
University of Murcia
Faculty of Informatics

# On Using LLMs for Kernel Generation: A Critical Evaluation of Performance, Correctness, and Generality

UNDERGRADUATE THESIS
Degree in Computer Science

By
Daniel Antonio Martínez Sánchez

Advised by
José Manuel García Carrasco
Alexandra Jimborean

Murcia, June 2026
June call for submissions

University of Murcia
Faculty of Informatics

**On Using LLMs for Kernel Generation: A Critical Evaluation of Performance, Correctness, and Generality**

UNDERGRADUATE THESIS

Degree in Computer Science

By
Daniel Antonio Martínez Sánchez

Advised by
José Manuel García Carrasco
Alexandra Jimborean

Murcia, June 2026

June call for submissions

# Declaration of Authorship

I, Daniel Antonio Martínez Sánchez, declare that this undergraduate thesis, titled "*On Using LLMs for Kernel Generation: A Critical Evaluation of Performance, Correctness, and Generality*", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a 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.

Daniel Antonio Martínez Sánchez
Murcia, June 2026

# Abstract

GPU kernels are essential for modern deep learning performance. Yet, kernel code relies on masterful implementations written by experts.This approach is not sustainable considering the continuous development of new deep learning (DL) architectures and the development of new chips and domain-specific languages with different programming models. Moreover, machine learning (ML) compilers fail to handle the ever-changing workloads and diverse hardware.

In this scenario, the only certainty is that writing efficient GPU kernels and fine-tuning them for high performance is difficult. That prompted the Stanford Scaling Intelligence Lab to ask at the end of 2024: “Can LLMs write efficient GPU (CUDA) kernels?” This led to the first version of KernelBench, released in Jan 2025. They showed that frontier LLM models could generate correct kernel code in some cases and, in about 20% of correct cases, faster kernels than PyTorch Eager.

Since then, and throughout 2025 and 2026, the field has expanded with several new projects focused on end-to-end kernel evaluation and the data bottleneck in training LLMs, including BackendBench, FlashInferBench, and TritonBench. And many other projects focused on reward hacking, moving from the first attempts at test-time-search to more expensive evolutionary search methods and reinforcement learning with verifiable rewards (RLVR), pointing to hardware-aware agents to shape the future of kernel generation.

However, LLMs are prone to failure/"hallucination" and lack a fundamental understanding of hardware constraints and the general compilation process. Hence, the generated code is not guaranteed to be correct. This happens for instance with operations like Layer Normalization in which the LLM generates a faster but numerically unstable kernel.

On the other hand, Compilers are robust and reliable tools that have been used for decades. This research aims to identify what techniques LLMs are using

and why compilers are not using them. By analyzing these differences, we seek to improve current compilers by integrating LLM-derived insights into the Machine Learning (ML) compilation flow.

Along the way, there are many aspects to look for: correctness, optimizations and generality. As cited before, KernelBench already provides a good starting point to evaluate kernels generated by LLMs yet a kernel passing the KernelBench evaluation does not mean it is suitable for production. First, we need to make sure the kernel is correct not just numerically for a set of random values (which is what KernelBench does) but also algorithmically as mentioned before.

We also categorize the performance gains from LLMs into two primary sources: Better parameters selection for existing optimizations (e.g: Tile size) or the application of new or different optimization passes. The first source indicates non-optimal fine-tuning on current compilers while the second indicates missing optimization opportunities. Furthermore, we investigate the generality of these optimizations, as compilers often bypass aggressive transformations to maintain stability or limit pipeline complexity.

Ultimately, this work aims to fuse the performance of LLM-generated kernels with the reliability of established ML compilers. By building a bridge between these two paradigms, we envision a future where LLMs can autonomously suggest new, pluggable compiler passes, significantly reducing the manual effort required to evolve compiler high-level structural organization.

# Resumen extendido

En la actualidad, la Inteligencia Artificial (IA) y, más específicamente, las redes neuronales profundas (DNNs), se han consolidado como el área más relevante de la informática, impulsadas por avances en hardware que han viabilizado su transición de la teoría a la práctica. Consecuentemente, la ejecución de estos modelos concentra una cantidad de recursos sin precedentes en la industria, acaparando un altísimo consumo energético, masivas inversiones financieras y el esfuerzo de miles de ingenieros.

La infraestructura física que sustenta el cómputo de estas cargas de trabajo se basa en los denominados aceleradores de hardware, entre los que destacan las Unidades de Procesamiento Gráfico (GPUs). Aunque estos dispositivos fueron concebidos originalmente para el renderizado gráfico, su arquitectura masivamente paralela ha provocado una revolución en su mercado, reorientándose de manera progresiva hacia la ejecución de cargas de IA.

Sin embargo, este hardware carece de utilidad sin un ecosistema de software capaz de explotar su potencial. A la par del desarrollo de nuevos aceleradores, han surgido plataformas de software específicas para darles soporte, siendo CUDA, el lenguaje de programación de NVIDIA, el ejemplo más prominente. No obstante, el desarrollo sobre estos dispositivos difiere sustancialmente de la programación tradicional en CPU. Resulta considerablemente más complejo debido a que exige operar a un nivel de abstracción mucho más bajo, requiriendo por parte del programador un conocimiento profundo de las particularidades de la arquitectura subyacente.

En el contexto actual, la finalidad primordial del código desarrollado para aceleradores es dar soporte a las cargas de trabajo de IA. Para gestionar esta complejidad, han surgido los compiladores de machine learning (orientados a redes neuronales). Estas herramientas toman como entrada un modelo de IA y lo transforman en código ejecutable optimizado mediante el proceso de

compilación, abstrayendo al programador de la complejidad subyacente a nivel hardware. Internamente, estos compiladores se apoyan por lo general en librerías basadas en CUDA (como cuDNN o cuBLAS ) o generan código en lenguajes de más alto nivel como Triton, dependiendo de lo que resulte más conveniente.

En este punto, emerge un compromiso fundamental: a menor nivel de abstracción, mayor es el rendimiento potencial que obtenemos. Por norma general, un código minuciosamente escrito en CUDA superará en eficiencia a uno equivalente en Triton o en otros lenguajes de más alto nivel. Esto se debe a que los lenguajes de bajo nivel exponen en mayor medida las singularidades del hardware, como el manejo de las memorias dedicadas. Estos elementos son ocultados por los lenguajes de mayor nivel para proporcionar un uso más sencillo, lo que a su vez conlleva una pérdida implícita de rendimiento. Sin embargo, incluso lenguajes de más alto nivel como Triton poseen una complejidad considerable. En consecuencia, se crea una fuerte dependencia hacia piezas de software sumamente complejas de escribir, cuyo desarrollo recae de forma exclusiva en un número reducido de expertos.

Para ilustrar la magnitud de este desafío, resulta útil imaginar las redes neuronales como gigantescos edificios construidos a partir de bloques básicos. En el ámbito del hardware y la programación de bajo nivel, estos bloques se denominan *kernels*. Un *kernel* no es más que un pequeño pero crítico fragmento de código, diseñado para ejecutar una operación matemática muy específica (como, por ejemplo, multiplicar dos matrices) de la manera más rápida y paralela posible en la GPU. Si un solo *kernel* sufre de una optimización deficiente, todo el "edificio" de la IA se ralentiza, consumiendo una mayor cantidad de tiempo y energía. Históricamente, programar y exprimir al máximo el rendimiento de estos *kernels* ha sido una labor puramente artesanal, reservada para una élite de ingenieros altamente especializados.

Paralelamente a estos retos de infraestructura, los *Large Language Models* (LLMs), como ChatGPT o Claude, han experimentado un avance significativo y se han integrado plenamente en los flujos de desarrollo de software gracias a su notable capacidad para la generación de código. Ante este panorama, a finales de 2024, investigadores de la Universidad de Stanford plantearon la viabilidad de utilizar LLMs para automatizar la creación de este código de bajo nivel altamente complejo, introduciendo un sistema pionero para su evaluación denominado KernelBench. Desde entonces, se ha producido una rápida proliferación de trabajos de investigación que exploran multitud de enfoques alternativos, incluyendo sistemas multi-agente o aprendizaje por refuerzo, para abordar la generación y evaluación de estos *kernels*.

El software resultante de estos modelos se escribe en lenguajes de programación específicos para GPU (principalmente CUDA o Triton) y está destinado a computar operadores matemáticos fundamentales del machine learning. Entre ellos se incluyen operaciones aisladas como multiplicaciones de matrices, convoluciones o capas de normalización, así como secuencias más complejas que fusionan múltiples operadores dentro de un único *kernel*.

Actualmente, estos estudios reportan un muy alto porcentaje de código generado por LLMs que es correcto es decir, código que en primera instancia compila y que genera los resultados esperados. Lo más sorprendente es que de esos códigos algunos reportan una mejora en rendimiento comparados con el estado del arte esto es, comparado con códigos generados por el compilador o por códigos de librerías altamente optimizadas por grandes expertos.

Actualmente, la literatura científica asociada reporta que un alto porcentaje del código generado por estos LLMs es funcionalmente correcto; es decir, compila con éxito y produce los resultados numéricos esperados en los entornos de prueba. Aún más llamativo es que, en una fracción de estos casos válidos, los estudios documentan mejoras de rendimiento frente al estado del arte. Esto implica que el código generado por inteligencia artificial aparentemente logra superar en velocidad tanto a la salida de los compiladores estándar (como Triton) como a las implementaciones de librerías cerradas (como cuDNN), las cuales han sido optimizadas a mano por ingenieros expertos.

Motivada por estas afirmaciones, el presente trabajo tiene como propósito principal analizar críticamente los *kernels* generados por LLMs, evaluando de forma conjunta su rendimiento bruto, su corrección estructural y su capacidad de generalización. El objetivo es diseccionar cómo el modelo de lenguaje logra estas mejoras de velocidad y determinar si el código resultante es verdaderamente viable para un entorno de producción. En la industria, un *kernel* no solo debe funcionar para un caso aislado, sino que debe garantizar resultados estables, numéricamente robustos y correctos ante multitud de configuraciones (tamaños de tensores) e incluso a través de diferentes microarquitecturas de hardware. En consecuencia, este trabajo trasciende la mera medición empírica de tiempos de ejecución para adentrarse en una auditoría manual exhaustiva a nivel de código.

Para garantizar la rigurosidad y homogeneidad de las mediciones, la metodología experimental se ejecutó íntegramente sobre una GPU NVIDIA RTX 4090. Como línea base de comparación (estado del arte), se tomó el código generado dinámicamente por el compilador de redes neuronales Triton (ampliamente integrado en PyTorch), así como las implementaciones provenientes de

librerías propietarias altamente optimizadas por NVIDIA, tales como cuDNN y cuBLAS.

Cabe destacar que la evaluación contenida en este estudio, en consonancia con la mayor parte de la literatura reciente, se centra exclusivamente en la *forward pass* relacionada con el proceso de inferencia de la red neuronal, omitiendo el proceso de retropropagación o entrenamiento (*backward pass*). Esta decisión metodológica se fundamenta en que la inferencia presenta exigencias de latencia mucho más estrictas en entornos de despliegue reales. Además, en términos globales, representa la etapa que mayor volumen de recursos computacionales consume, dado que, una vez finalizado su entrenamiento, el modelo opera de manera constante y masiva atendiendo peticiones.

Para este estudio, se ha analizado un total de 10 *kernels* extraídos de 6 trabajos de investigación recientes. Esta selección abarca una diversidad técnica sustancial, contemplando tanto operadores limitados por cómputo (*compute-bound*) como limitados por el ancho de banda de memoria (*memory-bound*). Asimismo, se han evaluado operaciones unitarias y secuencias algorítmicas más complejas que fusionan múltiples operadores en un solo código. Entre las piezas analizadas se encuentran componentes fundamentales que constituyen el núcleo estructural de cualquier arquitectura moderna de IA. Por ejemplo, se auditaron *kernels* de convolución (esenciales en tareas de visión artificial), multiplicaciones generales de matrices o GEMM (el motor computacional subyacente en el procesamiento de lenguaje natural) y funciones de normalización como Layer Normalization o RMSNorm (mecanismos críticos para mantener la estabilidad numérica de la red durante su ejecución).

La evaluación de un abanico tan heterogéneo de cargas de trabajo permitió verificar no solo la viabilidad sintáctica del código generado por los modelos de lenguaje (es decir, su capacidad para compilar con éxito), sino, de manera más crítica, evaluar si la IA posee una comprensión profunda de las restricciones físicas del hardware al enfrentarse a problemas con cuellos de botella radicalmente distintos.

Para garantizar el rigor analítico, el código generado por el LLM se contrastó exhaustivamente frente a las implementaciones producidas por los compiladores de referencia del estado del arte (como Triton). Esta comparativa representa un desafío técnico de gran magnitud, dado que exige auditar implementaciones de alta complejidad a muy bajo nivel. El proceso no se limitó a la inspección superficial del código fuente en lenguajes como CUDA o Triton, sino que requirió descender hasta la Representación Intermedia (IR) del compilador y el código ensamblador (PTX). A medida que el compilador aplica sus pasadas de opti-

mización, el código desciende en su nivel de abstracción acercándose al código máquina, transformándose en una representación mucho más extensa y compleja de analizar. Por consiguiente, descifrar las diferencias estructurales subyacentes entre estas implementaciones requirió una sólida base de conocimientos previos en microarquitectura y numerosas horas de auditoría manual.

También resulta fundamental comprender el espectro de optimizaciones aplicables a este tipo de código de bajo nivel, ya que es un requisito indispensable para evaluar si el modelo de lenguaje las implementa de manera correcta o si, por el contrario, omite transformaciones cruciales. Para estructurar este análisis analítico, el presente trabajo categoriza las optimizaciones en tres grandes grupos: de memoria, de cómputo y algorítmicas.

El primer grupo, las optimizaciones de memoria, adquiere una relevancia superlativa en este ámbito. Las cargas de trabajo de IA suelen estar fuertemente limitadas por el ancho de banda (*memory-bound*), lo que provoca que las unidades de ejecución de la GPU permanezcan ociosas a la espera de que lleguen los datos desde la memoria global. El propósito de estas optimizaciones es mitigar este cuello de botella mediante la explotación de jerarquías de memoria más rápidas, de menor latencia y cercanas al procesador (como la memoria compartida y el banco de registros). Asimismo, buscan ocultar la latencia de lectura solapando el movimiento de datos con el cómputo activo, empleando técnicas avanzadas como el *tiling* (bloques de memoria) o el *software pipelining* mediante copias de memoria asíncronas.

Por su parte, las optimizaciones de cómputo persiguen maximizar la utilización del hardware subyacente, garantizando que las unidades funcionales operen a su máxima capacidad teórica y reduciendo al mínimo la sobrecarga de instrucciones y los tiempos de bloqueo. Un imperativo en las arquitecturas modernas de NVIDIA es la correcta explotación de los Tensor Cores, unidades altamente especializadas en operaciones de multiplicación y acumulación de matrices (MMA) en precisión mixta. El uso eficiente de estos componentes, junto con otras técnicas como el desenrollado de bucles o la comunicación directa entre hilos mediante primitivas a nivel de warp, es lo que permite escalar el rendimiento hasta los límites del hardware.

Finalmente, las optimizaciones algorítmicas se centran en reestructurar la lógica matemática subyacente del problema. El objetivo es reducir su complejidad algorítmica o alinear intrínsecamente el algoritmo con el modelo de ejecución masivamente paralelo del acelerador. A diferencia de las técnicas de memoria y cómputo, estas transformaciones semánticas suelen quedar fuera del alcance de los compiladores de machine learning tradicionales, ya que estos operan bajo

restricciones estrictas de equivalencia matemática y no pueden alterar la fórmula original planteada por el programador.

