PEAK: A Performance Engineering AI-Assistant for GPU Kernels Powered by Natural Language Transformations

MUHAMMAD USMAN TARIQ, Stanford University, USA
ABHINAV JANGDA, Microsoft Research Redmond, USA
ANGELICA MOREIRA, Microsoft Research Redmond, USA
MADAN MUSUVATHI, Microsoft Research Redmond, USA
TYLER SORENSEN, Microsoft Research Redmond and University of California Santa Cruz, USA

Advancements in large language models (LLMs) are showing promising impact in software development and programming assistance. However, these models struggle when operating on low-level backend code. This challenge is exacerbated in the domain of GPU kernels, where performance-critical details are coupled to rapidly evolving hardware characteristics and available code examples are sparse.

In this work, we introduce Peak, a Performance Engineering AI-Assistant for GPU Kernels powered by natural language transformations. Peak utilizes the key insight that iterative code transformations (optimizations) can straightforwardly be written in natural language, and then carried out by LLMs. Thus, these transformations can be rapidly developed, encoding general portable optimizations, but also easily specialized to specific GPU devices and even kernels. These natural transformations are supported by a modular and extensible infrastructure that additionally performs validation and performance evaluation. We demonstrate the flexibility of Peak by instantiating it for three backends, CUDA, HIP, and HLSL, and create 16 natural transformations for optimizing matrix multiplication kernels. We show that our resulting implementations are competitive with vendor libraries when available, and for HLSL (without a library) our implementations match the hardware documented FLOPS. Peak allows the fine-grained exploration of several research questions around how LLMs behave in this domain, including characterizing transformations and their errors; and how performance evolves along optimization sequences. Peak provides an interface that can either be utilized by performance engineers to improve productivity, or driven completely autonomously (e.g., by an AI agent), providing a forward-compatible design that can continue to improve with advances in AI capabilities.

1 INTRODUCTION

Advances in large language models (LLMs) have significantly impacted software development, assisting, or even fully automating, a range of programming tasks [2, 13]. Due to the abundance of high-level code examples and mature frameworks, LLMs have been most effective in front-end and application-level domains, while low-level backend programming remains a greater challenge [5]. Nonetheless, given the potential of LLMs to improve productivity and democratize software development, extending AI assistance to lower-level code is a promising research direction. GPU kernels are a natural target: they are notoriously difficult to write and tune, yet have enormous performance implications, particularly in the context of modern AI workloads. Even modest reduction in kernel runtime can yield significant gains in cost efficiency and energy consumption, especially at scale.

Despite efforts to provide more accessible GPU programming languages [38, 39], development of efficient GPU kernels remains a domain reserved for experts, requiring deep understanding of complex memory and concurrency hierarchies, synchronization models, and architecture-specific optimizations. Compilers for these programming languages necessarily aim to be general-purpose, while maximizing performance requires performing optimizations that are specific to the kernel, the particular hardware backend, and possibly specific sizes of the inputs. Even if specialized compilers are engineered to optimize to the peak performance for some kernels, the GPU ecosystem evolves rapidly, with new architectural features required to fully exploit hardware capabilities, making it

Authors' addresses: Muhammad Usman Tariq, Stanford University, USA; Abhinav Jangda, Microsoft Research Redmond, USA; Angelica Moreira, Microsoft Research Redmond, USA; Madan Musuvathi, Microsoft Research Redmond, USA; Tyler Sorensen, Microsoft Research Redmond and University of California Santa Cruz, USA.

very difficult for these frameworks to keep up. However, performance engineering efforts tend to follow some structure; if this structure can be captured, especially utilizing new AI capabilities, then there may be opportunities to impact this domain.

Recognizing this potential, there have been several prior (and ongoing) efforts to generate optimized GPU kernels using AI, spanning academic work, large tech companies, and startups. For example, KernelBench [27] and the AI CUDA Engineer [29] explores optimizing CUDA kernels in PyTorch using LLMs. NVIDIA has demonstrated how LLMs can be used to optimize attention kernels [8], and several startups appears to be building a product around this use case, e.g., see [33–35]. While these efforts introduce promising techniques for transforming GPU code, they share a critical limitation: most adopt an all-or-nothing and opaque approach to automation, with minimal support for human-AI collaboration, interpretable iterative refinement, or extensible modular interfaces. This rigidity has led to brittle behavior and unintended side effects, in two cases, the system made critical mistakes which were not caught by the testing infrastructure [24, 40].

1.1 PEAK: Optimizing GPU Kernels through Natural Language Transformations

This paper presents Peak, a GPU kernel optimization framework, powered by natural language transformation specifications executed by LLMs, to assist performance engineers. Peak captures the essence of expert performance-engineering processing with the following components:

Natural Transformations. Performance engineers deploy a set of optimization strategies, each expressed informally in natural language (either explicitly as documentation or implicitly in their thinking). Peak's core contribution is to embrace this concept through *natural transformations*. These natural transformations are written in natural language and range from general strategies, such as "unroll a loop" to precise directives, such as "tile the inner loop over the K dimension," that are specific to the kernel being optimized. Because natural transformations are simple to write (e.g., as compared to formal compiler passes), they can be highly specialized to a specific kernel.

Kernel Context. A kernel context consists of a GPU kernel, host-side code that launches the kernel, input sizes, and a set of performance tuning parameters. This provides sufficient scope to perform aggressive optimizations (matching the performance of handwritten libraries for complex kernels), while also checking correctness and analyzing performance. Natural transformations can transform one kernel context into a new kernel context using LLMs.

Correctness Validators and Performance Evaluators. GPU kernels are complex, as they are executed by many threads across a complex hierarchy with features that are often poorly documented; errors are common, even when programmed by experts, let alone generated by an LLM. Thus, any natural transformation should be rigorously checked. Thus, Peak provides an interface for *correctness validators* to check the functional correctness of a kernel context. The base correctness validator simply compares kernel output values to a baseline. Moreover, Peak exposes an extensible interface for adding backend-specific validators, such as the compute sanitizers from NVIDIA. In the case of a failure, a previous kernel context can be restored; from this, the LLM could simply retry, or fall back to human-in-the-loop debugging, enabling developers to refine or rephrase natural transformations.

Similar to the correctness validators, the performance evaluators operate on a kernel context. These provide information about the performance, which can be attached to the kernel context and used to drive decisions about further optimizations. The basic performance evaluator simply samples tuning parameters and checks the runtime of the kernel. However, these are extensible and we illustrate this by incorporating more advanced tools, including the OpenTuner [4] autotuning framework and NVIDIA's Nsight profiler.

Performance Workflows and Portability. Performance engineers iteratively apply transformations using a mix of intuition, experimentation, and domain expertise. Peak provides a structured way to capture this exploration with its *performance workflow* abstraction. Users string together a sequence of natural transformations in a way that resembles `git` for software versioning. Every time that a natural transformation modifies a kernel context, it can be analyzed for correctness and performance; upon successful completion, it creates a checkpoint in the performance workflow. These checkpoints can be named, compared, restored, and revisited. As users dynamically explore their optimization process, the performance workflow and its checkpoints create a documentation of this exploration, similar to popular blog posts in this area [6], enabling others to replicate successful workflows and explore similar optimizations in other contexts.

A key aspect of Peak is that its fundamental core is general: that is, it does not utilize any backend specific tools, or operate on any specific programming language. Thus, it can be easily ported to different backends. In this work, we implement support for CUDA (NVIDIA GPUs), HIP (AMD GPUs), and HLSL (a portable shading language). However, its extensible interface also allows new tools to be incorporated for specific backends. For example, CUDA has a rich ecosystem of tools, and these could be implemented, e.g., in the correctness validators or performance evaluators. Similarly, branching in performance workflows allow users to write custom transformations that are specific to certain backends, while also sharing more general transformations across backends.

Immediately Pragmatic and Future Compatible. All of the interfaces provided by Peak can be accessed directly by users, e.g., manually or in a script. Furthermore, they also have an MCP interface, so that they can be accessed by natural language during kernel development in an IDE. However, the interfaces can also be driven by an AI agent, creating a completely automated workflow. Thus, Peak is designed to be immediately pragmatic, providing productivity benefits for performance engineers today, while also being future-compatible, enabling workflows to become increasingly autonomous as AI technologies evolve.

