. The reason why PyTorch excels particularly in this arithmetic-intensive Input 3, is potentially due to a different implementation when using a GEMM algorithm, as it even surpass CuDNN's performance with both compilers. While HDNN and the CuDNN program used to run the convolution use Legacy API, PyTorch may utilize the newer Graph API. Still, PyTorch falls short of achieving the best Input 3 performance as it cannot accumulate in FP16 and doesn't outperform HDNN and CuDNN when using tensor cores natively with FP16 data.

4.5 Discussion

This document presents several key contributions that enhance the capabilities and performance of HDNN, particularly in leveraging Tensor Cores for deep learning operations. The main contributions are detailed as follows:

User-Controlled Tensor Core Usage for FP32 Data: A mechanism has been implemented for user-controlled Tensor Core usage in HDNN programs for FP32 data. This is achieved through downconversion to FP16 and TF32, allowing users to select one of three compute modes that optimize for either performance or precision (see Table 4.5).

These adjustable compute modes can be easily set by the user for each program

Compute Mode	Device	Input Data	Compute Data	Output Data	Precision	Performance
cuda-core-fp32	CUDA Cores	FP32	FP32	FP32	Highest	Lowest
tensor-core-tf32	Tensor Cores	FP32	TF32	FP32	Medium	Medium
tensor-core-fp16	Tensor Cores	FP32	FP16	FP32	Lowest	Highest

Table 4.5: Compute Modes, Data Types, and Performance Metrics

without requiring changes to the original program code. This flexibility ensures effective utilization of Tensor Cores based on specific requirements, enhancing both usability and performance. They can be utilized for both convolution and the newly introduced matmul operations alike.

Native Tensor Core Usage for FP16 Operations: Native support for FP16 operations has been added to HDNN by incorporating the FP16 datatype into HDNN's MLIR dialect. This enhancement allows HDNN to fully utilize Tensor Cores for FP16 deep learning operations, which are essential for achieving high throughput and efficiency in modern AI applications. By supporting the FP16 MLIR datatype, HDNN can perform tensor operations with significantly reduced computational overhead, leading to faster and more efficient processing. This usage of Tensor Cores can also be used with both the convolution and newly implemented matmul.

Matrix Multiplication via CUDA CuBLAS Library: A new HDNN operation, Matrix Multiplication, has been introduced, implemented using the CUDA CuBLAS library within HDNN's MLIR dialect. Matrix multiplication is a fundamental operation in many deep learning algorithms, and its efficient implementation is critical for overall performance. By integrating the CuBLAS library, HDNN can leverage highly optimized GPU routines for matrix multiplication, ensuring peak performance and compatibility with existing CUDA-based workflows.

Chapter 5

Conclusions and future work

5.1 Conclusions and Main Contributions

The importance of Domain-Specific Accelerators (DSAs) like GPUs and TPUs has surged due to the performance needs of Deep Learning (DL) applications, necessitating the use of distinct Domain Specific Languages (DSLs) for programming. This complexity brought different device-agnostic solutions, such as HDNN (Heterogeneous Deep Neural Networks), to ensure portability, productivity, and performance. HDNN employs LLVM MLIR's progressive lowering to transition from device-agnostic to device-specific code using vendor-optimized libraries, achieving remarkable performance. HDNN supports CPU, GPU, and TPU through oneDNN, CUDA, cuDNN, and PyTorch. Even so, the rapid development of GPUs has highlighted a limitation in HDNN's lack of extensive utilization of Tensor Cores, which are essential in Nvidia GPUs for deep learning operations. Enhancing HDNN through Tensor Core support for various operations and data types makes exploiting their performance gains in DL tasks possible.

Testing and benchmarking have demonstrated significant improvements in Tensor Core utilization through these contributions. The compute modes for FP32 data achieve performance levels comparable to vendor specifications for convolution and matrix multiplication operations, due to the minimal overhead exhibited by HDNN. The performance was similar to direct implementations using vendor-specific libraries and compilers and, in several scenarios, exceeded that of established frameworks like PyTorch. Furthermore, these results validate that HDNN and its MLIR dialect can easily accommodate new deep learning

operations and datatypes, demonstrating the framework's extensibility and robustness.