Partiendo de este marco de análisis, la auditoría de los *kernels* generados por LLMs revela múltiples matices. En primer lugar, se detectó que los modelos de lenguaje recurren con frecuencia a atajos algorítmicos que, si bien logran superar los mecanismos de validación sintáctica y numérica en pruebas aisladas, resultan categóricamente inviables para entornos de producción. Un caso paradigmático documentado en este estudio es el operador de *Layer Normalization* aquí, el LLM descarta algoritmos robustos (como el algoritmo de Welford utilizado por Triton) en favor de una reducción estadística más ingenua y veloz, pero inestable a nivel numérico. De manera análoga, en el cálculo de convoluciones en precisión de 32 bits (FP32), el LLM fuerza internamente una reducción del tipo de dato a 16 bits (FP16) para obligar al hardware a utilizar los Tensor Cores. Aunque esto acelera el cómputo, introduce una pérdida de precisión silenciosa y no deseada por el usuario.

A pesar de estas deficiencias, existen escenarios concretos donde los modelos logran identificar estrategias de optimización notables. Por ejemplo, al evaluar una secuencia de operadores compuesta por una Convolución 2D seguida de una función de activación ReLU y un MaxPool, el código generado implementó una fusión de operadores sumamente agresiva apoyada en operaciones atómicas, logrando superar el rendimiento de las alternativas del estado del arte. Sin embargo, este aparente éxito se desvanece al evaluar la generalidad del código. Cuando se modifica mínimamente el tamaño del tensor de entrada respecto a la configuración exacta que se le proporcionó al LLM durante su generación, el rendimiento del *kernel* sufre una degradación catastrófica debido a la masiva contención de memoria generada por dichas operaciones atómicas.

En la ingeniería de software, un principio fundamental es la generalidad y reutilización del código; un algoritmo diseñado para procesar una matriz de $224x224$ píxeles debería escalar correctamente ante variaciones dimensionales. Sin embargo, este estudio demuestra que los LLMs padecen de un grave problema de "sobreajuste" (overfitting) a los parámetros de entrada. Durante la generación, los modelos incrustan valores rígidos y configuraciones estáticas (los llamados "números mágicos", como el dimensionamiento fijo de bloques o la asunción de alineaciones de memoria perfectas) que se ajustan exclusivamente a las dimensiones específicas proporcionadas en el prompt. Esto genera un espejismo de superioridad frente a los expertos humanos en ese caso aislado. No obstante, si en un entorno de producción la dimensión de los tensores varía mínimamente, el rendimiento del código generado se desploma drásticamente, o bien desencadena

contenciones de memoria insalvables. Compiladores modernos como Triton descartan sistemáticamente estas estrategias frágiles para priorizar y garantizar la robustez del software bajo cualquier circunstancia.

A pesar de estas limitaciones, también se han documentado instancias donde el LLM introduce optimizaciones algorítmicas legítimas que escapan a las capacidades de un compilador tradicional. Por ejemplo, en secuencias complejas de operadores (como la combinación de GEMM, suma, división y escalado), el LLM demostró ser capaz de identificar factorizaciones algebraicas que reducen la complejidad de la operación. En estos escenarios, el modelo no actúa como un experto en microarquitectura sorteando limitaciones físicas del hardware, sino más bien como un revisor algebraico automatizado, descubriendo una formulación matemática mucho más simple y eficiente para alcanzar el mismo resultado.

Como conclusión, los resultados indican que los LLMs aún no poseen la capacidad de superar de manera autónoma, genérica y fiable el estado del arte en la generación de *kernels* para GPU, los compiladores consolidados continúan siendo la herramienta más robusta para entornos de producción. Sin embargo, el potencial demostrado en áreas específicas sugiere que el futuro pasa por la adopción de un enfoque híbrido. A la luz de los hallazgos de este trabajo, se propone integrar los modelos de lenguaje en las etapas tempranas de la compilación de grafos computacionales. En esta fase, el LLM puede desempeñar labores de simplificación matemática a alto nivel antes de delegar el proceso de traducción a código de bajo nivel al compilador tradicional. Adicionalmente, el uso de LLMs resulta viable para generar rutas rápidas de ejecución dirigidas exclusivamente a aquellas piezas estáticas de los modelos de IA que operan sistemáticamente con tensores de tamaño fijo e inmutable.

# Acknowledgments

Han sido cuatro años de carrera, en los que he cambiado mucho como persona y como informático, a lo largo de estos años han sido muchos los que me han acompañado en mi camino y quiero darles las gracias de corazón.

A mis padres, que me han permitido el lujo de estudiar la carrera, que han hecho posible que me dedique completamente al estudio, que siempre quieren lo mejor para mi y que aguantan todas mis locuras, gracias.

A mi hermano Pablo, pues quien tiene un hermano tiene un tesoro. Él compartió su pasión por la informática conmigo haciendo que ahora esté dónde estoy. Siempre serás un referente para mi.

A mis tutores, José Manuel, que confió en mí desde primer curso y siempre ha sabido cómo empujarme para ir mejorando no solo en lo profesional sino también en lo personal, y Alexandra, que también me ha motivado y conducido durante este curso, siempre dándome nuevas ideas y oportunidades. Espero que podamos seguir trabajando juntos.

A mis amigos de siempre, los que han visto mis más y mis menos estos cuatro años, Walid, Gonzalo, Sergio y Camilo que han estado cuando los he necesitado y me han querido por como soy.

A mis compañeros José Ángel, Jesús, Marco, María, Domingo, Vicente, Nacho, Diego, Alexandra y muchos más. Este año ha sido el mejor de los cuatro gracias a vosotros y el buen ambiente en clase, los descansos, las comidas, los paseos por el campus, los viajes en tranvía..., sin duda lo echaré mucho de menos.

A mis amigos de Hakuna y Effetá, sois tantos que no puedo nombraros a todos, gracias por vuestra acogida y por poder desconectar con vosotros todas las semanas, gracias por vuestro apoyo, ayuda y amor.

Finalmente, gracias a Dios, que hace que todo esto sea posible.

# Contents

# List of Figures

# List of Tables

Chapter 1

# Motivation & Introduction

In the past few years, Large Language Models (LLMs) have taken over the world of computer science and software engineering [10]. Today, almost everyone in the industry relies on these models to assist with or directly execute their daily tasks. This is especially true in software development, where models perform exceptionally well at generating code in high-level languages like Java, Python, or C++. We are transitioning through a historical paradigm shift where code is no longer exclusively written by humans; it is increasingly being partially or even fully authored by artificial intelligence (AI).

Simultaneously, a massive race for the most capable AI model is underway between major tech companies like OpenAI, Google, and Anthropic. These organizations are investing unprecedented amounts of effort and capital into optimizing these models in every possible way. Unsurprisingly, these models represent the most significant computational cost of our time, driving the construction of massive data centers designed exclusively to support these incredibly demanding workloads.

This creates a fascinating paradox: while we rely on AI to write software, the software required to execute the AI itself remains a highly manual, human-driven bottleneck. LLMs are not yet ready to fully replace many aspects of the software engineering world, particularly the foundational aspects related to AI infrastructure. Improving the efficiency of how these models are deployed is the key to making AI sustainable, and it begs a critical question: can we use LLMs to optimize the very code that makes LLMs run?

## 1.1 Problem Context

When researchers and engineers develop AI models, they typically use high-level application frameworks like PyTorch [22] and TensorFlow [23]. These libraries provide a highly abstracted, mathematical view of the problem, allowing developers to focus on architecture rather than hardware. However, the underlying code that actually executes these AI models is exponentially more complex than a few lines of Python.

Every single operator present in these libraries has been meticulously optimized by expert engineers with years of experience so that it can run as fast as possible on the target hardware. These highly optimized pieces of code are called kernels. Kernels are the silent backbone of modern AI, turning massive neural networks into a rapid series of mathematical computations (primarily matrix multiplications). A small improvement in a kernel can mean a big improvement over the whole AI model [31].

The complexity of this task is magnified by the fact that modern AI models are rarely run on standard CPUs. Instead, they rely on Domain-Specific Architectures (DSAs) usually known as accelerators such as NVIDIA or AMD GPUs, Google TPUs, or custom NPUs [19]. Writing efficient kernels requires a vast, intimate knowledge of how this specific DSA hardware works, including memory hierarchies, warp synchronization, and register allocation. Furthermore, it requires mastery of Domain-Specific Languages (DSLs) like CUDA or Triton, which expose these hardware complexities to the programmer, often paired with highly complex compiler environments.

Because of this, writing high-performance kernels remains one of the most difficult and specialized tasks in software engineering [1] [4]. As new AI architectures emerge and hardware rapidly evolves, relying on a small pool of human experts to manually craft and update these kernels is becoming an unsustainable bottleneck.

## 1.2 Goals and Scope

Given the need for faster kernels and the emergence of capable coding LLMs, a series of recent works have started exploring the idea of using LLMs to autonomously generate GPU kernels. The first work to propose this idea was KernelBench [20] in early 2025. In that work different LLMs are tested for generating GPU kernels achieving a variable rate of correctness and performance.

Some of those LLM generated kernels managed to outperform State-of-the-Art (SoTA) kernels. This caused a revolution in the field and many other works have started trying to improve these metrics [30].

The primary goal of this thesis is to critically evaluate the true potential of LLMs writing kernel code. More importantly, we seek to understand how LLMs achieve performance improvements over masterfully crafted human code or industry-standard machine learning compilers. By analyzing these generated kernels, we aim to identify the specific optimization techniques the LLMs employ, determine why these techniques might be missing from current SoTA compilers, and explore whether LLM-derived insights could be systematically integrated into traditional compilation flows.

To bound this research, the scope of this work is limited to Level 1 and Level 2 kernels rather than full end-to-end machine learning architectures. We benchmark these LLM-generated kernels against highly optimized vendor libraries and compiler-generated code (e.g., Triton/cuDNN). All evaluations are executed on an NVIDIA RTX 4090 GPU to provide a consistent hardware baseline.

Crucially, this research does not just measure raw execution speed. A significant portion of this work involves manually auditing the LLM-generated code to identify algorithmic compromises, "hallucinations," or incorrect optimizations (such as bypassing numerical stability for speed) that would render the kernel unsafe for real-world production.

## 1.3 Structure of the Document

The rest of the document is as follows: Chapter 2 reviews the state of the art, covering GPU architecture and programming, LLM generated kernels and their evaluation, and an introduction to the world of machine learning compilers. Chapter 3 describes the kinds of optimizations we can expect in GPU kernels, their goals and an explanation of each one. Chapter 4 presents the performance evaluation of 10 studied kernels explaining what they do right and wrong (and why) and what we can learn from each one of them. Finally, Chapter 5 recaps the previous contributions and proposes directions for future research.

Chapter 2

# Background & State of the Art

## 2.1 GPU Programming Paradigms

The exponential growth of DL has been fundamentally driven by the parallel compute capabilities of GPUs. However, fully utilizing a GPU's theoretical throughput requires specialized programming models that can efficiently map complex neural network operators onto underlying hardware architectures. The journey to the modern ML compiler stack is a story of progressively abstracting hardware complexity.

In their early iterations throughout the 1980s and 1990s, GPUs were highly specialized, fixed-function hardware pipelines designed exclusively for rendering graphics. As the hardware evolved to include programmable pixel and vertex shaders in the early 2000s, researchers realized that these massively parallel processors could be repurposed for general-purpose computing (GPGPU).

However, before dedicated compute architectures existed, programmers had to rely entirely on graphics APIs like OpenGL or DirectX to perform mathematical calculations [13] [21]. This meant that algorithms had to be painstakingly disguised as graphics rendering tasks: input data was loaded as textures, compute kernels were written as pixel shaders, and the final mathematical output was "rendered" into a framebuffer rather than a standard memory array. This process was notoriously tedious and required a profound understanding of the graphics rendering pipeline. The landscape shifted dramatically in late 2006 when NVIDIA introduced the Compute Unified Device Architecture (CUDA), which was officially released to the public in early 2007 [9].

CUDA revolutionized parallel computing by completely stripping away the requirement to map computations to graphics primitives. Instead, it provided a C-based programming interface that allowed software to interact directly with the GPU's parallel cores. By exposing the underlying hardware memory hierarchy and thread execution model, CUDA transformed the GPU into a highly versatile co-processor. This hardware-software synergy eventually became the critical infrastructure that enabled the deep learning revolution triggered by models like AlexNet in 2012 [12].

While CUDA provides maximum performance and granular hardware control, it is exceedingly complex. Writing efficient CUDA code requires deep expertise in low-level memory management, warp synchronization, and register allocation. As machine learning architectures exploded in size and diversity, writing bespoke, optimized CUDA C++ kernels for every new mathematical operator became an unsustainable bottleneck.

To bridge this gap, the industry witnessed a rapid proliferation of Domain-Specific Languages (DSLs) tailored specifically for deep learning workloads. Frameworks and languages like TVM, and eventually OpenAI's Triton [25] were developed to provide higher-level abstractions. These DSLs allow developers to express tensor computations using familiar semantics without needing to manually manage the lowest-level GPU hardware details. By combining the expressive power of languages like Python with powerful compiler backends, these modern DSLs automate complex optimizations, democratizing high-performance GPU programming and laying the foundation for today's state-of-the-art ML compiler stacks.

### 2.1.1 The Abstraction Tax: Complexity vs. Performance

The evolution from raw graphics APIs to CUDA, and subsequently to modern DSLs, highlights a fundamental trade-off in high-performance computing: the inverse relationship between language abstraction and the theoretical performance ceiling. As programming models become more abstract and easier to use, they inherently impose an "abstraction tax" on performance [26].

Languages that operate closer to the bare metal expose the granular details of the hardware architecture. By forcing the developer to manually orchestrate register allocation, shared memory bank alignments, and warp-level synchronizations, these languages offer the highest possible performance ceiling. However, achieving this requires immense engineering effort, and the resulting code is often highly brittle and completely tied to a specific hardware generation.

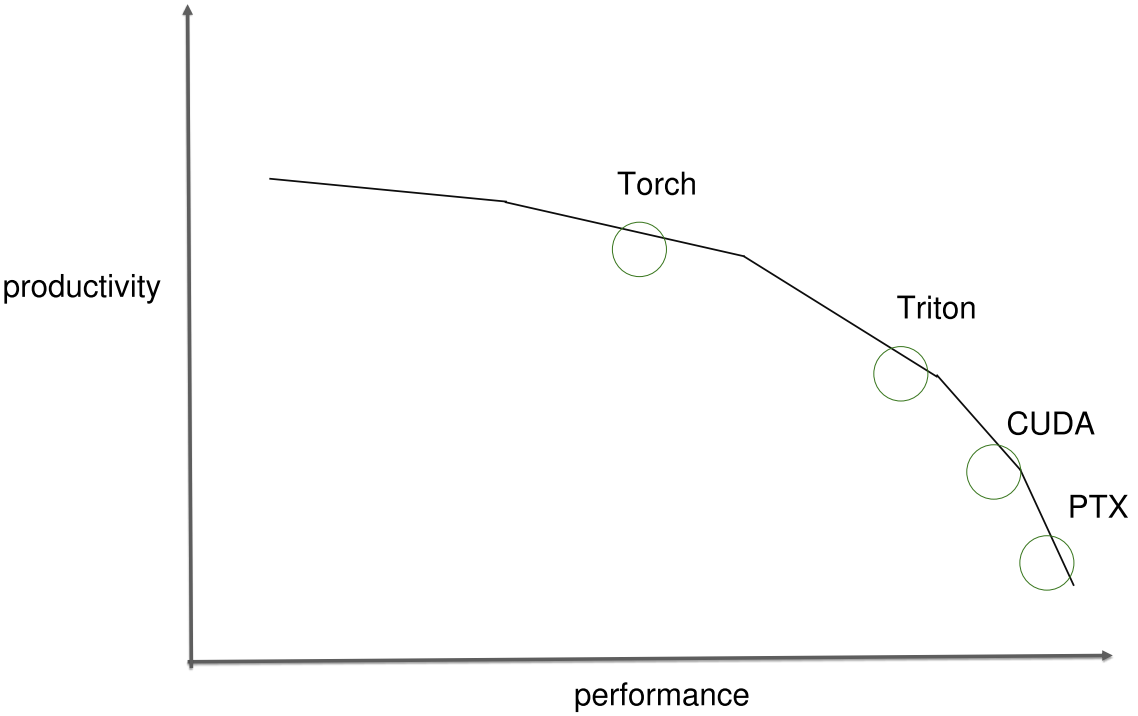


Figure 2.1: Productivity and performance trade-off based on the DSL [24]

DSLs and framework-level abstractions prioritize developer productivity, memory safety, and hardware portability. They rely on ML compilers to automatically translate high-level tensor operations into executable code. To ensure stability and correctness across diverse workloads, compilers often employ conservative optimization heuristics. They intentionally bypass aggressive, edge-case transformations to avoid pipeline complexity or algorithmic instability.