1.2 Optimizing Matrix Multiplication and Exploring Research Questions

Given the low-cost of writing natural transformations, Peak can be used in narrow domain specific optimization situations, which is often the case for performance-engineers, who must hyper optimize a small number of kernels. In this style, we focus our evaluation on a single, but critically important, kernel: matrix multiplication (or MatMul). This kernel is a fundamental operation in many high-impact domains where GPUs are deployed at scale, specifically AI. At the same time, it is notoriously difficult to optimize due to the need to balance memory hierarchies, thread utilization and backend-specific features such as tensor cores. As such, kernels are often specialized not only to backend, but to specific devices and even specific input sizes.

Although optimized libraries for MatMul exist (e.g., cuBLAS), they are complex and difficult to adapt or modify. It is useful to have an incremental optimization approach where modifications (e.g., such as those done in kernel fusion) can be incorporated. Furthermore, the knowledge base around MatMul can be useful in more complex kernels that share similar patterns, e.g., convolution and attention. Furthermore, as we illustrate with HLSL, the knowledge base can be used to build libraries for widely deployed GPUs that might otherwise not have a high-performance library.

We explore both `fp32` (32-bit floating point) and `fp16` (16-bit floating point) variants of MatMul and two sizes, 2048 x 2048 and 4096 x 4096 (abbreviated to 2k and 4k). We implement 16 natural transformations, which encapsulate 37 LLM calls. These transformations range from general-purpose GPU optimizations (e.g., loop tiling) to backend- or device-specific strategies (e.g., exploiting tensor cores) and are described in Sec. 4.1.

Table 1. Performance summary across devices, backends, and precisions. Speedup is given over the simple input kernel. The two numbers in each column denote values for square matrix of size 2048 & 4096. The % Max FLOPS is given as a ratio of the maximum FLOPS we observed in top performing library calls (cuBLAS and hipBLAS for NVIDIA and AMD) or for Qualcomm, over the reported FLOPS for the device.

Device	Backend	Precision	Baseline Speedup	% of Max FLOPS	Transformations
NVIDIA A6000	CUDA	fp32	9.36 & 10.36	95.0 & 91.6	11
		fp16	41.06 & 45.34	99.4 & 89.5	9
AMD MI200	HIP	fp32	24.39 & 25.21	91.9 & 86.2	8
		fp16	36.14 & 39.56	48.2 & 37.2	5
Qual. Adreno X1-85	HLSL	similar	4.16 & 4.71	107.3 & 109.8	3
Average			19.52	78.26	7.3

Evaluation and Research Questions. To evaluate Peak, and more broadly to characterize the capabilities of LLMs in GPU kernel optimization, we instantiate a realistic, semi-autonomous optimization scenario. For each GPU and each MatMul variant (`fp32` and `fp16`), we provide a sequence from our natural transformations that mirrors how a human performance engineer might plan an optimization strategy. Peak then executes this sequence, applying each transformation, validating correctness, and evaluating performance. A summary of the GPUs and their resulting performance is shown in Tab. 1. Our final kernels have significant speedup over their baseline inputs and competitive performance with vendor supplied libraries. Surprisingly, although we could not find a standard BLAS library for HLSL, we found that a small number of transformations produced MatMul kernels that achieved slightly over the reported number of FLOPS for this device. This shows the power of Peak in transferring knowledge from one backend to another.

At the time of writing, we note that Peak embodies an attractive design choice in this fast-moving area. On one hand, it outperforms many existing on this difficult kernel (matrix multiplication); on the other hand, it does not require a complex RL infrastructure. For example, the KernelBench leaderboard reports MatMul kernels to be around 20% of library performance. At the time of writing, the AI CUDA Engineer reports their MatMul kernels to be at 42% of library performance (the public leader has since seemed to be taken down). The CUDA-L2 system reports higher performance on matrix multiplication, but requires a complex RL pipeline and thus, likely many curated examples [35]. While PEAK requires lightweight manual prompts, we achieve an average of 78% of library performance and above 90% in many cases. Furthermore, Peak can easily be extended with natural transformations once performance issues are diagnosed, e.g., for AMD `fp16`, for which there are generally fewer resources optimization strategies and Peak currently performs suboptimally.

The human provided sequence of natural transformations is not a limitation of Peak. In practice, the transformation space could be searched automatically, or sequences could be proposed by an LLM, potentially guided by a learned or curated knowledge base. Our configuration simply provides a practical and interpretable starting point for systematically exploring the capabilities and limitations of LLM-assisted GPU optimization. Furthermore, this setup enables us to explore a set of research questions at fine-granularity. These questions aim to illuminate both the strengths and current limitations of LLMs in the domain of GPU kernel optimization, helping to identify where existing models are effective and where further research is needed.

Contributions. We introduce Peak, a performance engineering AI assistant for GPU kernels powered by natural transformations. To summarize, our contributions are:

- The design of a modular and extensible framework built around enabling safe and performant natural language driven transformations for GPU kernels (Sec. 3)
- An implementation across three GPU backends (CUDA, HIP, and HLSL), demonstrating both cross-backend knowledge transfer and backend-specific specialization. (Sec. 4.2).
- A case study optimizing MatMul kernels, achieving competitive performance and optimal GFLOPS for backends without a library (Sec. 4.3).
- An investigation of research questions offering insight into the current capabilities and limitations of LLMs for GPU kernel optimization (Sec. 5).

2 BACKGROUND: MATRIX MULTIPLICATION ON GPUS

We first review the architecture of modern GPUs using the example of an optimized MatMul kernel. Unless otherwise noted, we adopt the terminology used by NVIDIA's CUDA programming model.

2.1 GPU Programming Model

Most GPU programming models adopt a split execution model between the *host* (CPU) and the *device* (GPU). A GPU *kernel* is a function executed by many GPU threads typically following Single Instruction Multiple Data (SIMD) paradigm. A GPU kernel is written using a GPU programming language, such as, CUDA for NVIDIA GPUs, HIP for AMD GPUs, and HLSL is a portable shading language that can target many different GPUs. The kernel programming model closely reflects the underlying hardware hierarchy, exposing low-level control needed to implement efficient, high-performance kernels.

GPUs contain multiple Simultaneous Multiprocessors (SMs) to support large-scale parallelism. Each SM typically contains 64 or 128 CUDA cores, where each core executes one thread. Data center GPUs contain over 100 SMs while mobile or consumer-class GPUs typically contain between 2 and 20 SMs. This hierarchical design enables GPUs to scale performance across a wide range of workloads and device classes.

A GPU organizes threads in a three-dimensional grid, where each thread has a three-dimensional index. The grid of threads is divided into a group of consecutive threads, known as *thread blocks*. Consecutive threads of a thread block are further divided into fixed-size groups known as a *warp*, where threads execute in a Single Instruction Multiple Threads (SIMT) model. In a SIMT model, all threads in a warp attempt to execute the same instruction in lockstep; however, when control flow divergence occurs, threads can take different execution paths, reducing the execution efficiency. The size of a warp depends on the hardware, e.g. 32 for NVIDIA GPUs, 64 for some AMD GPUs, and 128 for Qualcomm GPUs. The host launches a GPU kernel based on a launch configuration that specifies the number of thread blocks in three dimensions and the number of threads for each thread block. This three-dimensional layout allows developers to naturally express computations over multidimensional data (e.g., images, matrices, volumes).

A GPU kernel reads its input data and writes its output data to the GPU DRAM, typically referred to as the *global memory*, because it is accessible to all threads of the GPU. The host is responsible for allocating buffers on the global memory and transferring data from/to these buffers. The host must wait for the GPU kernel to finish before reading the output from global memory buffers. A smaller and faster on-chip memory, called *shared memory*, is available to all threads in a thread block. The shared memory is typically used as a programmer-managed cache to avoid multiple trips to the global memory. Threads in a thread block can synchronize reads and writes to shared memory using built-in barrier primitives such as `__syncthreads()`. Moreover, each thread has access to hundreds of registers, and a thread of a warp can read the registers of other threads of the warp.

Several GPU programming models expose a variety of data types tailored to GPU hardware capabilities. In addition to standard scalar types like `float` and `int`, most GPU backends support low-precision types such as `fp16` (16-bit floating point) and `fp8` (8-bit floating point). Modern GPUs also include *tensor cores* which are specialized hardware units to accelerate low-precision operations and are invoked at the warp level instead of thread level. GPUs allow batched memory accesses using vectorized types, such as `float2`, `float4`, or `int4`, to improve memory bandwidth.