In summary, the main contributions of this paper are:

1. Implementation of user-controlled Tensor Core usage for FP32 data through FP16 and TF32 downconversion.
2. Addition of native Tensor Core support for FP16 deep learning operations by incorporating the FP16 MLIR datatype.
3. Introduction of a Matrix Multiplication operation via the CUDA CuBLAS library within HDNN's MLIR dialect.

These contributions collectively enhance HDNN's capability to leverage modern GPU hardware, particularly Tensor Cores, for improved performance in deep learning tasks.

5.2 Future Work

Future enhancements for HDNN include expanding its MLIR dialect to support a broader range of operations and data types, such as BF16 and INT8, to fully exploit modern hardware accelerators. On top of that, updating clang's version to support the later CUDA versions could enable newer optimizations and features. Currently, HDNN relies on clang 13, which supports up to CUDA 11.2, limiting potential performance gains from more recent CUDA releases. Though, as proven, clang 13 is still fairly competitive and offers better performance for some cases, benchmarking a new compiler version that fully supports the latest CUDA could leverage additional performance improvements specially for bigger size matrix multiplications.

Another significant growth path would involve integrating NVIDIA's Graph API instead of the Legacy API. The Graph API enables more efficient management and execution of GPU tasks through the creation of complex execution graphs, optimizing resource usage and minimizing overhead. This shift would improve the efficiency of deep learning workloads, particularly those with multiple dependent operations, by reducing latency, increasing throughput and likely bridging the gap to PyTorch for some use cases.

By addressing these areas, HDNN can continue to evolve as a robust and versatile framework, capable of meeting the growing demands of modern AI and deep learning applications.

Bibliography

[1] Cublas documentation. https://docs.nvidia.com/cuda/cublas/. Accessed: 2024/04/21.

[2] Cublas library. https://developer.nvidia.com/cublas. Accessed: 2024/05/22.

[3] Cuda applications. https://developer.nvidia.com/cuda-zone. Accessed: 2024/05/22.

[4] Cuda deep neural network library (cudnn). https://developer.nvidia.com/cudnn. Accessed: 2024/05/22.

[5] Cuda libraries for gpu-accelerated applications. https://developer.nvidia.com/gpu-accelerated-libraries. Accessed: 2024/05/22.

[6] Cudnn documentation. https://docs.nvidia.com/deeplearning/cudnn/latest/index.html. Accessed: 2024/04/21.

[7] Deep learning performance: Convolutional. https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html. Accessed: 2024/04/21.

[8] Deep learning performance: Matrix multiplication. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html. Accessed: 2024/04/21.

[9] Efficient deep learning: Cvpr 2023 tutorial. https://nvlabs.github.io/EfficientDL/data/presentations/CVPR2023_eff_tutorial_molchanov.pdf. Accessed: 2024/05/20.

[10] Getting started - introduction to sycl - codingame. https://www.codingame.com/playgrounds/48226/introduction-to-sycl/getting-started. Accessed: 2024/05/22.

[11] Learn - sycl.tech. https://sycl.tech/learn/. Accessed: 2024/05/22.

[12] Llvm compiler infrastructure project. https://llvm.org/. Accessed: 2024/04/21.

[13] Mlir. llvm project. https://mlir.llvm.org/. Accessed: 2024/04/21.

[14] Mlir toy tutorial. https://mlir.llvm.org/docs/Tutorials/Toy/. Accessed: 2024/04/21.

[15] Nvidia ada gpu architecture. https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf. Accessed: 2024/05/17.

[16] Nvidia cuda tensorfloat-32 precision format (tf32). https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/. Accessed: 2024/04/21.

[17] Nvidia cuda toolkit documentation. https://docs.nvidia.com/cuda/index.html. Accessed: 2024/05/22.

[18] Sycl overview - the khronos group inc. https://www.khronos.org/sycl/. Accessed: 2024/05/22.

[19] Tensor core performance and precision. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9176-tensor-core-performance-and-precision.pdf. Accessed: 2024/06/03.

[20] Tensor ops made easier in cudnn. https://developer.nvidia.com/blog/tensor-ops-made-easier-in-cudnn/. Accessed: 2024/04/21.

[21] What are convolutional neural networks? https://www.ibm.com/topics/convolutional-neural-networks. Accessed: 2024/05/22.

[22] Dialects - mlir. https://mlir.llvm.org/docs/Dialects/, 2024. Accessed: 2024/06/01.