## 2.2 LLM Generated Kernels & Evaluation

LLMs have quickly integrated in the world of software programming, as they are useful and powerful assistants capable of writing code. Their adoption has become mostly universal in the software world however, not all pieces of software are equally complex. One of the most complex software tasks is writing efficient code for Deep Neural Networks (DNNs) operators (called kernels), these tasks becomes even harder if targeting hardware accelerators like GPUs.

In this context some works started exploring the idea of LLM generated kernels the first being KernelBench [20] which proposes a framework and methodol-

ogy to generate and evaluate LLM generated kernels. In that work they divided the kernels into three levels:

- Level 1: Single primitive operations
- Level 2: Operator sequences
- Level 3: Full Machine Learning architectures.

Building upon the foundations of KernelBench, subsequent works [18][3][16][2] have expanded the scope of LLM kernel generation by exploring new languages, improving training mechanisms, and bridging the gap to production deployment.

While KernelBench heavily evaluates raw CUDA generation, LLMs often struggle with the unique syntax and semantics of Domain-Specific Languages (DSLs). To address this, TritonBench [16] was introduced as a comprehensive evaluation framework specifically tailored to Triton operator generation. TritonBench consists of two complementary channels: TritonBench-G, which contains 184 carefully curated real-world operators sourced from GitHub, and TritonBench-T, which tests operator generation aligned with PyTorch interfaces. Unlike traditional benchmarks that only test for functional correctness, TritonBench emphasizes hardware efficiency by profiling execution performance directly on NVIDIA GPUs. Studies on this benchmark revealed that while current code LLMs struggle with efficient Triton generation out-of-the-box, domain-specific supervised fine-tuning (SFT) can considerably improve their execution accuracy.

Moving beyond mere evaluation, CUDA-L1 [18] aims to actively improve an LLMs capacity for CUDA optimization by employing a novel contrastive Reinforcement Learning (RL) framework. Since conventional LLMs lack sufficient exposure to CUDA code during pre-training, CUDA-L1 utilizes a three-stage training pipeline: Supervised Fine-Tuning (SFT) via data augmentation, self-supervised learning, and finally, contrastive reinforcement learning.

By providing models with previous CUDA implementations and their execution scores, the contrastive RL approach forces the LLM to compare effective and ineffective optimization strategies before generating new code. As a result, CUDA-L1 can automatically discover and combine advanced optimization techniques, such as memory coalescing, operation fusion, and hardware-specific memory layout optimizations. Importantly, the developers of CUDA-L1 also identified and mitigated severe reward hacking behaviors typical in RL, where

models would artificially manipulate timing measurements (e.g., via hidden asynchronous streams) or use lazy evaluation to bypass correctness checks.

Furthermore, FlashInferBench [29] implements a dynamic substitution mechanism (`apply()`) that allows the best-performing, validated kernels to be injected seamlessly into production LLM engines with zero code intrusions. Through its evaluation, FlashInferBench again highlighted a crucial language trade-off for AI agents: high-level DSLs like Triton yield significantly higher correctness rates because they require less long-context reasoning and rely on the compiler for hardware mappings. However, low-level languages like CUDA offer a substantially higher performance ceiling by allowing explicit control over fine-grained optimizations like shared memory management, pointing to where future agent training must improve.

In addition to using only a single LLM to generate the kernel, other works use a multi-agent [27] [14] approach meaning that several LLM agents interact with each other in a pipeline to generate the kernels. For example, one agent can be in charge of profiling. Next, that agent will interact with the coding agent in charge of optimizing the kernel and then there can be a third agent in charge of debugging at the end of the pipeline.

## 2.3 Machine Learning Compilers

For decades, traditional compilers have served as the fundamental bridge between human-readable source code and machine-executable instructions. Toolchains like the GNU Compiler Collection (GCC) established the early standards for translating high-level programming languages into optimized assembly. However, the landscape shifted significantly with the introduction of the LLVM compiler infrastructure. By introducing a modular architecture centered around a strict, well-defined Intermediate Representation (IR), LLVM separated language-specific frontends from hardware-specific backends. This decoupling revolutionized compiler design, allowing developers to build new languages or target new hardware architectures without having to reinvent the entire compilation pipeline from scratch.

As Deep Learning (DL) architectures grew in size and complexity, it became evident that traditional scalar-centric compilers were ill-equipped to handle the massive, highly parallel tensor operations required by neural networks. To address this, a specialized generation of Machine Learning compilers emerged. Frameworks like Apache TVM [5] provided some of the first end-to-end com-

pilation stacks capable of optimizing graph-level tensor computations across diverse hardware backends. However, as different frameworks and hardware vendors began creating their own bespoke intermediate representations, the ML ecosystem faced severe fragmentation. To solve this, MLIR (Multi-Level Intermediate Representation) [15] was developed. Rather than acting as a single, rigid IR, MLIR is a modular compiler infrastructure that allows developers to define custom "dialects" for different levels of abstraction. This enables a unified, reusable pathway to progressively lower complex, high-level ML graphs down to hardware-specific instructions.

Today, MLIR serves as the foundational backbone for some of the most critical compilers in the deep learning ecosystem. Google's XLA (Accelerated Linear Algebra) heavily utilizes MLIR dialects to optimize linear algebra computations for TPUs and GPUs. Similarly, IREE (Intermediate Representation Execution Environment) leverages MLIR to provide a streamlined, end-to-end execution framework tailored for both datacenter and edge deployments. Even higher-level Domain-Specific Languages rely on this infrastructure; Triton [25], for example, utilizes MLIR under the hood to lower its Python-centric operations into a specialized Triton-IR dialect before ultimately compiling it down to LLVM IR and NVIDIA's PTX.

To fully appreciate the impact of these modern compiler stacks, it is necessary to contrast them with the traditional execution paradigm of deep learning frameworks. Historically, Tensorflow and PyTorch fought for the spot of being the main DL language. Ultimately, PyTorch emerged as the de facto standard, although TensorFlow offered robust Ahead-of-Time (AoT) compilation and static graph execution early on, PyTorch won out through its highly Pythonic, developer-centric design. At first, PyTorch operated almost exclusively in *Eager mode*. Eager mode employs an imperative execution style where operations are evaluated immediately as they are encountered in the Python interpreter. While this provides an incredibly intuitive developer experience and simplifies debugging, it introduces significant performance overhead. Because the execution is bound to the Python runtime, the framework cannot look ahead at the broader computational graph. Consequently, it executes operations sequentially as isolated, heavyweight kernel launches, missing critical opportunities for global optimizations such as kernel fusion or memory layout transformations.

In contrast, `torch.compile` shifts the paradigm from eager execution to graph-based compilation. Instead of executing operations one by one, `torch.compile` uses a component called TorchDynamo to safely intercept Python bytecode right before execution and trace the underlying computational

graph. This captured graph is then passed to a compiler backend, TorchInductor, which leverages the MLIR and Triton ecosystems. By analyzing the entire graph holistically, the compiler can fuse multiple element-wise operations (e.g., combining an activation function with a matrix multiplication) into a single, highly optimized GPU kernel. This drastically reduces memory bandwidth bottlenecks and CPU GPU synchronization overhead, transforming PyTorch from a purely imperative framework into a hybrid system that retains developer flexibility while achieving static-graph compilation performance.

Despite the unifying technical architecture that MLIR provides, the broader machine learning compiler ecosystem remains highly divided, with different vendors and research labs continuously pushing competing standards and toolchains. Nevertheless, Triton has firmly established itself as the most popular and widely adopted choice among developers. This breakout success is largely driven by its seamless integration with PyTorch, specifically acting as the default backend for `torch.compile`, which successfully bridges the gap between the granular, hardware-aware performance of an ML compiler and the accessible, high-level abstractions of Python.

Chapter 3

# Classification of Optimizations

Throughout this work, we search for code optimizations that increase kernels' performance. There are numerous optimizations in this context, so we propose to categorize them into: memory-focused, compute-focused, and algorithmic.

## 3.1 Memory optimizations

With memory optimizations, our goal is to reduce the latency of memory access as much as possible. This translates to searching for ways to use the closest memory level possible during our computation bringing memory to registers, shared memory or cache. It can also mean hiding such latencies using asynchronous memory copies. This has become a more relevant optimization with the introduction of the `cp.async` instruction in the Ada Lovelace architecture and the posterior addition of the Tensor Memory Accelerator (TMA) introduced in the Hopper architecture.

### 3.1.1 Tiling

Tiling (also known as blocking) is a fundamental optimization technique designed to maximize data locality and increase the arithmetic intensity of a kernel. Tiling mitigates the memory wall by decomposing large, memory-bound operations such as matrix multiplications or implicit GEMMs for convolutions into smaller, localized sub-problems or "tiles". These tiles are specifically sized to fit within the GPU's fast, on-chip memory structures, allowing for significant data reuse.

Optimal kernel design rarely relies on a single level of tiling. Instead, it mirrors the hardware's memory hierarchy, implementing a multi-level tiling strategy that bridges Global Memory, Shared Memory, and the Register File.

#### 3.1.1.1 Block-Level Tiling (Shared Memory)

At the highest level of the on-chip hierarchy, tiling leverages Shared Memory. Rather than having individual threads repeatedly fetch overlapping data directly from slow Global Memory, a thread block cooperatively loads a unified tile of data into Shared Memory. Once the data resides in this fast SRAM, it can be accessed and reused multiple times by all threads within the block. For instance, in a standard Matrix Multiplication, loading a block level tile drastically reduces the total number of global memory reads, effectively trading careful indexing for a massive reduction in memory latency.

#### 3.1.1.2 Warp and Thread-Level Tiling (Registers)

While Shared Memory is significantly faster than Global Memory, it still incurs latency and is susceptible to bank conflicts. Therefore, the tiling hierarchy must extend down to the Register File (RF), which is the fastest and most abundant memory available per thread on the Streaming Multiprocessor (SM).

Thread-level or warp-level tiling involves dividing the block-level Shared Memory tile into even smaller sub-tiles. Each thread (or warp) fetches its specific sub-tile from Shared Memory directly into its private registers. All actual mathematical computations, especially those leveraging specialized hardware like Tensor Cores, operate directly out of these registers. By keeping the immediate working set exclusively in the Register File, threads can feed the arithmetic logic units (ALUs) at maximum theoretical throughput without stalling for intermediate memory reads.

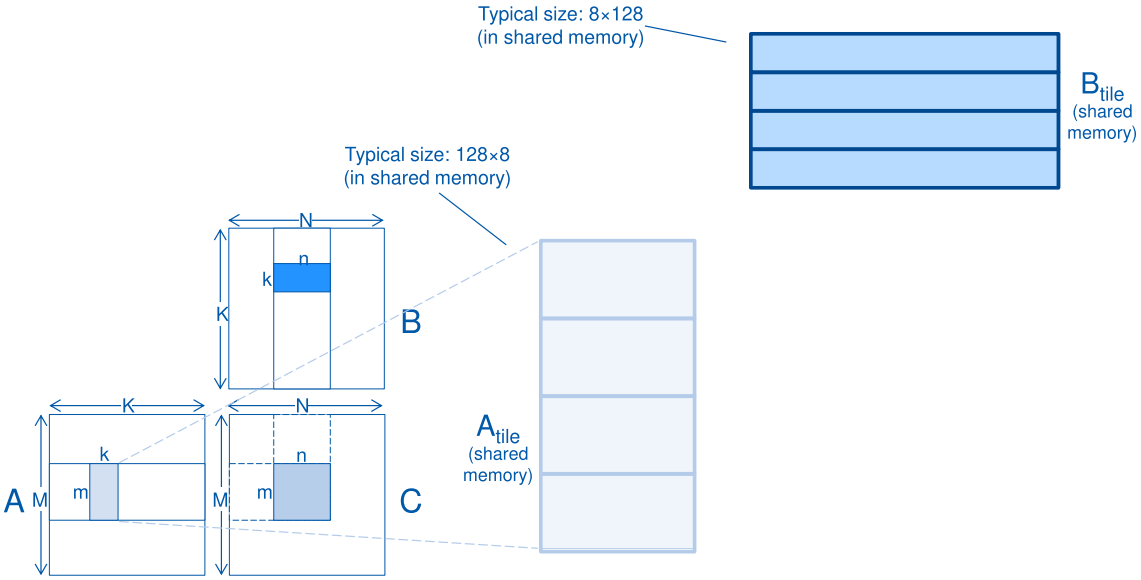


Figure 3.1: Block-level and thread-level tiling [28]

### 3.1.2 Software Pipelining

Software pipelining is a critical instruction scheduling technique used to hide memory access latency by overlapping memory fetching with active computation. While tiling maximizes data reuse, software pipelining ensures that the compute units (such as ALUs or Tensor Cores) are not starved while waiting for the next tile of data to arrive from the slower memory hierarchy. By proactively fetching data for future iterations of a loop while actively computing on the current iteration, highly optimized kernels can approach the theoretical maximum throughput of the underlying hardware.

#### 3.1.2.1 Double Buffering

The most fundamental implementation of software pipelining is double buffering, which corresponds to a two-stage pipeline. In this approach, memory is allocated for two separate tiles (or buffers). During any given loop iteration i, the Streaming Multiprocessor (SM) performs math operations on the data residing in Buffer 0, while simultaneously issuing memory requests to load the data for iteration i+1 into Buffer 1. In the subsequent iteration, the roles of the buffers swap. This technique effectively masks global memory latency behind arithmetic operations, provided that the computation time is roughly equivalent to, or greater than, the memory fetch latency.

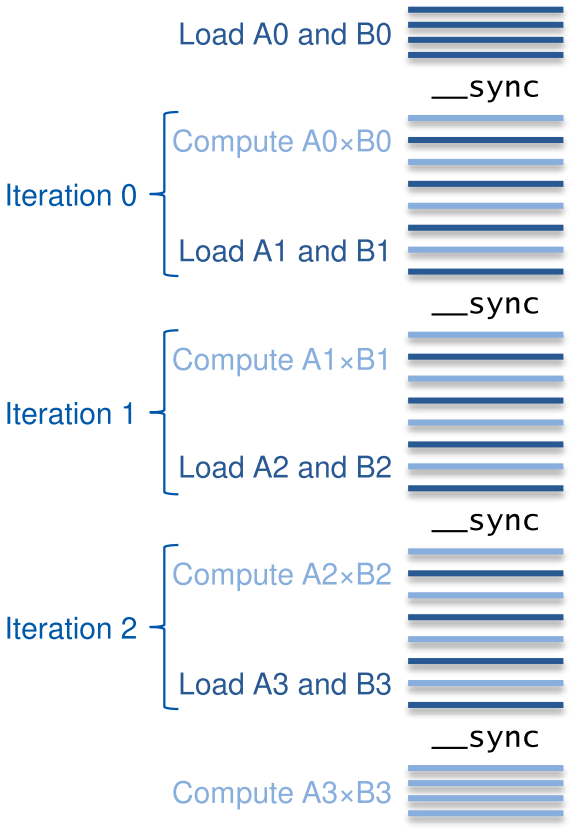


Figure 3.2: Double buffering example [28]

#### 3.1.2.2 Register-Based Pipelining

Historically, software pipelining was managed explicitly within the Register File. Threads would load the upcoming tile's data from Global Memory into a set of pre-allocated registers while concurrently computing on the current tile residing in a separate set of registers. While effective for simple kernels, this approach drastically increases register pressure. Allocating too many registers for pre-fetching limits the number of active warps that can reside on the SM, reducing overall occupancy and complicating the compiler's instruction scheduling.