2.2 Matrix Multiplication

Matrix Multiplication (or simply MatMul) is a core computation in many critical and popular workloads. A standard BLAS library API takes two input matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ and produces an output matrix $C \in \mathbb{R}^{m \times n}$ such that $C = A \times B$. In this work, we assume that all matrices are stored in row-major order and that elements may be stored in either `fp32` or `fp16` precision.

A naive MatMul kernel launches one thread per output element $C[i][j]$, with each thread computing a single dot product between the i-th row of A and the j-th column of B. While functionally correct, this approach fails to exploit memory locality or shared computation, and thus performs suboptimally on modern GPUs. Several optimizations have been developed to obtain near-maximum throughput provided by GPUs [11, 26]. The standard optimization strategy is to apply *tiling*, where the computation is partitioned into smaller blocks (tiles), across all levels of compute and memory hierarchy of GPUs.

The first level of tiling is *Thread Block Tiling*, where each thread block computes a $t_m \times t_n$ block of C. Therefore, there are exactly $\frac{m}{t_m} \times \frac{n}{t_n}$ number of thread blocks. To fully utilize memory reuse and minimize global memory accesses, thread block tiling stores t_m rows of A and t_n columns of B in the shared memory. However, since shared memory size is limited, this tiling also defines a size, t_k, such that, each thread block loads t_k elements of both rows and columns into shared memory.

The next level of tiling, *Warp Tiling*, defines two more tile sizes, w_m and w_n, such that there are exactly $\frac{t_m}{w_m} \times \frac{t_n}{w_n}$ warps in a thread block. A warp processes w_m rows of A and w_n columns of B to produce a $w_m \times w_n$ tile of C, which is stored in thread-local registers. Warp tiling is the last level of tiling for tensor cores for low-precision floating-point operations.

The last level of tiling for CUDA cores, *Thread Tiling*, defines two more tile sizes, r_m and r_n, such that $\frac{w_m}{r_m} \times \frac{w_n}{r_n}$ is equal to the warp size of the GPU. Each thread loads r_m rows of A and r_n columns of B into the registers to produce a $r_m \times r_n$ block of C. Similar to TB-Tiling, this tiling minimizes shared memory accesses by loading input rows and columns to registers from the shared memory.

2.2.1 *Advanced Optimizations.*

In addition to choosing where data resides in the memory hierarchy, developers must also consider how memory is accessed between threads. For example, threads within a warp should ideally access contiguous memory locations to fully utilize the global memory bandwidth. Similarly, when accessing shared memory, all threads in a warp should access addresses that fall into different banks to avoid shared memory *bank conflicts*.

Moreover, it is also important to utilize both compute and memory resources of a GPU simultaneously. To achieve this, efficient MatMul kernels perform *pipelining* in a thread block by overlapping the global to shared memory copy of next t_k with the computation of current t_k.

These optimizations interact in complex ways, and the most performant configuration depends on the specific hardware, precision, and matrix dimensions. Thus, MatMul serves as an ideal case study for evaluating GPU performance engineering and the capabilities of AI-based optimization tools like Peak.

3 DESIGNING PEAK: AN EXTENSIBLE AND MODULAR FRAMEWORK FOR NATURAL LANGUAGE-DRIVEN GPU KERNEL OPTIMIZATION

We now describe the design of Peak, illustrated in Fig. 1, which is centered around lightweight, natural language transformation specifications executed by large language models (LLMs) . While LLMs offer flexibility and generality, the complexity of GPU kernel development necessitates a structured framework to ensure correctness, performance, and reuse. Our high-level design goals are as follows: (1) to provide a rigorous scaffolding that enables many simple, natural language transformation specifications, both general and specific; (2) to support basic functionality across a wide range of GPU devices and frameworks, which can then be fine-tuned; and (3) to enable extensibility, so that researchers and practitioners can incorporate external tools for validation, performance evaluation, and different techniques for kernel transformation, LLM-based or otherwise. We describe the core interfaces of Peak and then detail our implementation across three GPU backends: CUDA, HIP, and HLSL.

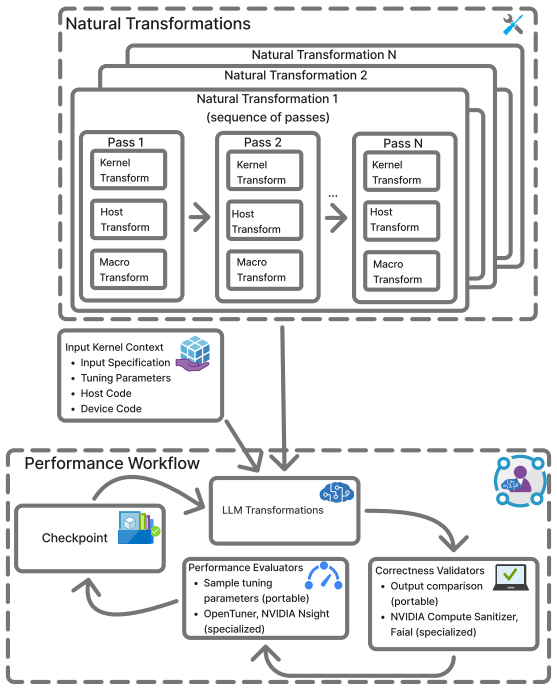

Fig. 1. Peak design overview

3.1 The Kernel Context

Intuitively, a *kernel context* encapsulates all information required to execute, evaluate, and refine a GPU kernel. This includes: (1) the kernel code and any associated device functions; (2) a host function that launches the kernel; (3) an input specification; and (4) a tuning parameter specification.

Input Specification and Performance Tuning Parameters. Input arguments are defined using a lightweight specification language. The language supports scalar and array types, including `f32`, `f16`, and `int`; arrays can further be annotated to be *output* arrays, which means their values can be used in the correctness validators. For scalars, all possible values must be enumerated explicitly or using range constructs, akin to list comprehensions in Python. For arrays, the specification must enumerate possible sizes (denoted with the field `.size`), while the values themselves may be populated using predefined initializations such as zeros, ones, or random values. Constraints may be applied to ensure validity across inputs. For example, if a kernel takes an array A representing an $m \times k$ matrix, the specification would enumerate valid values of m and k, as well as array shapes for A, while using a user-defined `valid_args` function to enforce that `A.size == m * k`.

While the current specification language is not designed to express complex structured inputs, e.g., graphs and sparse matrices, it is sufficient to capture a range of impactful GPU kernels, particularly those found in machine learning workloads. In these domains, performance-critical computations

are characterized by array shapes and dimensions, rather than specific data values in the arrays. Moreover, performance engineering often involves specializing GPU kernels for specific input shapes. As a result, the space of valid inputs is typically small and tractable, a common assumption in related work [7, 28]. Even production libraries like cuBLAS select different implementations internally depending on input sizes, motivating Peak's support for size specialization.

In addition to input arguments, a kernel context may include *performance tuning parameters*, represented as special string placeholders in the kernel or host code, with possible values stored in metadata. These parameters capture configuration choices that impact performance but not correctness, such as tile sizes and thread block dimensions. Like scalar inputs, tuning parameters are specified using enumerated sets or ranges. Explicitly incorporating tuning parameters into the abstraction is, to our knowledge, a novel feature of Peak, and one that mirrors common practice among human performance engineers. This design encourages transformations to introduce tunable degrees of freedom, which can then be explored automatically and efficiently. It reduces the burden on the LLM to produce highly specific values during code generation, while enabling flexible and generalizable transformations that defer fine-grained tuning to existing mature tools.

Executing a kernel context requires an instantiation of all kernel inputs and tuning parameters, which we call the *execution parameters*. Despite careful specification, not all execution parameters will be valid. For example, a particular tiling configuration may exceed the register or shared memory budget of a GPU. To handle such cases, we allow kernel contexts to return a special value when invalid configurations are detected at runtime; in turn, Peak can then simply prune the execution parameters from consideration.

