ABSTRACT

Title of Dissertation: LLM-DRIVEN DEVELOPMENT OF PORTABLE AND PERFORMANT GPU CODES

Joshua Hoke Davis

Dissertation Directed by: Professor Abhinav Bhatele
Department of Computer Science

Supercomputers increasingly rely on a widening range of Graphics Processing Units (GPUs) to provide maximal computing power at minimal energy consumption and deployment cost. As a result, their users need performance portability, or the ability for a single application to run with good performance (e.g., time to solution) across multiple supercomputers. Portable programming models allow a single piece of code to execute on multiple GPU types through an abstract interface, but nevertheless performance portability remains challenging to achieve. The extent to which each of the many models available actually enable performance portability is not well understood, and the process of converting an existing application codebase to use a new model is time-consuming, making it difficult to test and compare multiple options. Furthermore, when a programming model fails to deliver desired performance on a new GPU architecture, identifying ways to improve performance requires expert-level understanding of the hardware and programming model. In this dissertation, we focus on these key challenges to performance portability, dividing the problem of performance portability into three stages: planning, implementation,

and optimization. We present results of a comparative study which provides a comprehensive overview of how well GPU programming models enable performance portability, along with a novel level of depth in the analysis of results. To address the tedium and time expense of converting an application to a portable programming model, we also present a benchmarking effort which compares agentic and non-agentic translation methods using a range of open-source and commercial large language models (LLMs). Finally, to improve the ease of optimizing GPU kernels on new architectures, we present KEET, an LLM-based agentic framework for analyzing and explaining the performance of CUDA kernels using Nsight Compute profiles.

LLM-DRIVEN DEVELOPMENT OF PORTABLE AND PERFORMANT GPU CODES

by

Joshua Hoke Davis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2026

Dissertation Committee:

Professor Abhinav Bhatele, Chair
Professor Shuvra S. Bhattacharyya, Dean's Representative
Professor Alan Sussman, Full Member
Professor Mohit Iyyer, Full Member
Professor Sunita Chandrasekaran, Special Member

To my partner and wife-to-be, Hannah.

Acknowledgements

Countless individuals, whether through collaboration or companionship, have supported me through this journey. I owe a debt of gratitude to each of them.

I am tremendously grateful to have had an inspiring and insightful advisor, Professor Abhinav Bhatele, throughout my time as a Ph.D. student. It is difficult to overstate the impact of his guidance on my development as a researcher, from helping reshape my initial research ideas to feedback on my writing and data presentation. I am thankful for both his high standards and his patience and understanding.

I would like to thank my fellow graduate students at the Parallel Software and Systems Group, whose advice and support made the environment more creative, productive, and fun, whether in the lab or at Dan's Cafe. To all my colleagues and mentors beyond the University of Maryland, including at the University of Delaware, LLNL, NASA Langley, and NVIDIA, I am sincerely grateful for your willingness to share your expertise and provide opportunities to intern. I also express my gratitude to the great number of talented and dedicated undergraduate students at Maryland who I have had the opportunity to work with. I would be remiss not to acknowledge the support of the National Science Foundation Graduate Research Fellowship Program, which provided funding for three years of my graduate studies, and the University of Maryland, which covered the other two through fellowships and assistantships.

To my parents, who made me who I am today, thank you for supporting my education and inspiring me to always stay curious and hold true to my values and my passions. I hope that you will be proud when you read this dissertation. To my wide circle of family and friends, thank you for your encouragement, patience with my somewhat chaotic schedule, and willingness to listen

to and ask questions about my work. And finally, I cannot express enough how grateful I am to my fiancée, Hannah. Thank you for being my constant companion and confidant every single day of this program. You provided the sympathy, strength, support, and occasional pep talks and last-minute pushes I needed to get through the toughest moments. I could not have done this without you.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
BW	Bandwidth
CTA	Cooperative Thread Array
CUDA	Compute Unified Device Architecture
FLOPS	Floating Point Operations per Second
GCD	Graphics Compute Die
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
HIP	Heterogeneous-compute Interface for Portability
HPC	High Performance Computing
LBL	Lawrence Berkeley National Laboratory
LDG	Load from Global Memory
LLM	Large Language Model
LLNL	Lawrence Livermore National Laboratory
MCQ	Multiple-Choice Question
ML	Machine Learning
NCSA	National Center for Supercomputing Applications
NCU	Nsight Compute
NERSC	National Energy Research Scientific Computing Center
OPT	Optimization
ORNL	Oak Ridge National Laboratory
OSS	Open-Source Software
PM	Programming Model
PRI	Python Report Interface
SHMEM	Shared Memory
SM	Streaming Multiprocessor
STG	Store to Global Memory
USM	Unified Shared Memory