#### 3.1.2.3 Asynchronous Memory Copies

To alleviate the register pressure caused by traditional pipelining, modern GPU architectures (such as the Ada Lovelace architecture used in our testbed) introduced hardware backed asynchronous memory copy instructions, specifically

`cp.async`. These instructions allow threads to initiate a direct, hardware-managed data transfer from Global Memory to Shared Memory, completely bypassing the Register File. Threads can issue the asynchronous fetch for future tiles and immediately proceed to compute on the active Shared Memory tile, synchronizing via a barrier only when the future tile is strictly required. This bypass mechanism enables deeper, multi-stage pipelines (e.g., three or four stages) without exhausting valuable register resources.

#### 3.1.2.4 Tensor Memory Accelerator (TMA)

The Hopper architecture further revolutionized software pipelining by introducing the Tensor Memory Accelerator (TMA). The TMA acts as a dedicated, asynchronous Direct Memory Access (DMA) engine capable of transferring complex, multi-dimensional tensor blocks from Global to Shared Memory autonomously. Instead of requiring every thread in a warp to calculate memory addresses and issue individual load instructions, a single thread can configure a TMA descriptor and issue a bulk transfer. This innovation completely offloads the address calculation and memory fetching workload from the SM threads, freeing up warp schedulers and execution units to focus entirely on mathematical operations.

#### 3.1.2.5 Tensor Memory (TMEM)

Building on the offloading capabilities of the TMA, the Blackwell architecture addresses the final major bottleneck in dense matrix computations: accumulator register pressure. In previous architectures, even as data was efficiently staged in Shared Memory, the resulting matrix accumulators still had to reside in the SM's general-purpose Register File, which severely limited warp occupancy during large-scale operations. To resolve this, Blackwell introduces Tensor Memory (TMEM), a dedicated, physical on-chip memory tier specifically designed to feed the Tensor Cores. Utilizing the `tcgen05` instruction set, TMEM acts as a direct, addressable memory space for Tensor Core operands and accumulators, completely decoupling the accumulation process from the Register File. This architectural shift not only eliminates the register capacity limits for large matrices but also breaks traditional warp-synchronous execution, enabling true per-thread scheduling where individual threads can issue mathematical operations to the Tensor Cores independently.

### 3.1.3 Memory Coalescing

Beyond caching and tiling, the physical pattern in which threads request data from Global Memory dictates the effective bandwidth of a kernel. Memory coalescing is a critical hardware level optimization that ensures the threads within a single warp access contiguous, properly aligned memory addresses. When a warp of 32 threads issues a memory load or store, the GPU's memory controller attempts to consolidate these requests into as few 32-byte, 64-byte, or 128-byte memory transactions as possible. If the threads access sequential addresses, the access is perfectly coalesced and served in a single, highly efficient transaction. Conversely, scattered, sparse, or strided memory accesses force the controller to issue multiple transactions to fetch cache lines where only a fraction of the data is actually used, drastically wasting memory bandwidth and bottlenecking the kernel's overall throughput.

The degree to which a kernel can successfully coalesce its memory accesses is heavily influenced by the tensor's physical data layout. A prominent example is the choice between NCHW (channels-first) and NHWC (channels-last) memory formats in vision models. While NCHW keeps spatial dimensions contiguous, highly optimized convolution kernels typically map the operation to an implicit GEMM to leverage Tensor Cores, which requires computing dot products across the channel dimension. In these scenarios, the NHWC format ensures that the channel elements are strictly adjacent in memory. This layout allows the threads within a warp to coalesce their memory fetches perfectly along the channel depth. Therefore, selecting the appropriate data layout is not merely an API convention, but a strict algorithmic prerequisite for enabling memory coalescing and achieving peak hardware utilization.

### 3.1.4 Avoiding Bank Conflicts

While tiling data into Shared Memory dramatically reduces Global Memory access latency, it introduces a unique microarchitectural challenge: bank conflicts. Shared Memory in modern GPUs is physically divided into equal-sized memory modules called banks (typically 32 banks, corresponding to the 32 threads in a warp). These banks are designed to be accessed simultaneously. However, if multiple threads within the same warp request different memory addresses that map to the exact same bank, the requests cannot be served concurrently. Instead, the hardware is forced to serialize these accesses, which degrades the effective memory bandwidth by a factor equal to the number of conflicting requests.

Highly optimized kernels, frequently need to read column wise data from a 2D tile stored in Shared Memory. If the row length of this tile is a multiple of the number of memory banks, threads in a warp attempting to access a column will simultaneously hit the exact same bank, causing a severe N-way conflict. To circumvent this, there is a technique called memory padding or skewing. By allocating a slightly wider tile in Shared Memory, the memory addresses are artificially shifted. This ensures that subsequent column reads map to distinctly different banks, transforming stalled, serialized accesses into fully parallel transactions.

### 3.1.5 Operator Fusion

In traditional deep learning framework execution, each mathematical operator (e.g., a Convolution, followed by an activation like ReLU, followed by a MaxPool) is dispatched by the CPU as a separate, distinct GPU kernel. While this modularity simplifies framework design, it introduces significant compute and memory overhead. Between every operation, the intermediate results must be written out to slow Global Memory and immediately read back in by the subsequent kernel. Furthermore, the Streaming Multiprocessor (SM) execution units stall while waiting for the host CPU to dispatch the next set of instructions.

Operator fusion (or kernel fusion) eliminates these bottlenecks by mathematically and programmatically combining multiple sequential operations into a single kernel execution. By computing the entire sequence within a single thread block, the intermediate data (such as the output of the Convolution) never leaves the Register File or Shared Memory before being passed directly into the next function (like ReLU). This drastically increases the arithmetic intensity of the kernel, ensuring the execution units spend their clock cycles performing math rather than waiting on memory round-trips or host-side scheduling.

## 3.2 Compute Optimizations

While memory optimizations ensure that the GPU's execution units are constantly fed with data, compute optimizations focus on maximizing the efficiency and throughput of the execution units themselves. The foundational step in extracting maximum compute performance from a GPU is correctly configuring how the work is dispatched to the hardware.

### 3.2.1 Selecting Optimal Launch Parameters

When a kernel is invoked, the programmer or compiler must define its launch configuration, specifically the grid dimension (number of thread blocks) and the block dimension (number of threads per block). Selecting the optimal values for these parameters is non-trivial and fundamentally dictates the kernel's theoretical occupancy, defined as the ratio of active warps to the maximum number of warps supported on a Streaming Multiprocessor (SM).

High occupancy is generally desirable because GPUs rely on warp-level context switching to hide instruction and memory latencies. When one warp stalls waiting for data, the SM's warp scheduler instantly swaps in another ready warp. To maintain a sufficient pool of ready warps, a kernel must launch with enough threads per block. However, maximizing the sheer number of threads is rarely the optimal solution due to strict microarchitectural constraints.

The optimal launch configuration is tightly bound by resource limits, specifically the Register File size and the available Shared Memory per SM. If a kernel requires a large number of registers per thread (register pressure), launching blocks with the maximum allowable threads (e.g., 1024 threads per block) will quickly exhaust the SM's register allocation. This forces the hardware to either spill registers to slow local memory or reduce the number of active blocks scheduled on the SM, drastically lowering occupancy. The same principle applies to Shared Memory: if a block allocates a massive shared memory footprint, fewer blocks can physically reside on the SM concurrently. Consequently, selecting optimal launch parameters is an exercise in balancing occupancy against Instruction-Level Parallelism (ILP).

In the context of modern machine learning infrastructure, identifying the perfect launch configuration for a specific hardware architecture and problem size is incredibly complex. State-of-the-art compilers like Triton manage this by employing auto-tuning, searching through a vast configuration space to dynamically select the best parameters at runtime. In contrast, explicitly hardcoding these bounds creates fragile kernels that may perform exceptionally well on a target GPU but suffer severe performance degradation when problem dimensions or hardware architectures change.

### 3.2.2 Utilizing Tensor Cores

While traditional CUDA cores are highly versatile processors, the overwhelming compute demands of modern neural networks are predominantly met by

Tensor Cores. Introduced in the Volta architecture and significantly enhanced in subsequent generations (including the Ada Lovelace architecture used in this study), Tensor Cores are highly specialized execution units designed to accelerate the core computation of deep learning: the Matrix Multiply Accumulate (MMA) operation ($D = AB + C$). By performing mathematically dense 4x4 or 16x16 matrix operations in a single clock cycle across a warp, Tensor Cores offer an order of magnitude increase in theoretical throughput compared to standard floating point ALUs.

However, extracting this performance requires strict adherence to hardware-level constraints, particularly regarding data types. Tensor Cores are inherently designed for mixed precision arithmetic. They do not natively operate on standard 32 bit floating point (FP32) inputs for their matrix multiplication phase. Instead, they require lower precision input formats, such as FP16, BF16, or the newer TensorFloat32 (TF32). To prevent catastrophic precision loss, the hardware performs the multiplication in the reduced format but computes the final accumulation in full FP32 precision. Consequently, highly optimized kernels and the ML compilers that generate them must aggressively downcast or cast inputs to these supported datatypes before feeding the execution units, a transformation that must be carefully managed to maintain algorithmic correctness.

Furthermore, leveraging Tensor Cores dictates the fundamental algorithmic structure of the kernel. Because these units are strictly hardware accelerated matrix multipliers, operations that are not natively matrix multiplications must be mathematically reframed to fit this paradigm.

At the software level, feeding these units requires precise coordination of all 32 threads within a warp using the Warp-Level Matrix Multiply-Accumulate (WMMA) API or lower level `mma.sync` PTX instructions. Threads must collaboratively load strictly formatted "fragments" of matrices A and B from Shared Memory into specific registers before triggering the Tensor Core instruction. If a kernel fails to structure its inner loops around these warp collective MMA instructions, it fundamentally bypasses the GPU's most powerful compute engines, resulting in severely bottlenecked performance.

### 3.2.3 Using Warp Level Primitives

While Shared Memory is the standard medium for data exchange between threads within a block, it is not always the most efficient choice for fine-grained, highly localized communication. NVIDIA GPUs group threads into warps (typically 32 threads) that are scheduled and executed in lockstep. To leverage this

architectural trait, modern CUDA architectures provide specialized warp level primitives, most notably the family of shuffle instructions (e.g., `__shfl_sync`, `__shfl_down_sync`, and `__shfl_xor_sync`).

These primitives allow threads within the exact same warp to exchange data directly between their private registers. Bypassing Shared Memory entirely for this intra-warp communication yields multiple critical performance benefits. First, it completely eliminates the latency associated with writing to and reading from Shared Memory SRAM, reducing the operation to a fast register to register transfer. Second, it frees up Shared Memory bandwidth and capacity for other data structures, reducing the likelihood of bank conflicts. Finally, warp-level primitives inherently synchronize the threads participating in the exchange at the hardware level, alleviating the need to invoke expensive, block-wide `__syncthreads()` barriers that would otherwise force all warps in a block to stall.

### 3.2.4 Loop Unrolling

Loop unrolling is a classic compiler optimization that is particularly crucial for maximizing compute throughput on GPUs. In a standard loop, the hardware incurs overhead for every iteration: it must increment the loop counter, evaluate the loop condition, and execute a conditional branch instruction. Loop unrolling mitigates this overhead by replicating the loop body multiple times within a single iteration, effectively reducing the total number of branch instructions executed.

Beyond merely reducing branching overhead, unrolling exposes a massive amount of Instruction-Level Parallelism (ILP) to the compiler. By expanding the loop body, the compiler has a much larger window of independent instructions to analyze. This allows it to aggressively reorder instructions, interleaving independent mathematical operations with memory fetches to effectively hide latency. For example, while waiting for a memory load initiated in the first unrolled iteration, the execution units can immediately process the math instructions of the second unrolled iteration.

### 3.2.5 Vectorization

While memory coalescing ensures that a warp accesses memory efficiently as a group, vectorization optimizes how individual threads issue those memory requests and process data. In standard execution, a thread might issue multiple

consecutive 32 bit load instructions to fetch individual floating point values. Vectorization explicitly instructs the hardware to fetch larger, contiguous blocks of data such as 64 bit or 128 bit chunks using built in vector datatypes like `float2`, `float4`, or `half2`.

By loading four 32 bit elements simultaneously in a single `float4` instruction, the kernel drastically reduces the total number of memory instructions issued by the Streaming Multiprocessor (SM). This reduces instruction overhead, minimizes address calculation logic, and helps saturate the global memory bus more effectively. Furthermore, once the vectorized data resides in registers, the thread can often perform computations on these packed types directly, increasing Instruction Level Parallelism (ILP).

## 3.3 Algorithmic Optimizations

While memory and compute optimizations focus on mapping a predefined sequence of operations onto the GPU hardware as efficiently as possible, algorithmic optimizations fundamentally alter the mathematical approach or logic used to solve the problem. Rather than tweaking microarchitectural nuances like register allocation or shared memory padding, an algorithmic optimization seeks to reduce the inherent time complexity of the task or restructure the logic so that it natively aligns with the massive parallelism of the hardware. This often involves trading careful, sequential logic for brute force, dense computations that can fully saturate thousands of parallel execution units. In the context of GPU programming, an algorithm with a theoretically higher asymptotic time complexity $(O(N^2))$ might drastically outperform a mathematically "superior" algorithm $(O(NlogN))$ simply because the former can be mapped to highly uniform, non-branching tensor operations that the hardware accelerates natively.

The crucial distinction of algorithmic optimizations is that they fall almost entirely outside the domain of traditional and modern Machine Learning compilers. Compilers such as Triton, XLA, or LLVM operate under strict constraints of semantic equivalence. Their primary function is to take the Intermediate Representation (IR) of the user's explicitly written code and lower it safely into machine instructions. While they excel at applying complex algebraic simplifications, loop unrolling, and operator fusion, they inherently lack the high level semantic understanding required to completely swap one algorithm for another. If a programmer writes a specific, stable mathematical formula to compute a value, the compiler's responsibility is to execute that exact formula as fast as

possible; it is not permitted to arbitrarily replace it with another sequence of operations and rewrite them into a completely different, faster mathematical construct.

Consequently, algorithmic optimizations represent the frontier of human expertise, and potentially, the most promising capability of Large Language Models in code generation. Because compilers are locked into optimizing the code they are given, discovering and implementing a superior algorithmic pathway requires reasoning about the global objective of the function, hardware constraints, and acceptable numerical stability bounds. Identifying these high level algorithmic substitutions is what truly separates baseline framework implementations from masterful, custom built kernels.

Chapter 4

# Performance evaluation

## 4.1 Testbed

The hardware these tests were executed on is the NVIDIA GeForce RTX 4090 (Ada Lovelace microarchitecture) with 24GiB of VRAM. This GPU has support for asynchronous memory copies to shared memory with `cp.async` although it lacks some capabilities introduced in the Hopper Architecture like the TMA (Tensor Memory Accelerator) or Thread Clustering. Its 4th generation Tensor Cores also have support for the TF32 and FP16 datatypes which are extremely relevant in this context.

Regarding software, we used KernelBench to evaluate kernels using a Docker container with all of the dependencies needed and stated in the official documentation.

The methodology consist on taking LLM-generated kernels from popular works and not only run them on our testbed but perform a exhaustive inspection of such code comparing it to what the Triton compiler generates. This involves careful examination of highly complex GPU code and of Intermediate Representation (IR) code or even PTX to be fully aware of the differences between both codes.