Drivers. A kernel context encapsulates everything needed to generate a lightweight driver program. This driver is responsible for allocating, initializing, and copying inputs to the GPU. It then executes the kernel multiple times to measure runtime performance, reporting an average execution time. Optionally, it can capture and return output arrays, enabling correctness checks.

Limitations. While Peak currently operates on individual kernels, we anticipate that future extensions could support a wider context, e.g., considering multiple kernels and memory transfers between them. This would allow Peak to be used, e.g., for optimizations like kernel fusion, which combines multiple kernels into one. For now, Peak can be applied to kernels that have already undergone fusion via external tools, e.g., TVM [7].

3.2 Natural Language Transformations for GPU Kernels

The core innovation of Peak is its support for *natural language transformation specifications*, or simply *natural transformations*, which describe how to modify one kernel context into another. After a sequence of such transformations, the aim is for the resulting kernel to be significantly more performant than the input while remaining correct. Because transformations are written in natural language and validated by subsequent correctness checks, they can be authored rapidly, need not be fully formalized, and can incorporate both high-level intent and hardware-specific insights. This flexibility enables kernel-specific and device-specific transformations that would be difficult to express in traditional compiler optimization passes.

Despite the general capabilities of modern LLMs, effective transformations still benefit from thoughtful system design. In particular, we decompose transformation tasks both *spatially* and *temporally*, enabling simpler specifications, improved success rates (which we explore in Sec. 5.2), and more interpretable optimization sequences.

Spatial Decomposition. Peak partitions the kernel context into three regions: the *host code*, the *device code*, and *macro definitions*. Each transformation targets only one of these regions. Macros are

commonly used in performance-critical GPU code to avoid function call overhead, without relying on compilers to do inlining. Constraining each transformation to a single region reduces the risk of unintended code edits outside the transformation's intended scope, which we observed during early prototyping. For typical GPU kernels, which are short but intricate, this three-way partitioning has proven sufficient. For transforming larger code bases, e.g., as done in Github Copilot [13] and Amazon Q Developer [2], more complicated decompositions will be needed.

Temporal Decomposition. Transformations may also be expressed as a sequence of simpler *passes*. Each pass produces a new kernel context, however, it may be incomplete or temporarily incorrect, but it is structured to enable the next pass in the sequence. This decomposition allows complex or multi-part transformations to be specified as smaller, more manageable steps, improving reliability and more precise debugging capabilities.

Manual and Reusable Code. In addition to LLM-guided edits, Peak allows transformations to include manual code insertions, such as auxiliary macros or utility functions that do not depend on the surrounding kernel. For example, a transformation may reference a macro that performs efficient global-to-shared memory transfers. Rather than prompting an LLM to generate such helpers, which can be unreliable and inefficient, Peak provides a small library of reusable, hand-written utilities that can be imported on demand. One could imagine incorporating more library frameworks into Peak transformations, e.g., CUTLASS [26], CUTE [25], or Thunder Kittens [32].

3.3 Correctness Validators and Performance Evalulators

While Peak is primarily driven by LLM-guided code transformations, its supporting infrastructure provides the scaffolding necessary to ensure correctness and fine-tune performance. This is organized into two extensible interfaces: *correctness validators* and *performance evaluators*. Each contains a general and portable baseline task, with an extensible interface which can incorporate new tools, potentially specialized for a backend, as this field evolves.

Each task takes a kernel context and a parameter that specifies how many execution parameters to sample during evaluation. Exhaustively enumerating all execution parameters is often infeasible, especially when expensive dynamic analysis tools are used, so sampling provides a practical trade-off. We explore this tradeoff more rigorously in Sec. 5.3.

Correctness Validators. The correctness validator is responsible for checking the correctness and safety of a given kernel context, which is critical when kernels have been created by LLMs. The portable baseline task simply compares the output arrays of a kernel context against those produced by a reference implementation, although there needs to be some tolerance theshold to account for small floating point discrepancies.

While output comparison is often sufficient, more subtle or nondeterministic bugs, such as data races or memory safety violations, may go undetected without finer-grained analysis. These issues are increasingly important, especially given increasing concerns about GPU security vulnerabilities [15, 31] and rare, non-deterministic behaviors that can lead to bugs [1, 30]. To this end, Peak integrates specialized dynamic analysis tools, such as NVIDIA's Compute Sanitizer, to detect memory errors and race conditions. These dynamic analysis tools typically require backend-specific integration, and have high runtime costs but can greatly enhance the confidence in the optimization process. Similarly, Peak also integrates static data-race detection tools, such as Faial [9], which, when applicable, provide more coverage (over execution parameters) and execute faster, but might not handle the latest programming features, such as tensor core APIs. Overall, there does not yet seem to exist tool or methods that provide truly formal and rigorous analysis of GPU kernels, and thus, PEAK is designed so that new approaches can be easily incorporated as they are developed.

Performance Evaluators. The performance evaluator identifies performant tuning parameters, and optionally collects auxiliary performance metrics to guide further optimization. The portable baseline task enumerates all valid combinations of execution parameters, executes the kernel with each configuration, and records the runtime to identify the best-performing setup.

To scale beyond exhaustive search, the engine integrates with autotuning frameworks, such as OpenTuner [4]. The profiling engine shares the same task interface model as validation: it accepts a kernel context and a sampling budget, and returns a performance summary over the sampled configurations. In addition to runtime measurements, profiling tasks collect backend-specific performance counters and metrics. For example, Peak integrates NVIDIA's Nsight profiler to extract hardware metrics such as memory bandwidth utilization or shared memory bank conflict rates. These metrics can inform downstream decisions about which transformations to apply next.

3.4 Performance Workflows

Having described the core components of Peak, we now describe how these elements interact within a complete *performance workflow*. Peak supports iterative performance engineering, where a human user or AI agent applies transformations, validates correctness, evaluates performance, and records progress. For human interfaces, we expose a direct API, and also provide an MCP interface so that the tools can be engaged with through natural language.

Seeding the Workflow. The workflow begins with a manually specified *initial kernel context*. In practice, these initial kernel contexts are often easy to write and typically require fewer than 10 lines of code. However, while manual authoring is sufficient for many use cases, kernel contexts can also be generated automatically. For instance, related systems, such as KernelBench [27] can be used to generate kernels, and then Peak can be used to further optimize; this would enable workflows to start from either human-written or AI-generated code.

Validation. Validation in Peak is grounded by the behavior of the initial kernel context, which is assumed to represent correct functional behavior given that it should be a simple implementation. Upon initialization, this kernel is executed across the input configurations, and its outputs are recorded as a reference. Subsequent kernel contexts, e.g., produced by natural transformations, are validated by comparing their outputs to these references. This validation may be augmented with more sophisticated dynamic analysis tools, but even simple output checks provide a portable and backend-agnostic foundation for correctness checking.

Checkpointing. At any point during optimization, the system can create a *checkpoint*, which captures the current kernel context along with any metadata produced by validation or profiling. These checkpoints allow the system to track optimization progress, compare performance across versions, and backtrack after unproductive transformations. Checkpoints also enhance reproducibility and traceability, making it easier to share experiments or resume long-running optimization sessions.

This flexible design enables Peak to support a wide range of optimization workflows, from fully automated pipelines to exploratory, human-driven sessions, and positions the framework as both a practical assistant and a research platform for GPU performance engineering.

4 OPTIMIZING MATRIX MULTIPLICATION ACROSS GPU BACKENDS

We now show how Peak can be used to optimize MatMul kernels for a variety of devices and GPU programming backends. As discussed in Sec. 2.2, MatMul is a performance-critical operation that is also notoriously difficult to optimize.

Table 2. Nine out of 16 of our Peak's natural transformations for MatMul

Name	Description	Passes / LLM Calls	Tuning Params
Refactor	Refactors primitives and complex accesses to macros	2 / 3	None
TB-Tiling	Performs thread block tiling	6 / 9	`TILE_K_SIZE`
Warp-Tiling	Performs warp tiling	1 / 1	`WARP_X_DIM`
Thread-Tiling	Performs thread tiling	3 / 3	`TRD_X_DIM` `TRD_Y_DIM`
Tensor-Core	Utilizes tensor cores	1 / 1	None
Split-K	Splits the K dimension across thread blocks	1 / 3	`K_SPLITS`
Transpose	Transposes memory when tiling	1 / 1	None
Offset	Offsets shared memory tiles to avoid bank conflicts	1 / 2	`OFFSET_AMOUNT`
Pipelining	Pipelines loading and computing thread block tiles	1 / 2	`NUM_STAGES`

4.1 Matrix Multiplication Transformations

We begin by describing the natural transformations we provide for MatMul based on the strategies described at a high level in Sec. 2.2. These transformations were developed using a combination of sources, including published documentation (e.g., [6, 11]), reference implementations (e.g., NVIDIA CUTLASS [26]), and discussions with experienced GPU performance engineers. In total, we implemented 16 distinct transformations. For brevity, we summarize 9 of them in Tab. 2.