Chapter 1: Introduction

Supercomputers increasingly rely on a widening range of Graphics Processing Units (GPUs) to provide maximal computing power at minimal energy consumption and deployment cost. As a result, their users need performance portability, or the ability for a single application to run with good performance (e.g., time to solution) across multiple supercomputers. Portable programming models allow a single piece of code to execute on multiple GPU types through an abstract interface, but nevertheless performance portability remains challenging to achieve. The extent to which each of the many models available actually enable performance portability is not well understood, and the process of converting an existing application codebase to use a new model is time-consuming, making it difficult to test and compare multiple options. Furthermore, when a programming model fails to deliver desired performance on a new GPU architecture, identifying ways to improve performance requires expert-level understanding of the hardware and programming model. Focusing on these key challenges to performance portability, we divide the problem of performance portability into three stages: planning, implementation, and optimization.

Application developers need a more detailed understanding of how choice of programming model and application characteristics influence performance portability. To this end, we compare and analyze programming models in terms of how well they enable performance portability for a range of proxy applications. To understand the reasons for performance differences between

programming models, we employ comparative performance profiling and analysis to identify and resolve causes of performance degradation between portable and non-portable implementations of the same application.

Even with an effective programming model, converting an application to use that model remains laborious. We examine the use of Large Language Models (LLMs) to automate translation of source code between parallel programming models. We assemble a benchmark suite, PAREVAL-REPO, to compare full application translation approaches, extending the approach of prior work to full-scale application translation. We evaluate PAREVAL-REPO on a range of LLMs using agentic and non-agentic approaches to prompting. We also consider the inference expense of translation as it is affected by accuracy and LLM strength. Ultimately, PAREVAL-REPO provides a foundation to prepare tools that can not only simplify porting applications for developers, but also enable new studies of programming models by enabling automatic generation of new programming model ports to compare.

Tools which can explain performance behavior and automatically identify optimization opportunities in GPU code have tremendous potential for developer productivity. Tuning kernel performance for every new GPU architecture or vendor is time-consuming. Existing work has proposed automated approaches to assist with analyzing GPU kernel performance on NVIDIA GPUs, but they take a rule-based approach that can be somewhat rigid. Statically-defined suggestions do not consider how specific characteristics of the kernel algorithm might interact with performance bottlenecks. Further, existing tools require manual intervention to incorporate new GPU hardware features or performance metrics when new GPUs are released, and do not suggest code changes tailored to the existing kernel code. We present a new approach to address these limitations by interpreting GPU kernel performance profiles using Large Language Models

(LLMs). KEET, an LLM-based agentic framework, automatically generates a natural language report explaining the performance of the kernel using NCU profiles. We evaluate KEET against prior work using a suite of CUDA kernels.

1.1 Outline of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background details on performance portability, portable programming models, LLMs, and GPU performance analysis. Chapter 3 surveys related work in performance portability, LLM-driven code translation, and GPU performance analysis. Chapter 4 presents the study of performance portability enabled by different programming models. Chapter 5 presents the study of LLM-driven code translation. Chapter 6 presents the study of GPU performance analysis using LLMs. Chapter 7 presents a workflow for enabling performance portability in a hypothetical application, combining the lessons learned and tools developed in the proceeding chapters. Finally, Chapter 8 summarizes the dissertation and discusses directions for future work.

Chapter 2: Background

In this chapter, we provide background details on performance portability, portable programming models, LLMs, and GPU performance analysis.

2.1 Background: Portable Programming Models

In this section, we provide relevant background information on the various programming models we evaluate. HIP and CUDA act as our baselines in this study, as they are the native models for AMD and NVIDIA devices, respectively. Below, we describe the key characteristics of each category of programming model. All programming models used in this study support both NVIDIA and AMD devices except CUDA.