## 4.2 Kernel evaluation

The kernels evaluated in this research consist exclusively of forward pass implementations. Generally, forward passes receive more significant optimization efforts than backward passes, as they are utilized for both training and inference, whereas the backward pass is restricted to the training phase. Since inference is a high volume process that often represents a greater overall computational cost, it has historically remained a primary focus. Furthermore, backward passes present greater optimization challenges; consequently, the limited literature that addresses them [14], typically only demonstrates simple implementations. Table 4.1 shows all of the kernels evaluated, their level, source, GPU that was originally used and the claimed speedup obtained.

| Kernel | Level | Source | GPU used | Claimed Speedup |
|---|---|---|---|---|
| Convolution 2D | 1 | KernelBench | L40S | 1.8x |
| Conv2D + ReLU + MaxPool | 2 | KernelBench | L40S | 2.9x |
| LayerNorm | 1 | KernelBench | L40S | 4.8x |
| CrossEntropyLoss | 1 | KernelBench v3 | RTX 3090 | 3.84x |
| RMSNorm | 1 | FlashInferBench | B200 | 1.55x |
| Add + RMSNorm | 2 | Astra | H100 | 1.33x |
| GEMM + Sum + Divide + Scaling | 2 | CUDA-Agent | H20 | 24.04x |
| GEMM (fp32) | 1 | KernelBench V3 | B200 | 2.3x |
| GEMM (fp16) | 1 | FlashInferBench | B200 | 1.0x, 0.1x |
| Softmax | 1 | CUDA L1 & KernelBench v3 | L40S, RTX3090 | 1.0x, 1.14x |
| Diagonal Matrix | 1 | CUDA L1 | L40S | 64x |

Table 4.1: All the tested kernels on this work

### 4.2.1 Convolution 2D

A convolution is an operation where a small matrix (which is called kernel or filter) slides over an input matrix to extract features. This is the most important operator used in vision tasks. The mathematical formula is as follows:

$$out(N_i, C_{out_j}) = bias(C_{out_j} + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

Where $\star$ is a sliding dot product, $N$ is batch size, $C_{in}$ and $C_{out}$ the input and output channels respectively, $H$ and $W$ are the input height and width in pixels. In this kernel Stanford researchers claim 1.8x speedup for FP32. As before on our hardware we got different results but managed to still get 1.25x out of the box with this kernel.

To achieve competitive performance, a convolution kernel must be mapped to a General Matrix Multiplication (GEMM) to leverage the specialized Tensor Cores present in modern GPUs. Because Tensor Cores do not natively support the FP32 format, the LLM-generated kernel employs a strategy of downcasting FP32 elements to FP16 while maintaining accumulation in FP32. This precision reduction is subtle enough to pass standard correctness evaluations like KernelBench but represents a significant mathematical compromise. Furthermore, the kernel utilizes an "Implicit GEMM" algorithm, the standard for inference also used by PyTorch. The implementation incorporates several advanced optimizations, including double buffering, loop unrolling, shared memory skewing, vectorized memory access, and index precomputation

A critical evaluation of the LLM-generated kernel against the standard PyTorch implementation reveals differences in invocation overhead. The LLM executes a single CUDA kernel, whereas PyTorch typically dispatches two: an initial Triton kernel to transpose the input from NCHW to NHWC, followed by the convolution itself via the cuDNN library. While NHWC is the preferred format for GEMM-mapped convolutions, the initial transformation adds measurable overhead.

However, conducting a direct internal comparison of the convolution phase is impossible because cuDNN is closed-source and its implementation details remain proprietary. By examining the PyTorch source code, we can confirm that cuDNN utilizes the TF32 datatype for this operation. When PyTorch is configured to downgrade precision to FP16 to match the LLMs constraints, the LLM-generated kernel actually underperforms the baseline, achieving only 0.9x speedup. This suggests that the LLMs reported advantage is derived from exploiting the tolerance bounds of the validation suite rather than superior hardware utilization.

Because the closed-source nature of cuDNN obscures its internal logic, we also compared the LLM-generated CUDA kernel against a Triton kernel generated by PyTorch. Initially, the LLM kernel outperformed the Triton version by 4.24x, but this was largely due to Triton maintaining FP32/TF32 precision. After enabling FP16 downcasting in PyTorch, the performance gap narrowed to 1.85x. Comparing these implementations is complex because CUDA and Triton operate at different abstraction levels, and Triton's primary optimizations, such as loop hoisting, occur at the compiler level rather than within the kernel source. While the LLM kernel's structure might suggest Triton is missing optimizations like double buffering, closer investigation reveals that the compiler handles these complexities automatically.

On top of that, we also wanted to know how general these kernels are. We have already seen that vary across different hardware, but we also wanted to observe their behavior with different input dimensions. Until now we were testing with the default input which resembles AlexNet with a filter of 11x11 over an input of 224x224 with 96 output channels. We decided to test with other problems sizes as listed on the tables 4.2

| Layer | in_c | out_c | kernel | stride | padding | image_size |
|---|---|---|---|---|---|---|
| AlexNet Default | 3 | 96 | 11 | 4 | 2 | 224 |
| ResNet Stem | 3 | 64 | 7 | 2 | 3 | 224 |
| Standard 3x3 | 64 | 64 | 3 | 1 | 1 | 56 |
| Downsample 3x3 | 128 | 256 | 3 | 2 | 1 | 28 |
| Pointwise 1x1 | 256 | 1024 | 1 | 1 | 0 | 14 |

Table 4.2: Different sizes used to test the convolution

The first two (AlexNet Default and ResNet Stem) represent the first layer of a CNN, the Standard 3x3 a middle heavy layer and lastly a Downsample 3x3 layer that reduces the size of the image by half. No examples of depthwise convolutions are tested as the LLM generated kernel is hard-coded to only work with a value of `groups=1`, pointing out another lack of generality on such code. In the figure 4.1 we can see how different version behave under this different problem sizes

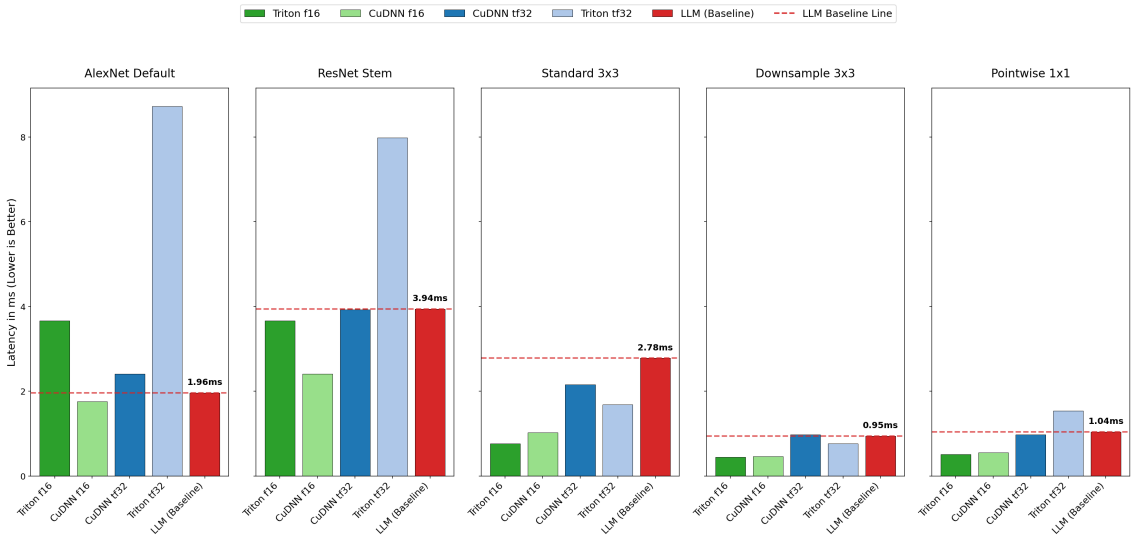


Figure 4.1: Comparison of Triton and CuDNN against the LLM kernel for Conv2D

We can see that the LLM generated kernel performs best in the AlexNet Default version, this is expected as this is the version the kernel was explicitly generated for but it still performs worse than cuDNN (which has the added overhead of the conversion from NCHW to NHWC). On the rest of the versions, the LLM generated kernel performs poorly and is outperformed by almost all other kernels.

Ultimately, the evaluation of the 2D Convolution kernel reveals a severe lack of generalization in the LLM-generated code. While it demonstrates competitive performance on the specific problem size it was explicitly prompted for (AlexNet Default), its efficiency degrades drastically across other standard layer configurations when compared to robust, hardware-optimized vendor libraries like cuDNN. This performance drop, coupled with the reliance on hardcoded parameters such as fixed group sizes, underscores a significant limitation: current LLM approaches tend to 'overfit' to specific prompt parameters rather than producing the versatile, generalized kernels required for dynamic deep learning workloads.

### 4.2.2 Conv2D + ReLU + MaxPool

This section evaluates a Level 2 kernel published by Stanford researchers prior to the release of KernelBench. The kernel integrates three sequential operations: a convolution (similar to the one analyzed in Section 4.2.1), a ReLU activation, and a MaxPool operation. The original authors claim a 2.9x performance speedup for this fused implementation.

The convolution segment of the LLM-generated kernel employs the same optimization strategies previously discussed: it downcasts data to FP16 precision and utilizes the NCHW memory format. Conversely, the default Triton compiler uses the TF32 datatype and transposes the tensors into the NHWC format. To ensure a fair and direct comparison, we explicitly enabled FP16 downcasting within Triton. Under this configuration, PyTorch intentionally bypasses the cuDNN library to facilitate operator fusion, relying exclusively on Triton's compilation capabilities.

The primary divergence between the two implementations lies in their approach to operator fusion. The Triton compiler successfully fuses the convolution with the ReLU activation; however, it leaves the MaxPool as a distinct, subsequent operation. This heuristic decision is rooted in the fact that MaxPool introduces a data dependency between thread blocks due to potential memory overlap. Consequently, dispatching a second, separate kernel inherently acts as a safe, global synchronization barrier. In contrast, the LLM-generated CUDA kernel

circumvents this limitation by utilizing a *scatter* approach with atomic operations, allowing it to aggressively fuse all three operators into a single kernel execution.

To evaluate the generalization of this aggressive fusion strategy, both implementations were benchmarked across the various problem sizes detailed previously in Table 4.2. The results, illustrated in Figure 4.2, demonstrate that the LLM-generated code fails to provide a speedup for any tensor dimension other than the specific configuration for which it was originally generated. This finding is consistent with the overfitting behavior observed in earlier tasks.

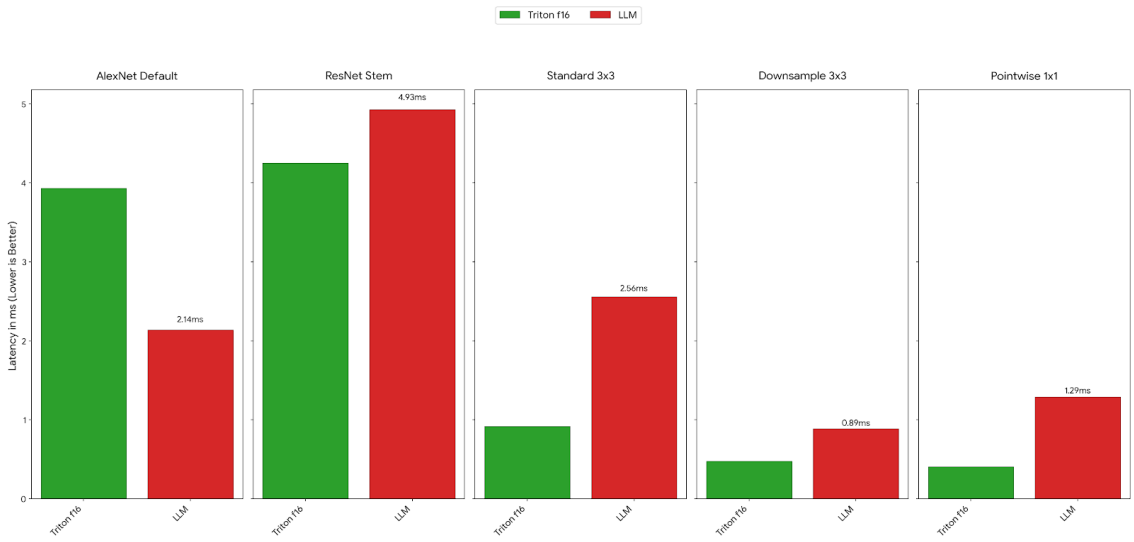


Figure 4.2: Comparison of Triton against the LLM kernel for Conv2D + ReLU + MaxPool

The underlying cause of this performance degradation relates to the scalability of atomic operations and memory layout choices. Fusing the three operators via atomics is highly effective when the output dimensions are relatively small (e.g., $55 \times 55$ for AlexNet). However, as the spatial dimensions increase, the massive influx of atomic operations creates severe memory contention, effectively serializing thread execution. Furthermore, while maintaining the NCHW format is acceptable for large convolutional filters (such as $11 \times 11$), it significantly hinders memory coalescing and cache utilization as filter sizes decrease.

Ultimately, while the LLM successfully identified a novel operator fusion strategy using atomics, this approach is inherently unscalable. The model achieved a significant speedup on its specific target configuration by aggressively fusing operations, but it failed to account for the detrimental memory contention caused

by atomic operations on larger output grids. Similar to previous tasks, the LLM prioritized a narrow, hardcoded "fast path" that proves unviable for generalized, dynamic workloads.

### 4.2.3 Layer Normalization

Layer Normalization is an operator that applies normalization across a layer over a mini-batch of inputs. This operation computes the formula below

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

This requires the calculation of the mean ($E[x]$) and variance ($Var[x]$) for the input, where $\epsilon$ is a constant for numerical stability and $\gamma, \beta$ are learnable affine parameters.

The LLM-generated kernel for this operation, originally presented by Stanford researchers, was reported to achieve a 4.8x speedup on an NVIDIA L40S GPU. However, benchmarks on our RTX 4090 (same microarchitecture) testbed yielded a more modest speedup of 1.44x. This discrepancy underscores significant portability issues inherent in LLM-driven kernel generation, where performance optimizations may be overly tuned to a specific hardware architecture

Despite the reduced performance on our hardware, the kernel still outperforms the baseline PyTorch implementation. The primary optimizations employed by the LLM include warp-level reduction primitives, efficient shared memory utilization, and vectorized memory access using `float4`. While SoTA compilers like Triton also implement these techniques, a fundamental algorithmic divergence exists. Triton utilizes the Welford Algorithm to compute variance, which is more computationally intensive but offers superior numerical stability. In contrast, the LLM generated kernel employs a "naive" reduction approach not suitable for production. Because standard benchmarks like KernelBench evaluate correctness using random inputs between 0 and 1, they fail to trigger the numerical instabilities inherent in the naive algorithm.

In summary, while the LLM-generated kernel for Layer Normalization achieves a measurable speedup, this performance gain is largely artificial. The LLM prioritizes speed over mathematical robustness by substituting the compute-heavy, numerically stable Welford algorithm used by SoTA compilers like Triton with a naive reduction approach. Consequently and as pointed out by other works, relying solely on basic correctness suites like KernelBench is insufficient for evaluating LLM-generated code in this domain, as these benchmarks fail

to capture the critical algorithmic compromises that would make the kernel unsuitable for production environments.

### 4.2.4 CrossEntropyLoss

The CrossEntropyLoss operator is utilized to calculate loss during neural network training by measuring the divergence between a true probability distribution ($p$) and an estimated distribution ($q$). The operation is mathematically defined as:

$$H(p,q) = -\sum_i p(i) log\ q(i)$$

This section examines an LLM-generated kernel sourced from KernelBench v3, generated by Claude 3 Opus for an NVIDIA RTX 3090. While the original authors claim a 3.84x speedup, executing this kernel on our RTX 4090 testbed yielded a more modest 1.12x speedup. This discrepancy further highlights the portability issues inherent in LLM-generated kernels when transitioning across different hardware generations

Despite the reduced speedup, the LLM-generated kernel still outperforms the PyTorch reference implementation for standard problem sizes. Because this kernel is written in Triton rather than CUDA, it allows for a direct comparison with the compiled Triton code generated by PyTorch. This comparison reveals two critical algorithmic differences that expose a severe lack of generality in the LLMs approach:

First, the generated kernel lacks a mechanism to ignore padding tokens (which typically carry a value of $-100$). If these padding tokens are present, the kernel's oversight can lead to mathematically incorrect results or invalid memory accesses. Second, the LLM-generated kernel employs a single, monolithic load instruction for the entire dataset. In contrast, the PyTorch compiler utilizes a loop to load and process data in manageable chunks. While a single-load approach is functional for small matrices, it inevitably causes severe register spilling when processing the massive amounts of data typical in LLM training.