Each transformation is written in a natural language specification. Many of these transformations are specific to MatMul, for example, explicitly referring to the "K dimension". We do not claim that these transformations are universally applicable across all GPU workloads. However, similar patterns often arise in other compute-intensive kernels, such as convolution and attention, suggesting that many of these transformations could be easily adapted. Furthermore, many are often reusable across GPU backends (CUDA, HIP, and HLSL), as well as data types (e.g., F32 and F16) which we believe represents an important and underexplored axis of portability in performance engineering.

As described in Sec. 3.1, each transformation may consist of a sequence of smaller *passes*, and each pass can invoke the LLM up to three times, once for each code region: host code, device code, and macro definitions. This decomposition enables complex optimizations to be applied incrementally without overwhelming the LLM. Some transformations simply prepare the code for later transformations, for example, Refactor simply moves primitives (like thread IDs) and array accesses into macros, which can then be targetted by later passes. Some passes are relatively complex, like TB-Tiling, requiring 6 passes. Our pilot experiments found that LLMs struggled with performing this transformation all at once, and breaking it down into simpler steps was more reliable. Indeed, prior work on compilers that perform tiling shows how transformations like this can be broken down into simpler steps [28]. We found LLMs were able to perform other types of tiling (i.e., across warps and threads) more reliably, and thus, needed less decomposition.

4.2 GPU Backend Implementations

A key strength of Peak is that its core design is entirely backend-agnostic: it contains no framework-specific grammars, parsers, and very little hardcoded tooling. This makes it straightforward to instantiate Peak for different GPU programming frameworks, which has often posed considerable difficulty for other frameworks, and they provide limited portability. To demonstrate this, we implement Peak for three widely used backends: CUDA, HIP, and HLSL.

The most backend-specific component is the driver logic for each kernel context. While the device kernel specifications is similar across backends, the host-side code used to manage memory and launch kernels varies. For example, HIP and CUDA require different API calls for memory

management and synchronization, while HLSL, being rooted in graphics APIs, requires significantly more boilerplate, including explicit creation of device handles, command queues, and memory views. Similarly, there is a small amount of hard-coded code in some transformations, mostly related to efficiently loading memory. We were able port this code from CUDA to HLSL straightforwardly.

Beyond this, only a small number of transformations required backend-specific adaptation. Many differences (such as how thread identifiers are provided) were handled by factoring out backend-specific syntax and adding translation tables directly into the transformation prompts. For example, thread indexing syntax differs across CUDA (`threadIdx.x`), HIP (`hipThreadIdx_x`), and HLSL (`SV_DispatchThreadID`). A small number of transformations required new specifications; for example, in the case of tensor core integration, the HIP backend required a dedicated transformation due to API differences in naming and supported tile shapes. However, this new transformation was quickly adapted from the CUDA version taking less than 30 minutes that were largely spent in consulting the HIP documentation.

As noted in Sec. 3.3, Peak contains portable validation and performance tasks, such as output comparison and tuning parameter exploration. Given the accessible tooling for CUDA, we were able to provide some specialized tooling. We incorporated OpenTuner for CUDA and HIP, but were unable to use this tool for HLSL on Windows; instead since only few transformations were applied, we could simply enumerate and search over all parameters.

4.3 Performance Workflows

Table 3. GPU devices used in our evaluation and their documented peak throughput (in TFLOPS), separated into `fp32` / `fp16` when available.

Device	Driver	Compiler	TFLOPS
N. A6000	575.57.08	`nvcc 12.2`	38.7 / 155
A. MI200	ROCm 6.3.2	`hipcc 6.3`	47.9 / 383
Q. X1-85	31.0.112.0	DXC 6.6	4.6

We now describe the Peak performance workflow used to optimize variants of MatMul across different devices and GPU programming frameworks. The input kernel context is a simple 6-line GPU implementation in which each thread computes a single element of the output matrix $C = A \times B$. The kernel accepts four arguments: three arrays (`A`, `B`, and `C`), and an integer `n` representing the matrix dimension. All matrices are square of size $n \times n$, and array `C` is designated as the output; all of them are initialized with random values. For this case study, we evaluate two matrix sizes: $n = 2048$ and $n = 4096$ referred to as 2K and 4K, respectively. We consider two data types: `fp32` and `fp16`. Our initial kernel context includes tuning parameters that control how the global thread grid is partitioned into thread blocks. We restrict the thread block dimensions to powers of two, constrained by backend-specific limits (e.g., 1024 for CUDA and 256 for HLSL).

We study the optimization trajectories of Peak across three devices, as listed in Table 3. For each device and precision variant, we combined documentation and discussions with GPU performance engineers to construct a transformation sequence designed to yield high-performance kernels. These sequences were assembled from Peak's natural transformations, many of which are given in Tab. 2, and the complete optimization paths for each configuration are summarized in Tab. 4.

While transformation sequences were constructed manually for this study, they could also be discovered via alternative methods, such as automated search or LLM-driven agent-based exploration. However, we believe that this semi-autonomous setting, with a human guiding and interpreting the optimization trajectory, provides a compelling and realistic use case. To support future work in more autonomous optimization, we evaluate how performance evolves throughout these transformation workflows in Sec. 5.1.

Table 4. Transformation sequences per device and precision. The final performance is given in Tab. 1.

Device	Precision	Transformation Sequence	# Transforms
A6000	fp32	Refactor → TB-Tiling → Thread-Tiling → Thread-Cache → Transpose → Thread-Chunk → Split-K → Pipelining → Register-Staging → Offset → Block-Swizzle	11
A6000	fp16	Refactor → TB-Tiling → Warp-Tiling → Tensor-Core → Tensor-Tiling → Pipelining → Register-Staging → Offset → Block-Swizzle	9
MI200	fp32	Refactor → TB-Tiling → Warp-Tiling → Tensor-Core → Tensor-Tiling	5
MI200	fp16	Refactor → TB-Tiling → Warp-Tiling → Tensor-Core → Tensor-Tiling → Offset → Block-Swizzle → Pipelining	8
X1-85	both	Refactor → TB-Tiling → Thread-Tiling	3

While many optimization sequences begin similarly, e.g., refactoring and performing thread block tiling, they diverge depending on architectural features. For instance, on NVIDIA GPUs, the `fp16` path transitions to using tensor cores while `fp32` uses thread tiling but both variants then converge again at pipelining. On AMD MI200, both `fp32` and `fp16` kernels utilize tensor cores; however, the effective tuning parameters differ between the two, as found by our performance evaluators. For example, the best-performing warp tile size for `fp32` was `(4,2)`, while `fp16` was not found to utilize tiles. Unlike on NVIDIA devices, we did not apply pipelining transformations for AMD on `fp32`, as the MI200 offers less shared memory, which is a critical resource for this optimization. For the HLSL backend on the Qualcomm Adreno GPU, only a few transformations were needed to reach the device's reported peak GFLOPS. Given this, and our focus on other devices and components, we did not explore deeper optimization sequences for this backend.

Table 5. End-to-end time for the A6000 performance workflow. Times are reported as in seconds, as *raw (percent)*. In both cases, the total time is less than 6 hours, which can be run overnight.

Type	Validation	Transforms	Performance	Total
fp32	825 (3.91%)	1285 (6.09%)	18995 (89.99%)	21105
fp16	1124 (1.55%)	257 (6.77%)	15226 (91.68%)	16607

For the sequences reported in Tab. 4, the LLM was able to correctly apply all the transformations in the sequence without human intervention using the o4-mini LLM; however we show in Sec. 5.2 that other models are able to successfully (and reliably) apply our transformations as well.

