WarpDrive: An Agentic Workflow for Ninja GPU Transformations

Sana Damani, Siva Kumar Sastry Hari, Mark Stephenson, and Christos Kozyrakis
NVIDIA
{sdamani,shari,mstephenson,ckozyrakis}@nvidia.com

Abstract

Performance engineering for GPU-accelerated applications is challenging and time-consuming. We propose WarpDrive, a customizable LLM-driven performance analysis and optimization workflow that automatically transforms and tests GPU applications. We demonstrate its effectiveness by customizing it to four different optimization levels – compiler options, compiler hints, function-level transformations, and application-level transformations.

1 Introduction

The increasing prevalence of GPU-accelerated applications has created a pressing need for efficient performance optimization techniques. While performance analysis tools, such as Nsight Systems (nsys) [10] and Nsight Compute (ncu) [9], have advanced significantly, CUDA performance engineering remains a challenging and time-consuming task as performance engineers must assess detailed profile reports and manually implement and evaluate various transformations. While this process can potentially improve performance by an order of magnitude, it is not only tedious and error-prone, but also requires significant expertise and resources. Additionally, ensuring performance portability of legacy applications written and optimized for older GPUs necessitates continuous profiling, adaptation, and testing.

GPU compilers attempt to automate this process, but have limited scope and rely on heuristics rather than profile information, making it difficult to implement complex, application-specific optimizations that require profile information and reasoning across broad sections of the code. The Meta Large Language Model Compiler (LLM Compiler) [1] enhanced a code generation foundation model with the understanding of intermediate representations, assembly language, and optimization techniques, demonstrating the potential for LLMs to act as auto-tuners that can select compiler heuristics and pass ordering. However, implementing complex optimizations not supported by the compiler remains a challenge. Moreover, while the LLM Compiler relied on the compiler for correctness, LLM-generated code can introduce errors necessitating verification after code generation. To address these challenges, we propose WarpDrive, a customizable performance engineering workflow that automates CUDA transformations using LLM agents.

2 Technical Approach

WarpDrive is an LLM-assisted profile-guided optimization (PGO) workflow that can be customized to specific CUDA optimizations. WarpDrive takes as input a user application, analyzes its profile and program characteristics to determine if an optimization should be applied, creates an optimization plan, and transforms the code. Finally, it tests the program for correctness and performance.

Figure 1 depicts the WarpDrive workflow, a typical performance engineering process that can be used to apply different types of optimizations. Each step (in green) in the workflow consists of one or more

Machine Learning for Systems Workshop at NeurIPS 2024.

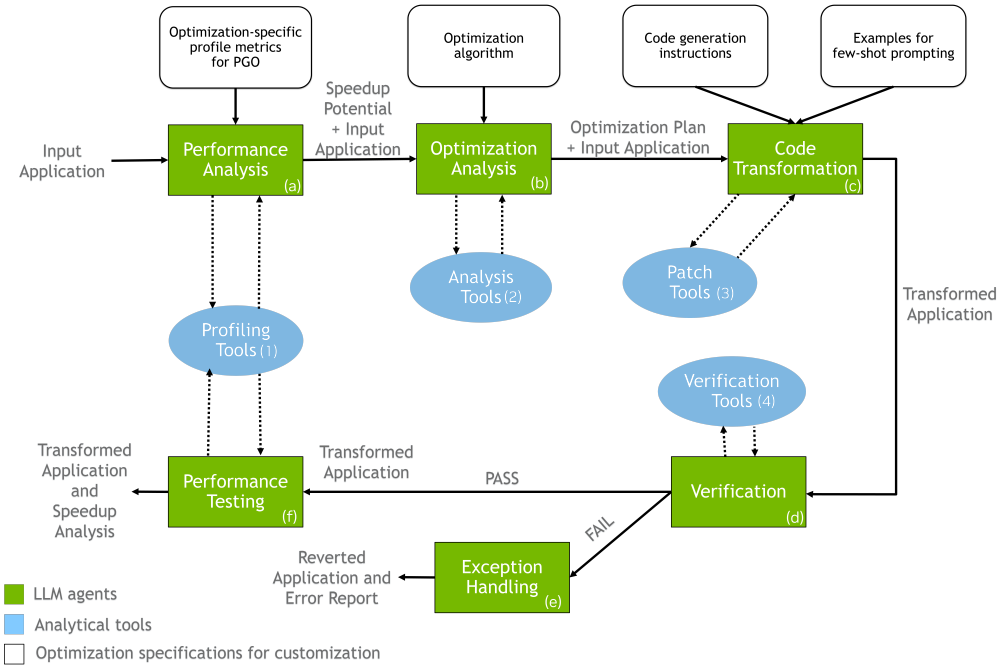

Figure 1: Profile-guided transformation agentic workflow for a single optimization algorithm.

ReAct agents [3] with access to a set of analytical tools (in blue) and provided with a system prompt customized to the optimization's specifications (in white). These prompts make use of in-context learning techniques such as chain-of-thought and few-shot prompting.