To evaluate the impact of this design flaw, we benchmarked the kernel across various input dimensions that reflect real-world applications (see Table 4.3). The tested configurations include the default KernelBench dimensions, representative sizes for vision and segmentation tasks, and dimensions typical of both small and large LLMs.

| Domain | N | C | torch.compile (ms) | LLM (ms) |
|---|---|---|---|---|
| Default | 4096 | 1024 | 0.0349 | 0.0312 |
| ImageNet (Vision) | 1024 | 1000 | 0.0145 | 0.0132 |
| Segmentation | 262144 | 20 | 0.0371 | 0.302 |
| Small LLM (GPT-2) | 4096 | 50257 | 0.911 | 0.909 |
| Modern LLM (Llama 3) | 8192 | 128256 | 4.38 | 24.4 |

Table 4.3: Different configurations tested for CrossEntropyLoss

The results demonstrate that the LLM-generated kernel performs slightly better on the smaller vision and segmentation tasks. The speedup obtained for these smaller workloads stems from the LLM employing a "one-shot" execution strategy eliminating the overhead of chunked loops and calculating the softmax in a single pass rather than in an online manner like the PyTorch compiler. In essence, the LLM successfully wrote a "fast path" kernel similar to those found in highly specialized libraries, but one that is only viable for a narrow subset of problem sizes.

However, as tensor dimensions increase, the performance of the LLM kernel degrades significantly. In the largest configuration (Modern LLM), execution time becomes unacceptably slow due to the aforementioned register spilling (see Appendix A.3 for the compiled PTX code evidence). Ultimately, while the LLM-generated kernel for CrossEntropyLoss successfully minimizes overhead for small matrices that fit entirely into SRAM, this advantage completely collapses at production scales. The model's naive single-load approach ignores fundamental hardware constraints, and its failure to account for necessary edge cases renders it unsafe for general-purpose use.

### 4.2.5 RMSNorm

This operator applies Root Mean Square Layer Normalization over a mini-batch of inputs.

$$y_i = \frac{x_i}{RMS(x)} * \gamma_i \; where \; RMS(x) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

LLM-generated kernels for this operation have been explored in various works. This section specifically analyzes the kernel provided by FlashInferBench. While the baseline implementation evaluated in that study differs slightly from the original KernelBench version, both share a notable characteristic: rather

than employing the native PyTorch operator `nn.RMSNorm`, they compute the normalization by manually calculating the mean and the reciprocal of the square root (see Appendix A.6 for the exact reference code).

Various studies report different performance speedups up to 1.55x, depending on the specific LLM employed and the tensor dimensions tested. Notably, the public codebase provided by FlashInferBench was exclusively evaluated on an NVIDIA B200 GPU. To facilitate a fair comparison on our testbed, we first encapsulated the reference code within a standard PyTorch module class, ensuring compatibility with the KernelBench evaluation framework.

Under these controlled conditions, our benchmarks indicated identical performance for both the LLM-generated kernel and the baseline. The underlying reason for this parity is straightforward: the original reference code provided to the LLM lacked a `torch.compile()` wrapper, meaning it was executed in PyTorch's slower, unoptimized Eager mode. By wrapping the code in a class to work with KernelBench, graph compilation was inherently enabled for the baseline, instantly closing the performance gap.

This conclusion is substantiated by three key observations. First, the LLM-generated implementation is written in Triton and is structurally almost identical to the compiled Triton code automatically generated by PyTorch. Second, empirical tests within KernelBench demonstrate a speedup solely when compared against Eager mode execution; no measurable improvement exists when compared against `torch.compile()`. Finally, if the native `nn.RMSNorm` operator is utilized as the baseline instead of the manually unrolled reference code, the performance gap between Eager mode and compiled execution narrows significantly, corroborating the previously discussed overheads associated with dispatching multiple unfused operations.

### 4.2.6 Add + RMSNorm

This section transitions to the evaluation of Level 2 kernels by examining an operator that fuses Root Mean Square Layer Normalization (RMSNorm) with an addition operation. This specific implementation is sourced from the Astra study. Unlike previously evaluated kernels, this code was generated using the baseline kernels from the SGLang serving framework as a reference. Consequently, it functions more as an LLM-optimized kernel rather than an implementation generated entirely from scratch.

The original authors generated this kernel for an NVIDIA H100 GPU, claiming performance speedups ranging from 1.0x to 1.33x over the SGLang baseline across

various tensor sizes. The primary optimization introduced by the LLM involves replacing the baseline's shared memory tree reduction with a more efficient warp-level reduction (for details on the specific shuffle instructions utilized, see Appendix A.7).

To evaluate its portability and general efficacy, we benchmarked this LLM-optimized kernel on our RTX 4090 testbed, comparing it directly against PyTorch compiled with Triton (`torch.compile()`). The evaluation utilized the specific combinations of batch size (bs) and hidden size (hs) detailed in the Astra GitHub repository, with the performance results illustrated in Figure 4.3.

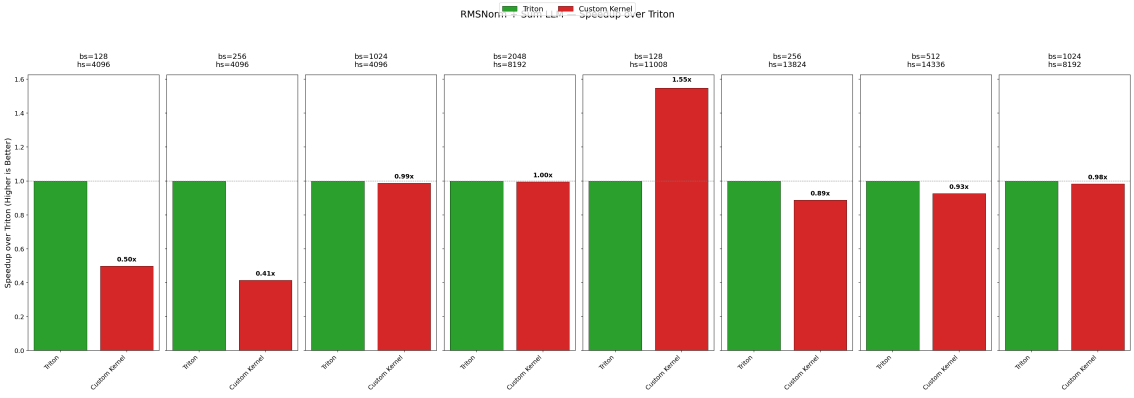


Figure 4.3: Comparison of LLM generated kernel vs Triton for the sizes mentioned in the Astra paper

The benchmarks indicate that both implementations achieve equivalent execution times across most tensor dimensions. However, Triton outperforms the LLM kernel at smaller sizes, while the LLM kernel achieves a notable 1.55x speedup in one specific configuration. Similar to the behavior observed in the Softmax operator (Section 4.2.9), this performance discrepancy is driven by PyTorch's compilation heuristics. For the majority of the configurations, Triton successfully fuses the entire operation into a single kernel. Conversely, in the specific case where Triton underperforms, the compiler heuristics prioritize maximizing hardware occupancy by splitting the operation into multiple distinct kernels. Because RMSNorm is fundamentally a compute-bound operation, bypassing operator fusion to increase occupancy introduces detrimental overhead, allowing the single kernel LLM implementation to excel in that isolated scenario.

Furthermore, an inspection of the Intermediate Representation (IR) confirms that Triton inherently utilizes the same warp-level reduction primitives that the LLM introduced to the SGLang reference code. Therefore, while the LLM success-

fully improved the specific SGLang baseline, it did not discover an optimization absent from SoTA compilers. The only instance where the LLM-generated code outperforms PyTorch occurs when the compiler's internal heuristics miscalculate the trade-off between operator fusion and hardware occupancy.

### 4.2.7 GEMM + Sum + Divide + Scaling

This section evaluates an LLM-generated kernel sourced from the CUDA-Agent study, which utilizes reinforcement learning (RL) and multi-agent systems to generate CUDA code. While the original authors report a 24.04x speedup for this specific operator sequence, empirical evaluation on our testbed yielded a 12.32x speedup.

The target workload integrates four distinct operations: a General Matrix Multiplication (GEMM), a summation, a division, and a scalar multiplication. Mathematically, these operations are defined sequentially as follows:

1. Gemm $Y_{i,j} = \sum_k X_{i,j} W_{j,k}$
2. Division and sum $Z_i = \sum_j \frac{Y_{i,j}}{2}$
3. Scaling $Out_i = S \cdot Z_i$

This entire sequence can be consolidated into a single mathematical expression:

$$Out_i = \frac{S}{2} \sum_j \sum_k X_{i,j} W_{j,k}$$

The performance gain achieved by the LLM in this scenario does not originate from advanced microarchitectural memory or compute optimizations. Instead, the model successfully identified an opportunity to reduce the algorithm's asymptotic complexity. By recognizing that the sum of the weights could be precalculated, the LLM effectively transformed a computationally expensive Matrix Multiplication (GEMM) into a much lighter dot product operation. The LLM ultimately generated a kernel based on this equivalent but significantly more efficient formulation:

$$Out_i = \frac{S}{2} \sum_k X_{i,j} (\sum_j W_{j,k})$$

The generated code executes this new mathematical pathway efficiently, appropriately utilizing shared memory, vectorized loads, loop unrolling, and operator fusion.

However, this LLM generated kernel has many hardcoded values and it will not work outside the configuration it was evaluated on. Although a clever optimization, we can also categorize this as a "human error" as we did with the diagonal matrix kernel since in this scenario the LLM acted more like a "algebraic reviewer" than like a kernel expert.

However, despite this impressive algorithmic simplification, the kernel relies heavily on hardcoded values and fails to execute correctly outside the specific tensor dimensions for which it was originally generated. Similar to the Diagonal Matrix kernel analyzed in Section 4.2.10, this performance gain is more accurately categorized as the correction of a flawed, human written reference implementation rather than a demonstration of profound hardware mastery. The LLM acted effectively as an "algebraic reviewer" by identifying a high level mathematical simplification, but its reliance on hardcoded parameters once again restricts the kernel's generalization and renders it unviable for dynamic, real world workloads.

### 4.2.8 GEMM

The General Matrix Multiplication (GEMM) operator is foundational to deep learning, serving as the computational backbone for DNNs. Consequently, it is historically one of the most heavily optimized operations by expert engineers. Because SoTA libraries already execute GEMM at the absolute limits of hardware capability, claims of significant performance speedups from LLM-generated GEMM implementations are exceedingly rare and warrant rigorous scrutiny.

This section first evaluates an LLM-generated GEMM kernel sourced from KernelBench-v3, originally generated for an NVIDIA B200 GPU with a claimed 2.30x speedup. However, on our RTX 4090 testbed, this kernel achieves merely a 0.60x speedup, representing a significant performance regression. The original performance gain on the B200 architecture can be readily explained by inspecting the source code: the LLM-generated Triton kernel explicitly enables the TF32 data format, thereby leveraging hardware Tensor Cores. By default, Triton does not enable TF32 for standard FP32 GEMM operations. Therefore, the LLM did not discover a novel algorithmic optimization; it merely toggled a compiler flag that alters numerical precision.

Furthermore, the fact that this LLM kernel performs worse on our hardware than the baseline PyTorch Triton implementation (which does not utilize TF32) underscores the severe architectural differences between a data-center B200 GPU and a consumer-grade RTX 4090. It demonstrates that even high-level Triton kernels, which theoretically abstract hardware complexities and rely heavily on the compiler for generalization, can fail to transfer across hardware generations when an LLM explicitly hardcodes configuration parameters.

In addition to full precision, we examined a half-precision (FP16) GEMM implementation sourced from FlashInferBench. While the best-performing LLM-generated kernel yielded a 1.0x baseline parity, source code inspection revealed that the model merely invoked the highly optimized cuBLAS library rather than synthesizing a custom kernel. When excluding these high-level workarounds and evaluating a purely LLM-generated, custom CUDA kernel for GEMM, the reported performance plummeted to a 0.1x speedup (a 10x slowdown)

An analysis of the generated CUDA code reveals that the LLM completely omits critical microarchitectural optimizations, such as software pipelining, shared memory skewing, and implicit vectorization. Furthermore, the selection and application of these optimizations is a highly non-trivial task, many algorithmic transformations are mutually exclusive or introduce competing resource constraints when combined.

This dramatic performance degradation highlights the extreme complexity inherent in optimizing GEMM at the microarchitectural level. Even for expert human engineers, maximizing GEMM performance is a highly iterative and time-consuming process. In the current literature, LLM generation pipelines are typically restricted to a fixed, limited number of compilation iterations, which is manifestly insufficient for a computational problem of this magnitude. This complexity is mirrored in modern compilers such as Triton, which may execute up to 50 distinct optimization passes for a single kernel, many of which are specifically tailored for GEMM workloads. However, given unbounded execution time and infinite iterative feedback it is hypothetically possible that an LLM could approach SoTA performance.

### 4.2.9 Softmax

Softmax is a common activation function and mathematical operator defined as follows, where $z$ represents the input vector:

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

This operation takes a set of raw values and maps them to a probability distribution, ensuring that the sum of all resulting values equals 1.

This section evaluates two generated kernels for the Softmax operator: a CUDA kernel sourced from the CUDA-L1 study and a Triton kernel from KernelBench v3. For the CUDA-L1 evaluation, we examined a kernel generated for an L40S GPU, which yielded no measurable performance improvement (1.0x speedup). For KernelBench v3, we examined a kernel generated for an RTX 3090, which claimed a 1.14x speedup.

The CUDA-L1 kernel was promptly discarded from our evaluation. An inspection of its source code revealed numerous hardcoded values, an absence of safety checks, and flawed assumptions regarding memory alignment. These deficiencies render the kernel unstable and fundamentally unreliable for a rigorous comparison against the PyTorch reference implementation. Conversely, the Triton kernel generated by KernelBench v3 does not exhibit these vulnerabilities, allowing for a direct, fair comparison with the Triton code compiled by PyTorch. We benchmarked this operator across various standard tensor dimensions (detailed in Table 4.4); the performance results are illustrated in Figure 4.4.

| **Name** | **Batch** | **Dimension** |
|---|---|---|
| KernelBench Default | 4096 | 16,384 |
| ImageNet Classification | 256 | 1,000 |
| LLM Decode (GPT vocab) | 32 | 32,000 |
| LLM Train (GPT vocab) | 1024 | 32,000 |
| LLM Decode (Llama 3) | 64 | 128,256 |
| LLM Train (Llama 3) | 512 | 128,256 |
| Attention rows (medium) | 2048 | 4,096 |
| Moderate LLM hidden | 1024 | 8,192 |

Table 4.4: Different Softmax tensor sizes tested

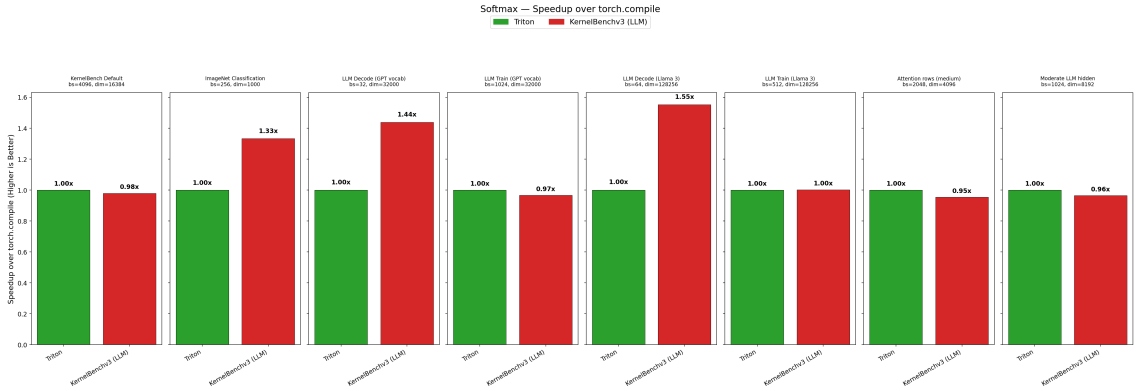


Figure 4.4: Comparison of Triton against the LLM Kernel for Softmax

In the majority of the tested configurations, both implementations achieve identical execution times. However, for specific larger tensor sizes, the LLM-generated kernel yields a measurable speedup. Unlike the previously analyzed cases, this LLM-generated kernel demonstrates strong generality; therefore, the performance gain cannot be attributed to a hardcoded selection of optimal launch parameters. Indeed, for the configurations where performance is equivalent, the source code of both the LLM kernel and the PyTorch Triton kernel is remarkably similar, as both correctly employ an online Softmax computation strategy.