2.1.1 Language extensions

SYCL, HIP, and CUDA are language extensions, which add features to the base language (C++, C, and/or Fortran) for programming GPUs. SYCL and HIP are open standards, while CUDA is proprietary. The language extensions we consider are more verbose than the other programming models. Users call runtime functions to manage memory and write functions that they then invoke as kernels to offload execution. SYCL provides multiple methods of memory management, including the *explicit USM (unified shared memory)* API, which uses CUDA or HIP

style runtime calls to move and allocate data, or the *buffer/accessor* API, which is more implicit, allowing the compiler and runtime to schedule data movement but not allowing explicit access to valid device pointers.

2.1.2 C++ abstraction libraries

Kokkos and RAJA are C++ abstraction libraries. These are template-based C++ libraries that provide high-level functions and data types. Users write their code directly employing these data types and typically structure GPU code as lambdas to pass into library function calls. The library translates the user code to a device backend such as CUDA, HIP, or OpenMP at compile-time or runtime. Note that Kokkos provides both memory and compute abstractions, while RAJA provides compute abstractions and users must employ the related Umpire or CHAI libraries to abstract memory management.

2.1.3 Directive-based models

OpenMP and OpenACC are directive-based models. They provide compiler directives, or *pragmas*, to parallelize or offload code. They are typically standard specifications implemented by a compiler front-end and a runtime library to implement parallel or offloaded execution that abstracts the underlying hardware architecture. Directive-based models are usually less verbose and less intrusive, as users can often annotate existing code with minimal refactoring. This facilitates incremental development. These models provide clauses and standalone directives to schedule data movement, which is carried out by the compiler and device runtime.

2.2 LLMs for Code Generation and Translation

In this section, we provide a brief introduction to large language models, their use in code generation and translation, existing metrics for assessing generated code, and agentic AI software engineering tools. We also include background on the problems faced in manual translation of scientific applications between parallel programming models.

To avoid ambiguity we employ the term *LLM* to refer to large language models (such as `llama-3.3` or `gpt-4o`) and *programming model* to refer to the parallel programming model (such as CUDA or Kokkos).

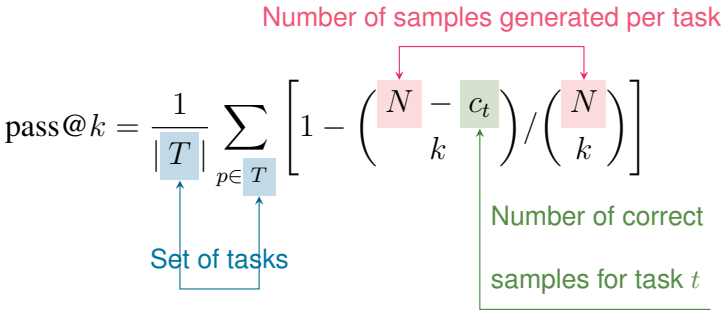
2.2.1 LLMs and reasoning LLMs

LLMs have become the predominant approach to text generation. Given a sequence of text, represented as a series of *tokens*, they iteratively predict the next token in the sequence to generate an extended sequence. More recently, LLMs have been fine-tuned with reasoning capabilities. These models are trained to first generate reasoning to a solution before generating the solution itself. DeepSeek-R1 [25] and other efforts have found that this reasoning training encourages the model to question and correct itself. The result is LLMs with powerful reasoning capabilities that excel at problem-solving tasks, including those that arise in programming.