The end-to-end time to perform all transformations varies greatly on how Peak is invoked. For example, how many execution parameters are sampled at each transformation to validate correctness and gather profiling information, which ultimately may just be collected at the end of a performance workflow. Similarly, it is difficult to provide reliable measurements from LLM calls, as we have found their latency varies significantly seemingly randomly (likely based on cloud availability). Thus, with these combined considerations, we provide a summary of the end-to-end execution time of fp16 and fp32 on the A6000 in Tab. 5 (Our other performance workflows were executed in a more distributed and exploratory manner for the results in Sec. 5). For the NVIDIA workflows, we sampled 16 execution parameters after each transformation to validate, using both output comparison and NVIDIA's compute sanitizer. After every transformation, we used the performance evaluator to enumerate all performance parameters, and prune all except for the top 128 to utilize in the next transformation. We utilized 128 CPU cores to compile kernels in parallel and 4 GPUs on the machine to execute performance and correctness tasks in parallel. We note that the tuning parameter exploration is the vast majority of the runtime. If future approaches, e.g., symbolic performance estimations, could accurately prune this search space effectively it could improve Peak significantly.

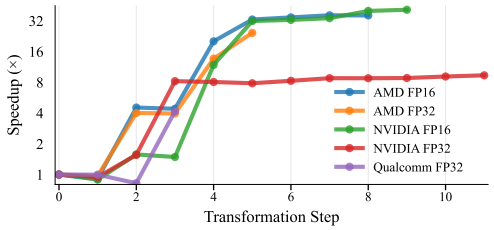

Fig. 2. Speedup of each transformation over the *naive input kernel context.*

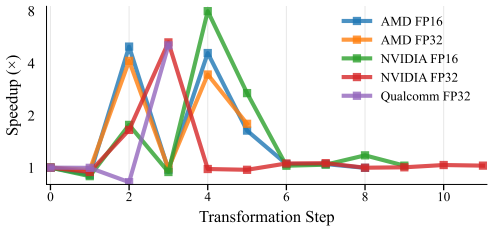

Fig. 3. Speedup of each transformation over the *previous transformation.*

5 RESEARCH QUESTIONS: EXPLORING LLMS FOR GPU KERNEL OPTIMIZATION

Unlike monolithic or opaque approaches that treat optimization as a black box, PEAK's structured workflow provides visibility into each transformation step, validation outcome, and performance change. This transparency enables systematic exploration of LLM capabilities and limitations in this domain. We investigated three key research questions that illuminate both the strengths and current limitations of LLM-assisted GPU optimization.

5.1 RQ1: How Does Performance Evolve During Iterative Optimization Sequences?

Understanding the performance trajectory during optimization is crucial for developing effective automated strategies and managing expectations about optimization potential.

Methodology. We analyze the performance evolution across our five device-precision combinations (AMD FP16/FP32, NVIDIA FP16/FP32, Qualcomm FP32) at size 2K through two complementary views: cumulative speedup relative to the baseline kernel (Fig. 2) and step-wise improvement between consecutive transformations (Fig. 3). The executed steps for each correspond to the sequences given in Tab. 4 and has been tuned to find efficient performance parameters. Note that the different instances have different lengths because they have a different number of transformations.

Key Observations. Fig. 2 reveals that optimization follows distinct phases rather than smooth, incremental progress. Most configurations exhibit a characteristic pattern: minimal improvement in early steps (transformations 0-2), followed by rapid acceleration during critical steps (transformations 3-4), and finally plateau behavior in later stages (transformations 5-10), where small improvements occur over many transformations. For example on NVIDIA `fp32`, while the red line appears flat starting at step 3, in reality, this long journey provides a 10% performance improvement. This follows the common sentiment that most effort is spent on the last percentages.

The non-linear performance evolution also suggests that early transformations unlock the potential for subsequent optimizations. That is, the very early transformations, such as refactoring, establishes code structure, but provides limited immediate performance benefit. The acceleration phase (with the highest jumps) appears to coincide with memory hierarchy optimizations and specialized accelerator usage (e.g., shared memory usage, register usage, tensor cores), while the final transformations are more about fine-tuning access patterns.

The magnitude of final speedups varies across device-precision combinations. NVIDIA FP16 and AMD FP16 achieve the highest speedups (35-40×), while NVIDIA FP32 shows more modest gains (8-10×). Qualcomm FP32 demonstrates the shortest optimization trajectory, reaching its performance ceiling after only three transformations. The precision-dependent speedup patterns reflect the underlying hardware capabilities and optimization opportunities. FP16 configurations

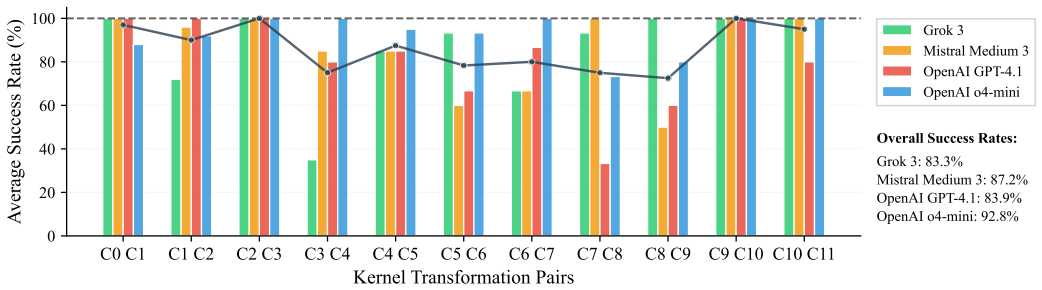

Fig. 4. Success rate of each model across transformation pairs. The transformations are done for our three devices - NVIDIA A6000, AMD MI200, and Qualcomm. The figure also presents the overall success rate for each model and also the average success rate for each transformation across all models.

achieve higher speedups partly due to tensor core utilization and increased memory bandwidth efficiency, while FP32 optimizations are more limited.

The early saturation observed in Qualcomm FP32 suggests that this platform reaches peak performance with fewer transformations, possibly due to architectural differences or more limited optimization opportunities compared to data center GPUs. This is likely due to much less parallelism and absence of dedicated GPU memory in the SoC.

Fig. 3 provides insight into which individual transformations drive the most significant improvements. The highest speedups occur at transformation steps 2-4 across most configurations, with individual transformations providing up to 8× improvement over the previous one.

5.2 RQ2: How Do Different LLMs Perform at GPU Kernel Transformations?

The choice of LLM for executing natural transformations impacts the success rate of individual transformations and, thus the overall optimization workflow reliability. Understanding which models excel at GPU kernel transformations, and which transformation types pose the greatest challenges, can guide practitioners in choosing between different LLM providers.

Methodology. We evaluate four contemporary LLMs across our complete set of transformation pairs (C0 → C1 through C10 → C11 given in Tab. 4); we run each transformation five times across each model and measure success rates. Success is defined as generating syntactically and functionally correct code, as checked by our validation engine (by sampling 16 execution parameters and performing output comparisons). Note that transformations may not be same across device-precision combinations, as noted in Tab. 4, instead the point is to evaluate how well LLMs perform across a performance workflow.

Key Observations. Our results are summarized in Fig. 4. At a high-level, we see that *all* models are able to successfully apply all transformation steps, even if they do not have 100% success rate. This means, any model can successfully run Peak, if given enough attempts at each transformation. We see this as a success of our transformation decomposition into a series of simpler passes.

Despite this, there are still performance differences between models, with overall success rates ranging from 83.3% (Grok 3) to 92.8% (OpenAI o4-mini). The two reasoning models (OpenAI o4-mini and mistral medium 3) are the top two performers, which highlights that reasoning models perform slightly better in this domain. We note that we utilized o4-mini in many of our pilot experiments, and thus, we may have inadvertently biased our prompts to work better with that model.

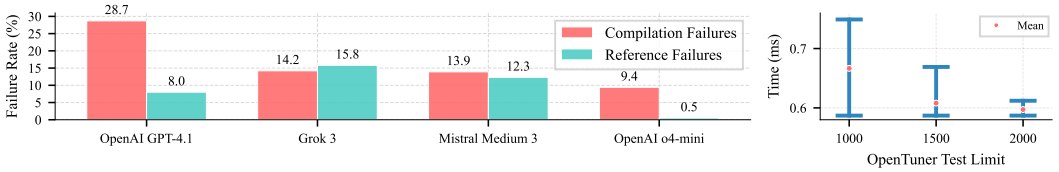

Fig. 5. Left: average failure rate for each model across all transformation pairs. Compile errors are counted as Compilation failures whereas kernels that compile but do not produce the correct output are counted as Reference failures. The transformations were done for our three devices - NVIDIA A6000, AMD MI200, and Qualcomm. Right: Time range achieved by OpenTuner for the final Matrix Multiplication kernel (FP32) on AMD MI200. Higher test limit means that OpenTuner was run for more iterations.