The performance discrepancy observed in larger tensor sizes originates from PyTorch's internal compilation heuristics. To maximize GPU hardware occupancy for these massive workloads, the PyTorch compiler intentionally splits the operator into several distinct kernels rather than performing aggressive operator fusion (see Appendix A.5). This heuristic ultimately backfires: because Softmax is a strictly memory bound operation, the overhead introduced by reading and writing to intermediate global memory buffers between kernel launches severely degrades overall performance. By maintaining the execution entirely within a single fused kernel, the LLM-generated code successfully avoids this memory bandwidth bottleneck, allowing it to outperform the PyTorch compiler in these specific scenarios.

### 4.2.10 Diagonal Matrix

This kernel performs a matrix multiplication between a diagonal matrix (represented solely by its diagonal elements) and a dense matrix. This section examines the code provided by the CUDA-L1 study, which claims an exceptional 64x performance speedup for this operation.

However, the code generated by the LLM in this instance is neither a CUDA nor a Triton kernel; rather, it is a modification of the high level PyTorch reference code. The performance discrepancy arises because the original reference implementation utilized a highly inefficient method for memory allocation and matrix multiplication, which the LLM successfully replaced with a more optimal broadcasting approach (see Appendix A.4 for reference).

While this substitution yields a significant speedup, the improvement is fundamentally predicated on flaws within the original reference code. It highlights a human error where the author of the reference was unaware of the severe inefficiencies associated with using one high level method over another. This conclusion is corroborated by the KernelBench authors, who subsequently corrected this oversight in their public GitHub repository upon its discovery.

In summary, the staggering 64x performance gain observed in this task is a mirage. Rather than writing a novel, highly optimized CUDA or Triton kernel, the LLM merely acted as an automated code reviewer, identifying and correcting a highly inefficient Python reference implementation. While swapping a full matrix multiplication for memory efficient broadcasting is undoubtedly the correct optimization, it does not demonstrate an LLMs ability to navigate low level GPU hardware constraints.

** The detailed source code supporting the claims presented in this thesis can be found in Appendix A and B.*

Chapter 5

# Conclusions and future work

## 5.1 Contributions

This thesis presents a comprehensive evaluation of Large Language Model (LLM)-generated GPU kernels for machine learning, critically analyzing the strengths and limitations of this increasingly prominent approach to code generation. As the demand for highly optimized deep learning infrastructure grows, relying on LLMs to automate kernel development has garnered significant attention. By systematically comparing these AI-generated implementations against established SoTA infrastructure, such as standard machine learning compilers and specialized vendor libraries, this study establishes a clearer understanding of the true capabilities of current models.

A primary takeaway from this research is that while LLMs are frequently capable of writing functionally correct kernels, claims that they consistently outperform SoTA compilers are largely illusory as seen in other works [14] [17]. The evaluations demonstrate that models often achieve superficial speedups by relying on techniques that severely compromise the kernel's generality, numerical precision, or algorithmic robustness. Furthermore, this study highlights that despite significant community efforts, contemporary benchmarking suites designed for LLM-generated kernels remain fundamentally limited. These evaluation frameworks often fail to capture critical edge cases or numerical instabilities, thereby validating kernels that contain underlying algorithmic errors and rendering them unsafe for deployment in production environments. Nevertheless, the analysis

acknowledges that in certain highly constrained scenarios, LLM-generated code can at least achieve performance parity with current SoTA implementations.

Beyond evaluating the models themselves, another significant contribution of this work lies in utilizing LLM-generated code as a diagnostic tool to identify inefficiencies within existing compiler infrastructures. By rigorously comparing the execution profiles of LLM-generated kernels with those produced by SoTA compilers, this research uncovered suboptimal decision-making within framework heuristics. Specifically, the analysis revealed instances where the PyTorch compiler unnecessarily prioritizes hardware occupancy at the expense of operator fusion, inadvertently degrading execution efficiency. Identifying these heuristic flaws provides a clear pathway for future improvements in traditional machine learning compiler architectures.

Finally, this study identifies the contexts in which LLMs currently offer genuine utility for kernel optimization. The research demonstrates that LLMs can successfully generate highly optimized, fixed-size kernels, providing efficient computational pathways that are particularly valuable for hardware architectures or specific models where layer dimensions remain strictly hardcoded. Moreover, the analysis highlights the significant potential of LLMs to function as high-level algebraic reviewers. By recognizing opportunities to factorize or mathematically simplify expressions prior to execution, LLMs can discover algorithmic optimizations that fall entirely outside the strict semantic equivalence constraints of traditional compilers, thereby yielding substantial performance improvements.

## 5.2 Future Work

Building upon the evaluations presented in this thesis, several promising avenues for future research emerge, particularly concerning the integration of Large Language Models and traditional machine learning compilers. Rather than positioning AI models as outright replacements for existing infrastructure, future work should explore collaborative pipelines. A primary direction involves deploying the LLM as a preliminary algebraic reviewer. In this proposed architecture, before the traditional compiler processes the computational graph, an LLM would analyze the high-level mathematical operations to identify factorization opportunities and non-standard algorithmic simplifications. By applying these mathematical transformations first, the LLM can unlock structural optimizations that fall outside the strict semantic bounds of conventional compilers and that could be manually or empirically validated. The simplified, mathemati-

cally optimized graph can then be passed to the traditional compiler for robust, hardware-specific lowering, merging the creative pattern-matching of the LLM with the safety guarantees of the compiler.

Furthermore, the diagnostic utility of LLMs demonstrated in this study can be systematically expanded. Future research should focus on utilizing LLMs as an automated probing mechanism to continuously identify and rectify suboptimal decision-making heuristics embedded within standard compiler codebases. Concurrently, this methodology can be leveraged to aggressively generate specialized, fast-path kernels for frequently encountered, rigid tensor shapes. By delegating these highly constrained scenarios to LLM-optimized fast paths, the broader infrastructure can bypass general-purpose compiler overhead, achieving peak execution speeds while maintaining the compiler as a safe fallback for dynamic or unpredictable workloads.

Finally, while this research has primarily evaluated forward-pass execution, extending this analytical framework to backward passes represents a critical next step. The backpropagation phase of neural network training introduces intricate gradient computations, complex memory access patterns, and atomic accumulation strategies that differ significantly from inference workloads. Investigating the capacity of LLMs to safely and efficiently generate reverse-mode automatic differentiation kernels will be essential for fully determining the viability of AI-driven code generation within comprehensive deep learning training infrastructures.

# Bibliography


[1] Why are CUDA kernels hard to optimize? Wayne joubert. https://www.johndcook.com/blog/2025/08/27/why-are-cuda-kernels-hard-to-optimize/, 2025. Accessed 2026-05-02.

[2] Elliot Arledge. Kernelbench v3: Rebuilding a gpu kernel benchmark from first principles. https://elliotarledge.com/blog/kernelbench-v3, 2026. Accessed 2026-21-09.

[3] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. https://arxiv.org/pdf/2507.11948, 2025.

[4] Jinqi (Kathryn) Chen. Towards effortless high-performance kernel development for llm workloads. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, August 2025.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. https://arxiv.org/pdf/1802.04799, 2018.

[6] Suraj Subramanian Dhruv Matani. Efficient pytorch: Tensor memory format matters. https://pytorch.org/blog/tensor-memory-format-matters/, 2021. Accessed 2026-04-09.

[7] Sukru Burc Eryilmaz. Mlperf hpc v1.0: Deep dive into optimizations leading to record-setting nvidia performance.

https://developer.nvidia.com/blog/mlperf-hpc-v1-0-deep-dive-into-optimizations-leading-to-record-setting-nvidia-performance/, 2021. Accessed 2026-04-09.

[8] Horace He. welfordreduce slows down forward layernorm in a bunch of cases. https://github.com/pytorch/pytorch/issues/120184, 2024. Accessed 2026-04-09.

[9] heavyai. What is gpgpu? definition and faqs. https://www.heavy.ai/technical-glossary/gpgpu. Accessed 2026-21-09.

[10] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. https://arxiv.org/pdf/2308.10620, 2023.

[11] Niranjan Kamat and Arnab Nandi. A closer look at variance implementations in modern database systems. https://arxiv.org/pdf/1509.04349, 2015.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6), May 2017.

[13] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3), July 2003.

[14] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. https://pub.sakana.ai/static/paper.pdf, 2025.

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

[16] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators. https://arxiv.org/pdf/2502.14752v1.

[17] Shiyang Li, Zijian Zhang, Guangyan Sun, Yuebo Luo, Winson Chen, Yanzhi Wang, Mingyi Hong, and Caiwen Ding. Cudahercules: Benchmarking hardware-aware expert-level cuda optimization for llms. https://arxiv.org/pdf/2605.08467, 2026.

[18] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. https://arxiv.org/pdf/2507.14111, 2025.

[19] Mobile SoCs: The Wild West of Domain Specific Architectures. Vijay janapa reddi. https://www.sigarch.org/mobile-socs/, 2018. Accessed 2026-05-02.

[20] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? https://arxiv.org/pdf/2502.10517, February 2025.

[21] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26:80 – 113, 03 2007.

[22] PyTorch Team. Pytorch documentation. https://docs.pytorch.org/docs, 2026.

[23] Tensorflow Team. Tensorflow documentation. https://www.tensorflow.org/api_docs, 2026.

[24] Phil Tillet. Blackwell programming for the masses with openai triton. https://semianalysis.com/wp-content/uploads/2025/03/Blackwell-Programming-for-the-Masses-With-OpenAI-Triton-Phil-Tillet.pdf, 2025.

[25] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for ai systems. https://arxiv.org/pdf/2504.17577, 2025.

[27] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. https://arxiv.org/pdf/2509.07506, 2025.

[28] Wen-mei Hwu, et.al. Pumps+ai summer school lecture 2: Advanced gemm, 2024.

[29] Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. https://arxiv.org/pdf/2601.00227, 2026.

[30] Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Dialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of llms. https://arxiv.org/pdf/2601.15727, 2026.

[31] Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. https://arxiv.org/pdf/2511.20100, 2025.

# Appendices

Appendix A

# Proofs

This appendix contains detailed proofs for discoveries made along the work.

## A.1 Layer Normalization

The LLM-generated kernel operates in three stages, mirroring the structural approach employed by Triton.

In the first stage, the kernel computes 32 partial sums and sums of squares, allocating one per thread block. To perform these reductions efficiently, it leverages the highly optimized `__shfl_down_sync` instruction for warp-level intra-block reduction. The intermediate results are subsequently written to shared memory, where warp 0 folds the results using a final `__shfl_down_sync` operation.

The second and third stages are computationally lighter. In the second stage, a single thread per block aggregates the 32 partial results from the first stage to compute the final mean and variance for each batch.

In the third stage, utilizing the precomputed mean and variance, the kernel normalizes every value. The LLM optimizes this normalization pass by vectorizing the memory accesses and computations, mapping each thread to handle four elements simultaneously via the `float4` CUDA datatype.

However, an inspection of the PyTorch-generated Triton kernels reveals a fundamental algorithmic divergence that accounts for the performance discrepancy discussed in Chapter 4. Specifically, an analysis of stages one and two in the Triton intermediate representation (IR) shows that reductions are strictly performed using numerically stable helper operations, namely

`triton_helpers.welford_reduce` and `triton_helpers.welford`. In contrast, the LLM utilizes a naive reduction approach (computing the sum of squares directly) [11] [8].

Despite this mathematical compromise, the LLM-generated kernel consistently passes standard correctness evaluations. This occurs because benchmark suites typically initialize inputs using `torch.rand()`, which generates uniformly distributed floating-point values between 0 and 1. These restricted inputs fail to trigger the numerical stability issues that the naive variance calculation exhibits on larger, real-world distributions.

## A.2 Convolution 2D

The performance discrepancies and overheads discussed in Chapter 4 regarding the Convolution 2D operator can be traced back to two fundamental factors: memory layout constraints and microarchitectural alignment limitations.

First, we must differentiate between the NCHW (channels-first) and NHWC (channels-last) memory formats. The acronyms denote the physical memory arrangement of the Number of samples (N), Height (H), Width (W), and Channels (C). When computing convolutions via Implicit GEMM on Tensor Cores, the hardware accesses data along the channel dimension.Consequently, ensuring that channels are contiguous in memory is paramount for efficient memory coalescing. For this reason, PyTorch dynamically converts the default NCHW tensors to the NHWC format before dispatching the operation to cuDNN. While NHWC is the recognized gold standard for this workload [7] and is explicitly recommended by the official PyTorch documentation [6], PyTorch natively defaults to NCHW, unlike other frameworks such as TensorFlow which inevitably introduces the initial transposition overhead observed during evaluation.

Second, an analysis of the compiled Triton Intermediate Representation (IR) clarifies why the Triton-generated kernel does not utilize software pipelining or double buffering in this scenario. When Triton was explicitly forced to utilize software pipelining via a custom configuration, the compiler issued a warning indicating that utilizing registers for this operation might actually degrade performance. At the microarchitectural level, optimal software pipelining relies on specialized asynchronous instructions, such as `cp.async`, which strictly require memory fetches to be aligned to four bytes. However, the initial layers of standard vision models process images with exactly three RGB channels. This 3-channel depth makes a perfect four-byte alignment impossible, effectively rendering

hardware-accelerated software pipelining ineffective for this specific hardware and datatype configuration. Even when operating in FP32 precision, where optimal pipelining is technically achievable, it restricts the pipeline to a single 4-byte element per thread, severely limiting the utility of the optimization.

## A.3 CrossEntropyLoss

To fully contextualize the performance degradation and register spilling claims made in Chapter 4, it is necessary to examine the mathematical formulation of the fused CrossEntropyLoss operator.

In standard machine learning classification tasks, the target probability distribution ($p$) is typically represented as a one-hot vector, where all elements are zero except for a 1 at the true class index $c$. Because of this, the standard cross-entropy summation collapses, simplifying the formula exactly to the Negative Log-Likelihood (NLL) of the true class:

$$H(p, q) = -log\ q(c)$$

In a native implementation, this evaluates as two sequential operations. First, a softmax function normalizes the raw neural network outputs (logits, denoted as $z$) into a valid probability distribution:

$$q_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Second, the Negative Log-Likelihood is applied over this normalized distribution:

$$Loss = -log(\frac{e^{z_c}}{\sum_j e^{z_j}}) = -z_c + log(\sum_j e^{z_j})$$

Finally, to guarantee numerical stability and prevent floating-point overflow when exponentiating large logit values, the maximum logit value ($z_{max}$) is found and subtracted from all logits prior to exponentiation. This standard "log-sum-exp" trick yields the final, robust mathematical form:

$$Loss = -z_c + z_{max} + log(\sum_j e^{z_j - z_{max}})$$

The LLM-generated kernel attempts to compute this entire sequence in a single, monolithic pass. While this "one-shot" strategy successfully minimizes

memory round-trips for small matrices, it completely breaks down at scale because the required number of private registers scales linearly with the tensor's column dimension.

An analysis of the compiled PTX assembly directly proves the severe register spilling claim discussed in Chapter 4. While modern hardware architectures strictly limit the number of architectural registers a single thread can allocate (with maximum bounds typically capping at 255 or 512 registers depending on the specific Streaming Multiprocessor configuration), the PTX code generated for the large vocabulary workloads features allocations such as:

```
.reg .b32 %r<3191>;
```

This instruction reveals that the kernel is attempting to use over 3,000 registers per thread, vastly exceeding physical hardware limits. Consequently, the compiler is forced to aggressively spill these excess variables into much slower local memory (L1 cache and Global Memory). This hardware-level bottleneck provides explicit evidence as to why the kernel's execution time becomes unacceptably slow on production-scale LLM workloads.