2.2.2 LLMs for code generation

The use of LLMs for code generation in high-performance computing contexts has been widely studied [66, 64, 40, 95]. As employed in prior efforts to assess the quality of parallel code generated by LLMs [66], we adopt the pass@k metric to estimate the likelihood of getting

a correct output given k attempts by the LLM. We define correctness for our translation tasks in section 5.5.1. Here, we provide the definition of $pass@k$, from [66]. Estimating pass@k requires generating N samples for a given LLM and task combination, where $N > k$. The number of correct samples c_t for a task t, as well as N and k, determine the estimation of pass@k as demonstrated by Equation (2.1).



$$\text{pass@}k = \frac{1}{|T|}\sum_{p\in T}\left[1 - \binom{N - c_t}{k} \Big/ \binom{N}{k}\right] \tag{2.1}$$

The speedup@k score [66] is conceptually similar to the pass@k score, but estimates the expected speedup in performance over the original (unoptimized) code achieved with k attempts. For N samples, where $N > k$, the speedup@k score is calculated as Equation (2.2). We denote the performance of the original code by T_p^*, and the performance of the updated code on sample j by $T_{p,j}$, where p is a problem from a set of P problems.

$$\text{speedup@}k = \frac{1}{|P|}\sum_{p\in P}\sum_{j=1}^{N}\frac{\binom{j-1}{k-1}}{\binom{N}{k}}\frac{T_p^*}{T_{p,j}} \tag{2.2}$$

We evaluate the quality of the LLM-generated code using the pass@k and speedup@k scores with $N = 20$ samples and $k = 1$ attempt.

2.2.3 Software engineering agents

There has been growing interest in *AI agents*: autonomous, LLM-driven systems capable of creating or modifying code with minimal human intervention. One notable example employed in this chapter is *SWE-agent*, a state-of-the-art autonomous coding agent designed to develop fixes for small software engineering (SWE) issues. SWE-agent leverages LLMs in a closed-loop framework including program analysis, generating and executing tests, and version control. It first generates a high-level strategy for resolving a given issue, before applying appropriate changes using its specialized editing tools, across multiple files if needed. Additionally, SWE-agent grounds its changes in real-time iterative feedback, allowing it to validate the correctness of patches through a single execution run. These features make SWE-agent potentially well-suited for complex repository-scale refactoring.

2.2.4 Scientific application translation

Translating scientific applications into new programming models, whether automatically or manually, has been a perennial challenge as the supercomputing hardware landscape continues to diversify. The manual form of this process, more commonly referred to as porting, first became of interest for implementation of grid computing support [45, 39], but ultimately became urgent with the advent of general-purpose computing on GPUs [52]. Even without the added concern of portability across programming models, converting legacy codes to use CUDA, NVIDIA's proprietary GPU programming model, has been studied extensively [26, 96, 28]. As additional vendors and GPU architectures proliferate in new supercomputers, the requirement for *portable* GPU support becomes predominant, compelling application developers who may have already

committed the effort to port to CUDA to now port to a newer and more abstract portable programming model like OpenMP, RAJA, Kokkos, or SYCL [70, 94, 7, 81]. Case studies in porting applications to portable GPU programming models have also been extensively documented [5, 58, 17, 35].

Across these documented experiences of manual translation, as well as some instances of partially-automatic non-LLM techniques [3, 96, 28], the user support services, collaboration between vendors and developers, careful high-level strategizing, and tedious manual effort required are consistently emphasized. Through the translation process, developers must consider changes to the build system, interfaces between files, parallelization strategy, and even overall program control flow. Furthermore, most pre-LLM automation efforts focus on assisting with planning ports, and only translate code directly using dictionary-style knowledge bases which explicitly map code constructs to their translated versions [3], if at all.

Converting existing scientific applications from non-portable to portable GPU programming models remains difficult yet urgent work. In Section 5.2, we describe the methods we compare for addressing this difficulty.

2.3 GPU Performance Analysis Tools

Below we provide some background details on the Nsight Compute (NCU) profiling tool, metrics for evaluating LLM-generated GPU code, and the DrGPU tool that we optionally incorporate into our framework.

2.3.1 Nsight Compute profiler

Nsight Compute (NCU) is a profiling tool for NVIDIA GPUs that collects detailed performance metrics at the level of individual GPU kernels, down to the source code line level if desired. NCU's command-line interface allows users to specify which kernels in the application to profile and which metrics to collect, alongside other options. NCU profiles targeted kernels by replaying each kernel for multiple passes, collecting different hardware metrics on each pass. NCU determines the number of passes required based on the set of metrics specified by the user.