[23] Graph api — nvidia cudnn v9.1.1 documentation. https://docs.nvidia.com/deeplearning/cudnn/latest/developer/graph-api.html, 2024. Accessed: 2024/06/03.

[24] Hpvm - heterogeneous parallel virtual machine. https://publish.illinois.edu/hpvm-project/, 2024. Accessed: 2024/06/02.

[25] Kokkos abstract. https://kokkos.org/about/abstract/, 2024. Accessed: 2024/06/02.

[26] Kokkos c++ performance portability programming ecosystem: The programming model - parallel execution and memory abstraction. https://github.com/kokkos/kokkos, 2024. Accessed: 2024/06/02.

[27] Legacy api — nvidia cudnn v9.1.1 documentation. https://docs.nvidia.com/deeplearning/cudnn/latest/developer/legacy-api.html, 2024. Accessed: 2024/06/02.

[28] Nsight systems | nvidia developer. https://developer.nvidia.com/nsight-systems, 2024. Accessed: 2024/06/02.

[29] Nvidia ampere architecture in-depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/, 2024. Accesed: 20/05/2024.

[30] Nvidia cuda - docker hub container image library. https://hub.docker.com/r/nvidia/cuda/, 2024. Accessed: 2024/05/27.

[31] Pytorch official website. https://pytorch.org/, 2024. Accessed: 2024/06/02.

[32] Quick guide to sycl implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/quick-guide-to-sycl-implementations.html, 2024. Accessed: 2024/06/02.

[33] What is cuda? https://www.supermicro.com/en/glossary/cuda, 2024. Accessed: 2024/06/01.

[34] What is gpgpu? definition and faqs. https://www.heavy.ai/technical-glossary/gpgpu, 2024. Accessed: 2024/06/03.

[35] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[36] PyTorch Contributors. Pytorch github repository. https://github.com/pytorch/pytorch, 2024. Accessed: 2024/06/02.

[37] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. https://arxiv.org/abs/1603.07285, 2018. Accessed: 2024/06/03.

[38] Adel Ejjeh, Aaron Councilman, Akash Kothari, Maria Kotsifakou, Leon Medvinsky, Abdul Rafae Noor, Hashim Sharif, Yifan Zhao, Sarita Adve, Sasa Misailovic, and Vikram Adve. Hpvm: Hardware-agnostic programming for heterogeneous parallel systems. *IEEE Micro*, 42(5):108–117, 2022.

[39] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. High performance GPU code generation for matrix-matrix multiplication using MLIR: some early results. *CoRR*, abs/2108.13191, 2021.

[40] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. MLIR-based code generation for GPU tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 117–128, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *ArXiv*, abs/2002.11054, 2020.

[42] Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Y Jim Kim, Haihao Shen, and Barukh Ziv. Lower numerical precision deep learning inference and training. *Intel White Paper*, 3(1):1–19, 2018.

[43] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, January 2023.

[44] Pablo Antonio Martínez Sánchez. Improving the performance, portability, and productivity of hardware accelerators. http://hdl.handle.net/10201/132506, 2023. Doctoral Thesis. Accessed: 2024/06/03.

[45] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020.

Appendices

Appendix A

List of acronyms

AI : Artificial Intelligence / Arithmetic Intensity.

API : Application Programming Interface.

CPU : Central Processing Unit.

CuBLAS : CUDA Basic Linear Algebra Subroutines.

CUDA : Compute Unified Device Architecture.

CuDNN : CUDA Deep Neural Network.

DSA : Domain Specific Accelerator.

DSL : Domain Specific Language.

FMA : Fused Multiply-Add.

FP16 : 16-bit Floating Point.

FP32 : 32-bit Floating Point.

FPGA : Field-Programmable Gate Array.

GPGPU : General-Purpose Computing on GPUs.

GPU : Graphics Processing Unit.

HPC : High Performance Computing.

HP : Half Precision.

HDNN : Heterogeneous Deep Neural Networks.

LLVM : Low Level Virtual Machine.

MLIR : Multi-Level Intermediate Representation.

MMA : Matrix-Multiply-and-Accumulate.

MP : Mixed Precision.

P3 : Productivity, Portability, and Performance.

ROCm : Radeon Open Compute.

SM : Streaming Multiprocessor.

SP : Single Precision.

SYCL : System-wide Compute Language.

TF32 : 32-bit Tensor Float.

TPU : Tensor Processing Unit.