## A.4 Diagonal Matrix

To fully understand the superficial nature of the performance speedup discussed in Chapter 4, we must examine the original PyTorch reference implementation provided to the LLM. The baseline code executed the following operation:

```
return torch.diag(A) @ B
```

This implementation is algorithmically and memory inefficient. The `torch.diag` function takes a 1D tensor A of size $N$ and allocates a full $N \times N$ matrix in global memory, populated almost entirely by zeros. Subsequently, the @ operator forces PyTorch to dispatch a full General Matrix Multiplication (GEMM) ($O(N^3)$ to multiply this sparse, memory-heavy matrix with the dense matrix B.

Rather than generating a novel, low-level CUDA or Triton kernel, the LLM simply modified the high-level Python API call to:

```
return A.unsqueeze(1) * B
```

This is undoubtedly a clever optimization. By utilizing `unsqueeze(1)`, the code converts the 1D tensor A into a 2D column vector. The * operator

then leverages PyTorch's native broadcasting semantics to perform a simple, memory-efficient element-wise multiplication across the rows of B ($O(N^2)$). This completely bypasses the massive memory allocation of the empty $N \times N$ matrix and avoids the computationally expensive GEMM operation entirely.

## A.5 Softmax

As established in Appendix A.3, production-grade Softmax implementations must employ a mathematically robust approach to ensure numerical stability and prevent floating-point overflow when exponentiating large logits. This requires a multi-pass approach over the data to compute the maximum value ($z_{max}$) before exponentiation, modifying the standard Softmax equation to:

$$\frac{e^{z_i - max}}{\sum_{j=1}^{K} e^{z_j - max}}$$

When utilizing PyTorch's Triton compiler for large tensor dimensions, specifically, the "LLM Decode (Llama 3)" configuration evaluated in Chapter 4 (Batch = 64, Dimension = 128,256) the compiler's heuristic actively intervenes to increase Streaming Multiprocessor (SM) occupancy. Allocating a single thread block per batch would result in only 64 active blocks, which severely underutilizes the GPU hardware. To artificially inflate occupancy, PyTorch splits the massive 128,256-element row into 16 smaller tiles, launching $64 \times 16 = 1,024$ independent thread blocks.

However, chunking the data across multiple blocks necessitates global synchronization to compute the final batch-level maximums and sums. Consequently, PyTorch is forced to execute a parallel tree reduction across five distinct kernel launches:

1. The first kernel spawns 1,024 blocks to calculate 1,024 partial maximums.

2. A second kernel aggregates these to find the absolute maximum for each of the 64 batches.

3. A third kernel calculates the partial sums of the exponentiated values ($e^{z_i - z_{max}}$) across the 1,024 blocks.

4. A fourth kernel reduces these to the final 64 batch-level denominators.

5. A fifth, final kernel performs the element-wise division to yield the normalized output.

This compiler heuristic is theoretically logical: for memory-bound operations, higher occupancy allows the hardware to spawn more warps, enabling the warp scheduler to better hide global memory latency. However, by splitting the computation across five kernels, it fundamentally shatters operator fusion. The intermediate partial results must be continuously written to and read from Global Memory, generating massive memory traffic (as illustrated in Figure A.1). The LLM-generated kernel outperforms the compiler simply by overriding this occupancy heuristic, maintaining the entire reduction within a single fused kernel. While this limits the number of active warps, the sheer reduction in Global Memory accesses overwhelmingly compensates for the diminished latency hiding capabilities.

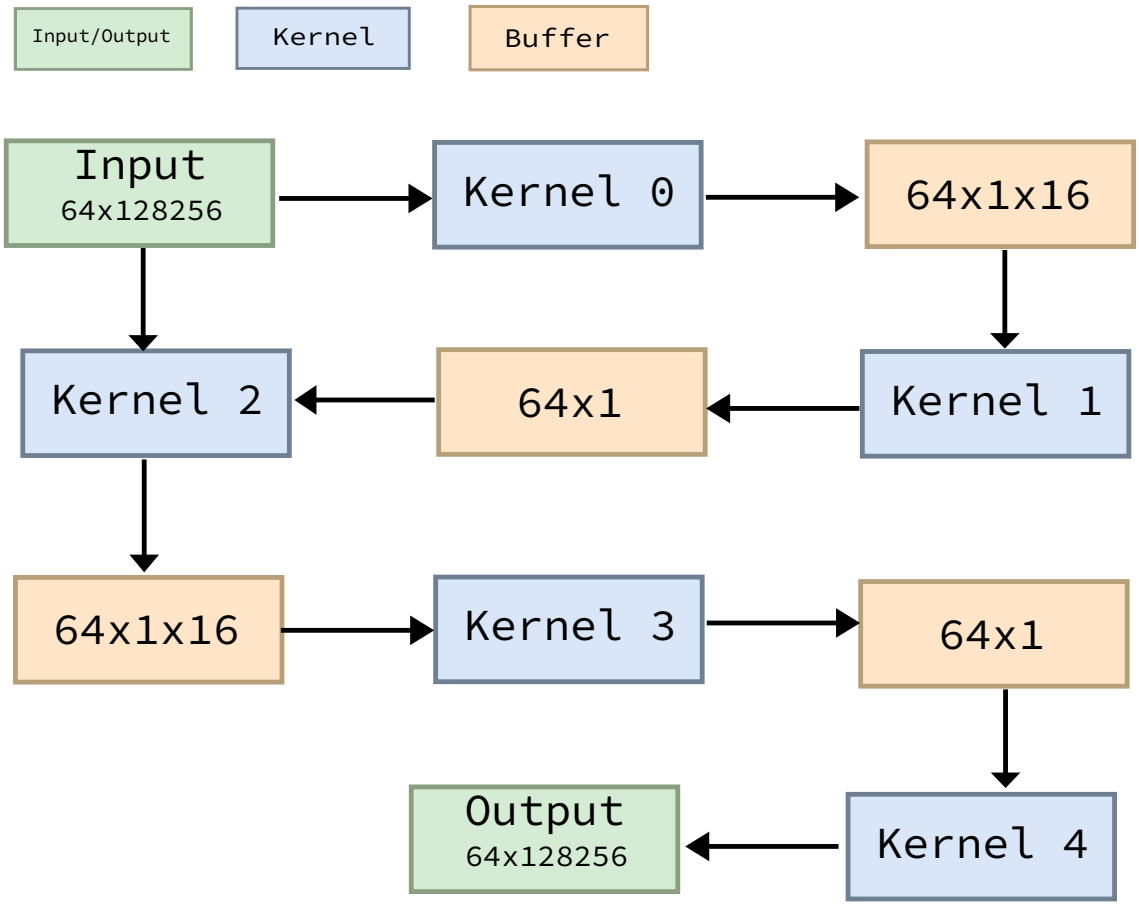


Figure A.1: Memory flow to increase occupancy

## A.6 RMSNorm

To contextualize the illusory performance gains reported for the LLM-generated RMSNorm kernel evaluated in Chapter 4, it is essential to examine the specific Python reference code supplied to the LLM. The baseline implementation explicitly computes the normalization using manually unrolled operations rather than calling the highly optimized, native PyTorch operator:

```
import torch

@torch.no_grad()
def run(hidden_states, residual, weight):
    _, hidden_size = hidden_states.shape

    EPS = 1e-6

    x = hidden_states.to(torch.float32) + residual.to(torch.float32)
    inv_rms = torch.rsqrt(x.pow(2).mean(dim=-1, keepdim=True) + EPS)
    y = (x * inv_rms) * weight.to(torch.float32)
    return y.to(hidden_states.dtype)
```

While this explicit mathematical formulation is functionally equivalent to PyTorch's native `nn.RMSNorm`, its execution profile diverges catastrophically depending on the runtime environment. When executed in PyTorch's default Eager mode (as was done in the baseline benchmarking) the Python interpreter dispatches this function as a sequence of distinct, isolated GPU kernels: one for the addition, one for the exponentiation, one for the mean reduction, one for the reciprocal square root, and two for the subsequent element-wise multiplications.

Because these discrete operations lack operator fusion, intermediate results must be repeatedly written to and fetched from slow global memory, introducing immense memory bandwidth bottlenecks and host-side kernel launch latencies. In contrast, `nn.RMSNorm` evaluates the entire mathematical formula within a single, optimized operation. When the LLM-generated kernel outperforms this reference code, it is merely simulating the operator fusion that should have already been applied natively.

## A.7 RMSNorm + Add

When evaluating the performance characteristics of the fused RMSNorm + Add operator, understanding the underlying reduction mechanism used to compute the variance is critical. The efficiency of this calculation relies heavily on how partial results are aggregated across threads, a process typically handled via either a Shared Memory tree reduction or a register-level warp reduction.

In a traditional parallel tree reduction, threads compute partial results and store them in the Streaming Multiprocessor's (SM) Shared Memory. The reduction then proceeds in $\log_2(N)$ steps, with half of the active threads reading values, aggregating them, and writing the new partial results back to Shared Memory at each step. While algorithmically efficient, this approach introduces significant microarchitectural overhead. It necessitates continuous read/write cycles to memory, which are susceptible to latency and bank conflicts, and mandates explicit, block-wide synchronization barriers (e.g., `__syncthreads()`) after every single step to prevent Read-After-Write (RAW) data hazards.

In contrast, modern GPU architectures expose a vastly more efficient mechanism for localized reductions: warp-level shuffle instructions, such as `__shfl_down_sync`. Because the 32 threads within a single warp are executed in physical lockstep by the warp scheduler, the hardware allows them to directly exchange data residing in their private registers.

As illustrated in the code below, utilizing `__shfl_down_sync` fundamentally alters the kernel's execution profile. By shifting the data exchange from Shared Memory to direct register-to-register routing, the kernel entirely bypasses memory access latency. Furthermore, because the shuffle instruction intrinsically acts as a synchronizing operation for the participating threads within the warp, it completely eliminates the need for expensive explicit barrier synchronization. While this hardware-accelerated mechanism is strictly bounded to the warp size limit of 32 threads, employing it to fold the initial reductions before staging the final results to Shared Memory drastically minimizes overall memory traffic and instruction overhead compared to a monolithic tree reduction.

```
/* tx = threadIdx.x, BS = BLOCK_SIZE */
__shared__ float sm[BS];

float s = ...;        // per-thread sum
sm[tx] = s;
__syncthreads();

for (int off = BS/2; off > 0; off >>= 1) {
  if (tx < off)
    sm[tx] += sm[tx + off];
  __syncthreads();
}
...
```

(a) Baseline: shared-memory tree reduction

```
/* lane = tx & 31, warp = tx >> 5 */
float s = ...;        // per-thread sum

unsigned m = 0xffffffffu;      // intra-warp
for (int off = 16; off > 0; off >>= 1)
  s += __shfl_down_sync(m, s, off);

__shared__ float ws[BS/32];    // one per warp
if (lane == 0)
  ws[warp] = s;
__syncthreads();

...
```

(b) Optimized: warp-level shuffle, brief shared-memory finalize

## A.8 Conv2D + ReLU + MaxPool

To fully comprehend the performance degradation of the fused Conv2D + ReLU + MaxPool kernel across varying workloads, it is critical to first understand how the spatial dimensions of the output tensor scale. The output size of a convolutional layer (e.g., the output width $W_{out}$) is governed by the input dimension ($W_{in}$), the filter size ($K$), the padding ($P$), and the stride ($S$), following the standard formula:

$$W_{out} = \lfloor \frac{W_{in} + 2P - K}{S} \rfloor + 1$$

Both the memory layout (NCHW) and the reliance on atomic operations play a detrimental role in the kernel's execution profile when evaluated across different layer configurations.

The LLM-generated kernel relies on atomic operations to safely compute the MaxPool reduction across overlapping memory regions. While functional for smaller output grids, this strategy introduces severe memory contention as the output scales. For example, in the ResNet Stem configuration, while the filter is relatively large ($7 \times 7$), the calculated output spatial dimension grows to $112 \times 112$. This massive influx of atomic instructions effectively serializes thread execution, crippling the Streaming Multiprocessor and obliterating any latency advantages gained by fusing the operators.

Conversely, the Standard $3 \times 3$ workload highlights the flaws of the kernel's hardcoded NCHW layout. Even though the output size is moderate ($56 \times 56$), the smaller filter size inherently lowers the kernel's arithmetic intensity compared to the ResNet Stem. Without a heavy mathematical workload to mask memory latency, the poor memory coalescing and L1 cache underutilization caused by the NCHW format become the dominant performance bottlenecks, rendering the kernel highly inefficient.

Resolving this inter-block synchronization bottleneck without resorting to atomics requires advanced hardware features, such as the Thread Block Clusters introduced in the Hopper and Blackwell microarchitectures. This feature would theoretically allow multiple thread blocks to cooperatively synchronize, enabling safe, atomic-free operator fusion for pooling layers. However, an analysis of the generated code confirms that none of the LLM kernels evaluated in this study leverage this architectural capability.

## A.9 GEMM + Sum + Divide + Scaling

To formally understand the 12.32x performance speedup reported for this operator sequence in Chapter 4, we must analyze the mathematical formulation of the workload. The naive sequence, comprising a General Matrix Multiplication (GEMM), a summation across columns, a division, and a scalar multiplication, can be consolidated into the following mathematical expression:

$$Out_i = \frac{S}{2} \sum_j \sum_k X_{i,j} W_{j,k}$$

Assuming the input tensor $X$ has dimensions $(N, H)$ and the weight tensor $W$ has dimensions $(H, D)$, executing this formula directly mirrors the default behavior of standard frameworks like PyTorch. The engine will first allocate memory and compute the full GEMM ($X \times W$), an operation governed by an asymptotic time complexity of $O(N \cdot H \cdot D)$.

However, because the summation over $k$ (the columns of the weight matrix) is a linear operation, the summations can be algebraically factored and reordered. By applying the distributive property, the LLM-generated kernel alters the execution pathway to evaluate the following mathematically equivalent expression:

$$Out_i = \frac{S}{2} \sum_k X_{i,j} (\sum_j W_{j,k})$$

This seemingly simple algebraic factorization has profound architectural implications. By precalculating the row-wise sum of the static weights $\left(\sum_k W_{j,k}\right)$, the formulation entirely eliminates the need to compute the intermediate $N \times D$ matrix. The operation is thus reduced to computing a 1D weight vector, followed by a batched dot-product between the rows of $X$ and this new vector.

Consequently, the time complexity plummets from $O(N \cdot H \cdot D)$ down to $O(H \cdot D) + O(N \cdot H)$. As argued in Chapter 4, this proves that the kernel's massive performance gain is not the result of advanced, microarchitectural hardware optimizations, but rather the consequence of the LLM identifying a high-level mathematical simplification that bypassed the computationally expensive GEMM entirely.

Appendix B

# Source code of LLM generated kernels & Triton kernels

The code of the LLM generated kernels and the reference Triton kernels evaluated on this work can be found at this GitHub repository.

Appendix C

# List of acronyms

**AI :** Artificial Intelligence.

**API :** Application Programming Interface.

**BF16 :** Brain Floating Point 16.

**CPU :** Central Processing Unit.

**CuBLAS :** CUDA Basic Linear Algebra Subroutines.

**CUDA :** Compute Unified Device Architecture.

**CuDNN :** CUDA Deep Neural Network.

**DSA :** Domain-Specific Accelerator.

**DL :** Deep Learning.

**DSL :** Domain-Specific Language.

**FP16 :** 16-bit Floating Point.

**FP32 :** 32-bit Floating Point.

**GEMM :** General Matrix Multiplication.

**GPGPU :** General-Purpose Computing on GPUs.

**GPU :** Graphics Processing Unit.

**IR :** Intermediate Representation.

**LLM :** Large Language Model.

**ML :** Machine Learning.

**MLIR :** Multi-Level Intermediate Representation.

**NLL :** Negative Log-Likelihood.

**RLVR :** Reinforcement Learning with Verifiable Rewards.

**SM :** Streaming Multiprocessor.

**SoTA :** State of The Art.

**TF32 :** 32-bit Tensor Float.

**TMA :** Tensor Memory Accelerator.

**TMEM :** Tensor Memory.