After profiling, NCU generates a profile report file (`.ncu-rep`) containing the collected metrics for each kernel. The report file can be viewed in the NCU graphical interface, which provides a variety of views of the collected data. These views include a summary view of the basic details of each kernel profiled, a detailed view of the metrics collected for a selected kernel, grouped into sections roughly by hardware component, and a source code view that maps metrics to source code and assembly lines. Notably, the detailed view also provides outputs from a rule engine that generates callouts mentioning potential performance issues and suggestions based on fixed metric thresholds.

Data generated by NCU is typically viewed through the graphical interface, but it can also be extracted from the `.ncu-rep` file using NCU's Python Report Interface (PRI). PRI allows for programmatically viewing collected metric names, values, units, and other metadata for each kernel profiled in the report file. Importantly, PRI does not provide full access to line-level profiling data, which must be manually exported to a CSV file through the graphical interface. We primarily use PRI to extract the raw metric data for each kernel profiled for use in our tool's analysis. DrGPU, as described in Section 2.3.2, also requires the line-level data to produce the

best results. We manually export the required CSV file to support DrGPU's analysis in both baseline and KEET evaluations.

2.3.2 DrGPU profile analysis tool

DrGPU is a non-LLM-based tool that generates performance reports and optimization suggestions based on an NCU profile. It examines kernel performance by decomposing stall reasons into categories in a tree structure, listing root causes at leaf nodes. DrGPU offers suggestions to address the most frequent stall reasons on these leaf nodes. The tool requires a specific set of NCU metrics to run its analysis. Furthermore, in order to provide source code lines alongside its suggestions, it also requires the user to manually export a CSV file containing the line-level profiling data, as described in Section 2.3.1. In this chapter we compare KEET against DrGPU as a baseline and optionally integrate its suggestions into our tool's analysis.

Chapter 3: Related Work

In this section, I summarize existing literature in examining performance portability, as well as highlight state-of-the-art approaches in LLM-driven code translation for parallel and repo-level tasks and automation of GPU performance analysis.

3.1 Examining Performance Portability in Practice

Several studies on programming language extensions, models, and libraries have been designed to assist developers achieve performance portability [70, 84, 8, 47, 94]. Additionally, several studies have assessed the portability of certain frameworks. We categorize the related work on empirical performance portability studies into three groups: metric studies, application or programming model studies, and broader studies that are not scoped to a particular model or app. In this section, we provide an overview of recent work in each category.

3.1.1 Studies of performance portability metrics

Pennycook et al. propose the metric Φ for performance portability, defining it as the harmonic mean of the performance efficiencies of an application across different systems [73, 74, 87, 72, 71]. Daniel et al. propose an alternative metric, P_D, which accounts for problem size [16], and Marowka compares Φ with $\overline{\Phi}$, a similar metric that uses the arithmetic mean instead of the

harmonic mean [59, 60].

3.1.2 Studies involving individual application categories or programming models

A number of studies evaluate performance portability in specific applications with multiple programming models or a single programming model model [61, 80, 79, 86, 12, 31, 20, 4, 77, 37, 84, 13, 78, 49]. For instance, Dufek et al. compare Kokkos and SYCL for the Milc-Dslash benchmark [31], while Rangel et al. examine the portability of CRK-HACC in SYCL [77]. Other studies investigate performance portability across applications using specific programming models. Brunst et al. benchmark the 2021 SPEChpc suite, which contains nine mini-applications in OpenMP and OpenACC, on Intel CPUs and NVIDIA and AMD GPUs [13]. Kuncham et al. evaluate the relative performance of SYCL and CUDA on the NVIDIA V100 using BabelStream, Mixbench, and Tiled Matrix-Multiplication [78].

While these studies provide useful information to developers working on similar applications or those interested in specific programming models, making more general statements about programming models themselves requires a more comprehensive evaluation of a diverse set of case studies.

3.1.3 Broader performance portability studies

Deakin et al. present performance portability studies of five programming models across a wide range of hardware architectures, using BabelStream, TeaLeaf, CloverLeaf, Neutral, and MiniFMM [22, 23]. More recent papers by Deakin et al. focus on more specific problems such as

reductions and GPU to CPU portability [19, 21]. Lin et al. evaluate implementations of C++17 StdPar against five models on AMD devices [56]. While these studies provide performance portability comparisons across systems, applications, and models, they do not include RAJA and sometimes omit HIP and OpenACC. Furthermore, they do not provide extensive analysis of the reasons for performance differences between programming models or ways to address differences.