We use *Streamify* optimization as a running example to describe our workflow. CUDA streams allow for concurrent execution of kernels on a single or multi-GPU system [8]. Streamify identifies independent kernels and assigns them to different streams with appropriate inter-stream synchronization to achieve concurrency and improve GPU resource utilization [13]. WarpDrive performs the following steps for a given optimization:

(a) **Performance Analysis** is the optional first step of WarpDrive. It analyzes the execution profile generated by profiling tools, such as nsys and ncu, to determine whether a given optimization addresses performance bottlenecks in the input application. For Streamify, this step involves inspecting the profile to determine whether the application launches multiple kernels on a stream and whether these kernels exhibit low SM occupancy.

(b) **Optimization Analysis** involves one or more agents that use analytical tools and LLMs to analyze the input program in the context of the custom optimization and creates an *Optimization Plan*. For Streamify, this step involves multiple sub-steps: (i) building a kernel dependence graph, (ii) assigning streams to kernels to maximize concurrency, (iii) inserting synchronization nodes to maintain dependencies, and (iv) creating a schedule of kernels and synchronization nodes.

(c) **Code Transformation** takes as input the *Optimization Plan* as well as the baseline program and leverages the code generation capabilities of LLMs to transform the input program with the aid of patching tools. The patching tools help limit the output context length and reduce errors.

(d) **Verification** employs an LLM agent to invoke verification tools and report the outcome – pass or fail. Such tools include compilers, unit tests, and code sanitizers to check for race conditions or memory safety violations. The agent may also invoke formal verification tools that check whether the properties of the generated code meet certain specifications. For Streamify, we test the transformed program using unit tests and with a tool that compares the new dependence graph to the original dependence graph to ensure that no dependencies were violated and no kernels were omitted during the transformation step. If verification fails, the workflow invokes exception handling.

(e) **Exception Handling** allows the user to customize actions in the event of test failures. Such actions include reverting the change and returning control to a previous agent in the flow. This step may also involve invoking a debugging agent to fix any bugs in the generated code or restarting the workflow with different hyperparameters (e.g., a different model). Our current implementation simply reverts the applied change and returns to the user who can choose to restart the workflow with the same or modified specifications.

(f) **Performance Testing** measures the speedup of the transformed program over the baseline. The LLM summarizes the changes in performance metrics and provides an explanation for the speedup or slowdown if any.

We now describe the tools we implemented for use with the agents in WarpDrive to demonstrate the four optimizations described in Section 3. Additional custom optimizations or exception handling techniques may require new tools.

(1) **Profiler tools:** We use the Nsight GPU profilers as well as an annotation tool that augments the source code with per-instruction profile information, which includes execution frequency, stalls, memory bank conflicts, and memory coalescing information.

(2) **Analysis tools:** We provide python-based tools for use by the optimization analysis agents, including an occupancy calculator for register target selection [12].

(3) **Patch tools:** We provide code insertion, deletion, and replacement tools that the Code Transformation Agent can invoke to transform the input program. We also provide a "read-only" marker that allows the user to demarcate parts of the input program that the model should not modify. The patch tools ensure that these read-only sections remain unaltered to help limit the impact of errors introduced. For Streamify, the kernel definitions and host-side verification code are labeled read-only.

(4) **Verification tools:** We provide a test tool that compiles the transformed program and invokes unit tests. These unit tests are currently user-provided but may also be generated by a fuzzer. To check the correctness of the generated program, the verification agent may invoke analytical tools that verify program properties. For Streamify, our verification tool takes as input the baseline and modified dependence graphs and checks if the transformed program violates dependencies between kernels.

3 Evaluation

We implemented WarpDrive as a LangGraph based workflow [4] with React agents [3] composed of the GPT-4-Turbo model [2] and tools as described in Section 2. Our current implementation relies on in-context learning and does not involve pre-training or fine-tuning the model. We evaluate WarpDrive by customizing the workflow to perform optimizations at four different levels of complexity:

- **Compiler options** provided by the programmer guide compiler heuristics for code generation. Using compiler flags has minimal impact on correctness, as we rely on the compiler to generate code. An example of an optimization flag in CUDA is *-maxrregcount*. Because of its significant impact not only on register spills and instruction-level parallelism, but also occupancy [14], setting this flag correctly can lead to significant speedups on a GPU. Prior work has looked into the use of techniques such as autotuning to select optimal register targets for GPU kernels [5]. We customize WarpDrive to use the occupancy calculator [12] to select the register target based on profile information and program characteristics. The transformation step in this case sets an environment variable that applies the compiler flag.
- **Compiler hints** such as pragmas or cache eviction hints provided by the programmer can also guide compiler heuristics with low risk to correctness. Unlike compiler options, compiler pragmas cannot be easily inserted by an autotuner because they require code changes. For evaluation, we customize the Performance Analysis agent to identify loops of interest in the input kernel and prompt the Code Transformation step to insert loop unroll pragmas.
- **Kernel-level transformations**, such as warp specialization and caching frequently used data in shared memory, sometimes require visibility into host code as well as profile information not available to the compiler, which must make safe decisions using static information. LLM-driven kernel-level transformations allow for the automation of otherwise tedious, manual optimizations. This approach does however have a higher risk of incorrect code generation, which we try to mitigate using testing, verification, and read-only guards. For evaluation, we customize the Performance Analysis agent to identify shared memory bank conflicts and prompt the Code

Transformation step to pad the shared memory allocations and change memory indexing to address such conflicts.