The transformation-specific analysis reveals that certain optimizations pose challenges while others are consistently well-handled across models. The success rates also seem to go in phases, with early and late transformations being performed well, but difficulty in the middle transformations, i.e., C3 → C4 up to C8 → C9. We believe this highlights the difficulties of the tasks: early transformations (thread block tiling, thread tiling, tensor cores) modify the code extensively, but they are typically well-understood (i.e., there are many examples) and follow regular patterns. Middle transformations tend to modify the code extensively, have fewer examples, and be less regular. For example, we found pipelining, transpose, and register staging (where global memory values are staged in registers before being stored to shared memory) to be particularly difficult for LLMs. Then later passes, while conceptually more complex, are highly targeted, and modify the code relatively little compared to earlier passes.

These error rates could likely be improved in several ways: (1) by fine-tuning the prompts to each model, or (2) by breaking down the transformations into even simpler passes. Given that Peak also supports human interaction, it is possible for the LLM to implement *most* of a transformation, and for a human to finish editing the kernel context, and then create a checkpoint, and move on to later transformations through the LLM, which tend to be more reliably executed.

5.3 RQ3: To What Degree Does Correctness and Performance Need to be Evaluated?

Understanding the depth of evaluation (both for correctness and performance) required for different LLMs and hardware configurations is important for designing efficient optimization workflows. While comprehensive evaluation provides confidence, excessive evaluation can become computationally expensive (even in our example workflow shown in Tab. 5, performance evaluation consumes 90% of the time). The challenge lies in determining the optimal balance between thoroughness and computational efficiency for different system configurations.

We analyze two dimensions of evaluation requirements: (1) the types and frequency of errors produced by different LLMs, categorizing failures into compilation errors and reference failures, i.e., whether the output array values match the reference, and (2) the performance evaluation depth required, examining how different OpenTuner test limits affect performance discovery across hardware backends.

Key Observations on Error Patterns. The left graph of Fig. 5 reveals variation in error types across LLMs. OpenAI GPT-4.1 exhibits the highest overall failure rate (28.7%), with compilation failures dominating over reference failures. In contrast, OpenAI o4-mini shows the lowest failure rate (9.9%), with compilation failures representing the majority of its errors (9.4% vs 0.5% reference failures). Grok 3 and Mistral Medium 3 show more balanced error distributions, with reference failures comprising a substantial portion of their total failures.

The dominance of compilation failures across most models suggests that syntactic correctness remains a primary challenge for LLMs in GPU kernel optimization. However, we see this as an opportunity, because typically compiler errors produce actionable error messages, which LLMs can use to recover. However, reference failures are far more worrisome. They take longer to catch, as the code must be compiled and executed. It is also unclear if the errors are deterministic (e.g., caused by a data race) or if they occur for all execution parameters, or if they require extensive searching. Finally, it is difficult to get actionable feedback that can be fed back into the LLM (however, this would be a promising direction for future work). Thus, we see strong potential in using models that have low reference failures; in our case, we found that o4-mini has significantly fewer reference errors (0.5%).

Key Observations on Performance Evaluation. The right graph of Fig. 5 shows the varation of runtimes found by OpenTuner when run on the AMD GPU for the final kernel of `fp32` on size 2K. We run OpenTuner 5 times for the three different iteration counts shown on the x axis. The graph shows the distribution of the times found across these 5 executions, showing the maximum, minimum, and mean. We see that extending test limits from 1000 to 2000 iterations (from roughly 10 to 20 minutes) shows continued performance improvements, with mean execution time decreasing and variance reducing substantially. This suggests that AMD hardware benefits from more extensive parameter exploration, likely due to complex interactions between tile sizes, memory hierarchies, and compute unit utilization. Conversely, analysis of NVIDIA fp32 configurations (not shown) indicated faster convergence, with results often converging in just several hundred iterations. This may be due to a more regular and predictable performance profile on NVIDIA GPUs, as OpenTuner implements advanced searching mechanisms that aim to exploit patterns. Regardless, this disparity highlights the hardware-specific nature of performance optimization and the need for adaptive evaluation strategies.

5.4 Putting it all Together

PEAK's design enabled exploring these three research questions in a precise and fine-grained manner, which provides concrete takeaways for any AI assistant for performance optimization. In particular, we highlight the following insights:

- Given the insights into RQ1, the performance differences between transformations may not always be smooth or even monotonically increasing. Thus, when searching for optimized kernels in a black box manner, similar to [29], there may need to be more sophisticated evaluation measures, or the ability to explore deeply without expecting immediate benefits.
- Given the insights into RQ2, models tend to perform well at early and late stages in optimization workflows but struggle in the middle. Thus, explorations should spend extra time in these stages; potentially providing finer-grained prompts, or enabling extra retries.
- Given the insights into RQ3, models should be evaluated on the types of errors they produce and validation tasks should be adjusted accordingly. Regardless, there should be a feedback mechanism to repair compiler errors. For performance, different devices should be calibrated in order to determine how much exploration is needed. This is especially important as this evaluation time can be very costly, as shown in Tab. 5.

6 RELATED WORK

Optimizing GPU Kernels. Several works have aimed to improve the developer experience in writing optimized GPU kernels. There are several higher-level domain specific languages (DSLs) for optimizing image processing and machine learning applications on GPUs, like Halide [28], Exo [18], PolyMage [19], and CoCoNet [20]. These DSLs express a computation on a tensor as

a stage, then enable a set of transformations on stages like fusion, reorder, overlapping, and generate optimized code for various GPU backends. REPTILE [37] and TACO [21] similarly provide scheduling abstractions for tiling strategies and sparse tensor operations respectively. Triton [38] and TileLang [39] are higher-level kernel languages that abstracts intricacies of the hardware and its low-level kernel language. Both of these languages provide a tile based abstraction, where the programmer writes a vectorized code for processing a tile and the tile is mapped to one or more thread blocks at runtime. Similarly, FlexAttention [22] is a higher level abstraction for generating various kinds of attention kernels. Lower-level intermediate representations languages, like TVM [7], TensorIR [12], and Graphene [16], apply several transformations to generate efficient GPU kernels. Unlike DSLs, ThunderKittens [32] is an extensive library that can be used to write CUDA kernels using warp-level primitives like tensor cores and efficiently manage memory transfers and pipelining. In contrast, Peak enables quickly writing efficient GPU kernels using existing kernel languages. Moreover, we believe Peak is extensible enough to utilize the transformations of the above compilers as natural language text to generate other optimized GPU kernels. We leave this study for future work.

AI Assisted Optimizations. New advances in AI have enabled LLM-based approaches for code optimizations. For example, LLM Compiler [10] trains on large set of LLVM IR and assembly to emulate optimizing compiler passes, achieving nearly the same performance. LLM-Vectorizer [36] applies an LLM to loop vectorization on LLVM IR and integrates the formal verifier Alive2 [23] to validate the transformed code. While these works show that LLMs can implement optimizations, they are either one-shot systems or implement opaque search strategies. They lack support for interpretable iterative refinement, rollback, or human-in-the-loop guidance, which Peak provides. Other approaches were surveyed in [14], showing trends across prompt-based, fine-tuned, and reinforcement-driven methods, while noting challenges such as limited real-world evaluation, lack of correctness guarantees, and difficulty integrating into production toolchains. This emphasizes the difficulty of utilizing LLMs in this domain and highlights the importance of Peak, as it can explore LLM performance in a fine-grained manner, as we do in Sec. 5.