Several other studies are similar in scope but different in focus. Kwack et al. evaluate portability development experiences for three full applications and three proxy applications across GPUs from multiple vendors [53]. Harrell et al. study performance portability alongside developer productivity [42]. However, in these studies each application is only ported to a single portable programming model. This makes it difficult to draw conclusions about each programming model's relative suitability to particular applications. Koskela et al. provide six principles for reproducible portability benchmarking, along with a demonstration of these principles in a Spack+Reframe CI infrastructure for a study of BabelStream on some CPU architectures and an NVIDIA V100 [51]. Other studies uniformly fail to follow these principles, making reproducing them an arduous task.

Overall, broader studies are generally limited in their breadth and reproducibility. They typically fail to investigate the underlying relationship between application characteristics and the performance portability of different programming models. Additionally, almost all prior studies are difficult to reproduce, lacking details on the build and run infrastructure used to collect their results, and leave the task of building applications on a wide range of systems with complex library and compiler flag dependencies to the reader.

3.2 LLMs for Code Translation

Prior studies have examined using LLMs for code translation. We separate the relevant literature into two categories, repo-level translation and parallel code translation and generation.

3.2.1 Repository-level code translation

Automating repository-level translation has been a growing area of focus. Prior studies leverage LLMs to minimize human intervention in the translation process when scaling translation to larger codebases. For instance, AlphaTrans explores translating Java to Python by breaking down large repositories into smaller, manageable fragments and iteratively validating results for syntactic correctness and functional equivalence [48]. Additional work also considers the problem from a benchmarking perspective. RepoTransBench proposes a benchmark suite of Python to Java translation tasks for assessing LLM translation capabilities, finding that LLMs consistenly achieve below-par results for these tasks [97]. However, these projects both consider translation between Java and Python, without parallel programming models and with no consideration of languages more common in high-performance computing, including C/C++ and Fortran. Furthermore, the methodologies in these studies is tightly integrated with Java and Python code and build systems, making the techniques difficult to generalize to other workflows, particularly those in HPC. As an example, we discovered that SWE-agent does not work with Makefiles, which are a critical part of many HPC codes.

3.2.2 Parallel code translation

Several efforts to develop LLM tools to translate parallel code exist. First, CodeRosetta [89] proposes an encoder-decoder transformer model to translate between C++ and CUDA as well as Fortran and C++. Unfortunately, its approach is evaluated only using the CodeBLEU similar score, without genuine runtime data. Similarly, Code-Scribe utilizes a scoped code hierarchy and retrieval-augmented generation (RAG) to guide LLM-based translations from Fortran to C++, ensuring compatibility through intermediate Fortran-C APIs [27]. LASSI, on the other hand, focuses on translating parallel programming models, such as CUDA and OpenMP, using iterative compilation and runtime feedback for self-correction [24]. Similarly, the dataset introduced in by Lei et al. [55] focuses on CPU OpenMP Fortran and C++. but does not extend or generalize to other programming models like Kokkos or OpenMP GPU offloading, and relies on only the CodeBLEU similar metric and human evaluation.

While these strategies demonstrate progress in translating parallel code, they often do not consider genuine code functionality, instead employing CodeBLEU code similarirty scores to assess translation quality, or fail to generalization to diverse programming models and new applications. Furthermore, many employ translation tasks that have already have publically-available implementations. For example, LASSI relies on HeCBench, which include reference implementations for CUDA and OpenMP already. This creates a risk of training dataset contamination with test problems, making it possible for LLMs to recite already-seen translated code rather than reasoning through the problem and carrying out a genuine translation.

3.3 Performance Analysis Tools and Automation Techniques

In this section I summarize related work in analyzing and generating high-performance parallel and GPU codes with and without the assistance of LLMs.