- **App-level transformations** include inter-kernel optimizations such as kernel fusion and Streamify that require insight into overall system profile information, host code, kernel code, and kernel-level profile information. Since the CUDA compiler only has visibility into a single kernel [11], it cannot perform inter-kernel transformations. Other high-level compilers similarly lack visibility into the kernel implementation details. LLM-driven application-level transformations leverage the broad visibility they have in to the source code and profiler report to make informed code generation decisions. As detailed in section 2, we customize WarpDrive to implement *Streamify*.

We evaluate WarpDrive across a set of microbenchmarks written in CUDA and Warp [6]. Table 1 shows the speedup for these microbenchmarks using WarpDrive running on an Ampere A100 GPU[7]. These results show the potential for the use of LLMs to automatically apply CUDA optimizations that are difficult for compilers or autotuners to perform and too tedious for manual implementation.

Optimization	Class	Performance Metrics	Speedup
Register Target Selection	Compiler Options	Occupancy, Stalls, Spills	1.59x
Loop Unrolling	Compiler Hints	Stalls	1.07x
Shared Memory Padding	Kernel-level	Bank conflicts	5.50x
Streamify	App-level	Underutilization of GPU Resources	1.38x

Table 1: Experimental results for our microbenchmarks.

4 Challenges and Future Work

WarpDrive harnesses the capabilities of modern AI to perform high-level code transformations that go well beyond the scope of traditional compiler transformations. Unlike traditional compilers, which are limited by predefined transformation rules, WarpDrive grants code transformation agents great latitude in the optimization process, which in the limit could theoretically include wholesale algorithmic changes to input programs.

While this flexibility could unlock large performance gains, it also presents new challenges. Traditional compiler algorithms, which typically yield small performance improvements, are designed to be correct by construction, and ensure that any code transformations performed on a program preserve the program's semantics for all possible inputs. In contrast, AI agents are not infallible and can introduce errors during code generation.

We have begun exploring potential guardrails to help detect AI code generation errors. We argue that verification and exception handling should be an integral part of the framework, and intimately connected to code transformation agents. For some optimizations, such as the Register Target Selection and Loop Unrolling case studies in Section 3, verification is trivial, but for other optimizations it is less clear how to identify logical errors and buggy transformations.

We have already implemented a required "Unit Test" suite that is suitable for catching obvious errors. We have also implemented user-directed read-only sections that prevent optimization agents from modifying the code therein. Within modifiable regions, future work should consider using static invariant checking tools that WarpDrive optimizations can invoke to verify that key program properties are preserved. For example, for the Streamify case study, we expect the kernel dependency graph that the optimization agent uses to be isomorphic to the resultant CUDA graph. Where static tools are insufficient, we can also consider relying on dynamic assertion checks and sanitizer tools during the verification phase to identify transformation bugs. Finally, future work should consider how best to involve human expertise in the verification process.

In addition to correctness concerns, WarpDrive is currently limited by context length. While GPT-4-Turbo's context length was sufficient for the case studies and micro-benchmarks we explore in this paper, real-world applications will require agents and tools that facilitate navigation through large code bases. Future work will aim to address these challenges using static analysis, navigation tools, and dynamic traces.

References

[1] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta Large Language Model Compiler: Foundation Models of Compiler Optimization, 2024. URL https://arxiv.org/abs/2407.02524.

[2] OpenAI et al. GPT-4 Technical Report, 2024. URL https://arxiv.org/abs/2303.08774.

[3] Shunyu Yao et al. ReAct: Synergizing Reasoning and Acting in Language Models, 2023. URL https://arxiv.org/abs/2210.03629.

[4] LangChain. LangGraph. https://github.com/langchain-ai/langgraph.

[5] Ang Li, Shuaiwen Leon Song, Akash Kumar, Eddy Z. Zhang, Daniel Chavarría-Miranda, and Henk Corporaal. Critical points based register-concurrency autotuning for gpus. In *2016 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1273–1278, 2016.

[6] Miles Macklin. Warp: A High-performance Python Framework for GPU Simulation and Graphics. https://github.com/nvidia/warp, March 2022. NVIDIA GPU Technology Conference (GTC).

[7] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper, .

[8] NVIDIA. CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide, .

[9] NVIDIA. Nsight Compute. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html, .

[10] NVIDIA. Nsight Systems. https://docs.nvidia.com/nsight-systems/UserGuide/index.html, .

[11] NVIDIA. Nvidia cuda compiler driver nvcc. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, .

[12] NVIDIA. CUDA Occupancy Calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls, .

[13] NVIDIA. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, .

[14] NVIDIA. GPU Occupancy. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm, .