AI Optimizations for GPU Kernels. As mentioned throughout, there are several works that utilize LLMs to optimize GPU kernels. KernelBench [27] provides an extensive corpus of AI tasks for GPU programs, from individual kernels, to entire DNNs. Their current leaderboard shows that LLMs can produce functional kernels in many cases, but complex kernels like MatMul, only achieve a fraction of the performance as vendor libraries. The GPU Kernel Scientist [3] and the CUDA Engineer [29] performs iterative search over kernel transformations, where an LLM repeatedly mutates CUDA or HIP kernels using runtime performance feedback. While this approach is also iterative, there is little insight into the search, and as we show in Sec. 5.1 the performance workflow may contain extended paths where there is little performance gain, and even performance loss occasionally. As a result, the current results for the AI CUDA Engineer shows only around 45% performance of vendor libraries for MatMul, whereas Peak, requiring some manual work in writing prompts, can achieve over 90% in many cases and will only improve as it's curated knowledge base grows. Recent concurrent work has explored similar directions. Hong et al. [17] investigate LLM-aided compilation for tensor accelerators. Zhou et al. [41] propose QiMeng-GEMM, which is very similar to PEAK, in which specialized prompts are used to optimize GEMM kernels, similar to the natural transformations in PEAK. While similar in spirit, PEAK provides additional experiments showing the fine-grained capabilities of LLMs in this domain. Furthermore, PEAK explores more diverse input and backend domains, considering FP16 (requiring tensor cores transformations) and shows that prompts can be generalized across different vendors, including AMD and Qualcomm.

We note that this is a fast moving area, and as mentioned in the introduction, there are now several start ups also making significant progress in this area, e.g., see [33–35]. We believe that PEAK offers an attractive design point in being interpretable, i.e., allowing inspection and development at every stage of optimization, and accessible, i.e., not requiring a complex RL infrastructure, while also producing competitive performance for important kernels.

7 CONCLUSION

Peak utilizes advances in AI technology to transform GPU kernel performance engineering from an ad hoc, expert-driven task into a transparent and reproducible process that combines human expertise with AI-driven automation. Rather than attempting monolithic and opaque end-to-end generation, it applies iterative natural-language transformations, with the ability to check correctness and fine tune performance in a portable, yet extensible, manner. Our evaluation on matrix multiplication shows that Peak outperforms other work in the area, and even achieves competitive performance with vendor-tuned libraries on NVIDIA and AMD GPUs. Furthermore, the curated knowledge base developed for one GPU can transfer to entirely different backends, creating high performance kernels where libraries might not even exist, as we illustrate for HLSL.

REFERENCES

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/2694344.2694391

[2] Amazon. [n. d.]. Amazon Q Developer: Your AI code assistant. https://aws.amazon.com/q/developer/build/. Accessed: 2025-09-10.

[3] Martin Andrews and Sam Witteveen. 2025. GPU Kernel Scientist: An LLM-Driven Framework for Iterative Kernel Optimization. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*. https://openreview.net/forum?id=K4XSvet59a

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[5] Anthropic. Apr 2025. Anthropic Economic Index: AI's impact on software development. https://www.anthropic.com/research/impact-software-development

[6] Simon Boehm. 2022. How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog. https://siboehm.com/articles/22/CUDA-MMM

[7] 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. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*.

[8] Terry Chen, Bing Xu, and Kirthi Devleker. 2025. Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling. https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/

[9] Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. 2024. Memory access protocols: certified data-race freedom for GPU kernels. *Formal Methods in System Design* 63, 1 (October 2024), 134–171. https://doi.org/10.1007/s10703-023-00415-0

[10] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) *(CC '25)*. Association for Computing Machinery, New York, NY, USA, 141–153. https://doi.org/10.1145/3708493.3712691

[11] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 42–51. https://doi.org/10.1145/3211346.3211354

[12] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization.

In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3575693.3576933

[13] GitHub. [n. d.]. GitHub Copilot: Your AI Pair Programmer. https://github.com/features/copilot. Accessed: 2025-09-10.

[14] Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. 2025. Language Models for Code Optimization: Survey, Challenges and Future Directions. *arXiv preprint arXiv:2501.01277v2* (2025). https://arxiv.org/abs/2501.01277

[15] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU memory exploitation for fun and profit. In *Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24)*. USENIX Association, USA.

[16] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3582016.3582018

[17] Charles Hong, Sahil Bhatia, Altan Haan, Shengjun Kris Dong, Dima Nikiforov, Alvin Cheung, and Yakun Sophia Shao. 2024. LLM-Aided Compilation for Tensor Accelerators. In *2024 IEEE LLM Aided Design Workshop (LAD)*. 1–14. https://doi.org/10.1109/LAD62341.2024.10691748

[18] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[19] Abhinav Jangda and Arjun Guha. 2020. Model-Based Warp Overlapped Tiling for Image Processing Programs on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/3410463.3414649

[20] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 402–416. https://doi.org/10.1145/3503222.3507778

[21] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1 (October 2017). Issue OOPSLA.

[22] Junyan Li, Delin Chen, Tianle Cai, Peihao Chen, Yining Hong, Zhenfang Chen, Yikang Shen, and Chuang Gan. 2025. FlexAttention for Efficient High-Resolution Vision-Language Models. In *Computer Vision – ECCV 2024*, Aleš Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (Eds.). Springer Nature Switzerland, Cham, 286–302.

[23] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[24] miru_why (@miru_why). 2025. Tweet: "@niklassheth @ronusedh @IntologyAI their 'superhuman' AI cleverly assigned all the work to non-default streams…". https://x.com/miru_why/status/1991773868806361138

[25] NVIDIA. [n. d.]. Getting Started With CuTe. https://docs.nvidia.com/cutlass/media/docs/cpp/cute/00_quickstart.html. Accessed: 2025-09-10.

[26] NVIDIA. 2025. NVIDIA CUTLASS. https://github.com/NVIDIA/cutlass

[27] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? arXiv:2502.10517 [cs.LG] https://arxiv.org/abs/2502.10517

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM. https://doi.org/10.1145/2491956.2462176

[29] SakanaAI. 2025. AI-CUDA-Engineer-Archive. https://huggingface.co/datasets/SakanaAI/AI-CUDA-Engineer-Archive

[30] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Programming Language Design and Implementation PLDI*. ACM. https://doi.org/10.1145/2908080.2908114

[31] Tyler Sorensen and Heidy Khlaaf. 2024. LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory. arXiv:2401.16603 [cs.CR] https://arxiv.org/abs/2401.16603

[32] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] https://arxiv.org/abs/2410.20399

[33] Standard Kernel Co. 2025. Standard Kernel — Building AI Infrastructure with AI. urlhttps://standardkernel.com/. Startup focused on AI-generated kernel optimization for hardware accelerators; accessed 21 Dec. 2025.

[34] Grace Stanley. 2025. Mako, a Faculty-Led Startup Based at Cornell Tech, Raises $8.5 Million. Tech.Cornell.edu — News. https://tech.cornell.edu/news/mako/ Accessed: 9-10-2025.

[35] Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li, and Chris Shum. 2025. CUDA-L2: Surpassing cuBLAS Performance for Matrix Multiplication through Reinforcement Learning. arXiv:2512.02551 [cs.LG] https://arxiv.org/abs/2512.02551

[36] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. 2025. LLM-Vectorizer: LLM-Based Verified Loop Vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 137–149. https://doi.org/10.1145/3696443.3708929

[37] Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. 2025. REPTILE: Performant Tiling of Recurrences. 9, OOPSLA2, Article 296 (Oct. 2025), 27 pages. https://doi.org/10.1145/3763074

[38] Philippe Tillet, H. T. Kung, and David Cox. 2021. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. https://openai.com/research/triton

[39] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, et al. 2025. TileLang: A Composable Tiled Programming Model for AI Systems. *arXiv preprint arXiv:2504.17577* (2025).

[40] Kyle Wiggers. 2025. Sakana Walks Back Claims That Its AI Can Dramatically Speed Up Model Training. https://techcrunch.com/2025/02/21/sakana-walks-back-claims-that-its-ai-can-dramatically-speed-up-model-training/. TechCrunch, Accessed: 2025-09-10.

[41] Qirui Zhou, Yuanbo Wen, Ruizhi Chen, Ke Gao, Weiqiang Xiong, Ling Li, Qi Guo, Yanjun Wu, and Yunji Chen. 2025. QiMeng-GEMM: Automatically Generating High-Performance Matrix Multiplication Code by Exploiting Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 21 (Apr. 2025), 22982–22990. https://doi.org/10.1609/aaai.v39i21.34461