3.3.1 Non-LLM-based tools

Existing non-LLM-based tools for analyzing GPU kernel performance data include GPA [105, 104] and DrGPU [41]. GPA, the GPU Performance Advisor, is a standalone tool that combines static analysis and runtime stall measurement to attribute stalls to instructions and suggest optimizations based on performance models [105]. DrGPU, which succeeds GPA and presents results in comparison to GPA, is a tree-centric tool which analyzes stall reasons collected with NCU and decomposes them into a tree structure, offering suggestions to address the most frequent stall reasons [41]. We discuss DrGPU in greater detail in Section 2.3.2, as KEET can optionally integrate DrGPU's suggestions into its analysis. We also directly compare KEET against DrGPU in our evaluation. While both of these tools are useful for interpreting GPU kernel performance data, they are limited to providing generic suggestions based on fixed metric thresholds or performance models. They also cannot reason generally about kernel source code and its relationship to performance bottlenecks.

There is also extensive prior work on non-LLM-based tools for automating performance analysis in HPC contexts generally. Hatchet [10] enables automating common performance analysis tasks in Python across profiling tools. Pipit [11] proposes a similar framework focusing on execution trace analysis. HPCToolkit [1, 2] has extensive support for collecting and visualizing performance profiles for both CPU and GPU workloads across vendors. AMD's ROCm Compute

Profiler [82] is a tool for collecting performance data for AMD GPUs, similar to Nsight Compute for NVIDIA GPUs [67]. PARAVER [75] is a tool for visualizing parallel code execution traces. EasyView [103] incorporates profile data directly into integrated development environment (IDE) software. Keiff et al. [50] propose a framework that provides a unified interface for a range of performance analysis tools and simplifies their usage. HTA [46] supports analysis of PyTorch application performance, and Scalasca [38] automatically analyzes execution traces for specific patterns and provides an interactive report and timeline to explore the results. CATS [85] supports tracing and analysis of memory and control flow patterns in parallel programs to identify performance bottlenecks. KEET focuses specifically on the problem of interpreting low-level GPU performance metrics, and leverages LLMs to provide natural language explanations and optimization suggestions tuned to the specific kernel under analysis.

3.3.2 LLM-based tools

LLMs have been shown to be useful for generating, reasoning about, and optimizing kernel source code, with and without performance data. Nichols et al. [66] develop ParEval, a framework which tests LLM capabilities for generating parallel code, including CUDA kernels. Cui et al. [14] assess LLM understanding of performance optimization using a suite of CPU-based HPC codes from varying domains, finding that LLMs struggle with maintaining correctness when optimizing code. Lange et al. [54] propose a framework which uses LLMs to translate PyTorch code into CUDA kernel code, and then optimize the kernel code in an agentic loop including a profiling stage. Zaeed et al. [101] propose Opal, a framework which incorporates NCU metrics and rule engine outputs to prompt GPT-4o to optimize CUDA kernels. Yu et al. [100] survey the

wide range of proposed techniques for using LLMs to generate CUDA kernels. Dai et al. [15] propose CUDA Agent, an agentic RL framework incorporating LLMs to generate CUDA kernels in an agentic development environment with access to profiling tools. Zhang et al. [102] propose CudaForge, a framework incorporating NCU data with Judge and Coder agents to optimize CUDA kernels. These frameworks have shown the promise of using LLMs to generate and optimize CUDA kernels with the assistance of profiling tools. Nsight Compute itself also includes NCU Copilot, an LLM integrated into the GUI to provide explanations of GPU and kernel performance concepts and suggest code changes [67]. KEET builds upon these efforts by focusing on the problem of generating natural language explanations of GPU kernel performance data, specifically in an HPC context.

Chapter 4: Evaluating Performance Portability of GPU Programming Models

4.1 Introduction

Heterogeneous CPU-GPU architectures have come to dominate the design of high performance computing (HPC) systems. Nine of the top ten systems in the June 2024 TOP500 list, and ~39% of the systems on the complete list, employ co-processors or accelerators [90]. Further, a diverse set of specific architectures are in use, supplied by a range of vendors, as the current top ten includes GPUs from AMD, NVIDIA, and Intel. A similarly-diverse range of programming models has emerged, which aim to allow application developers to write their code once and run it on any system. Programming models such as OpenMP [70], RAJA [47], and Kokkos [94] act as *portability layers*, bridging the gap between high-level implementation of an algorithm and low-level execution on a given target architecture. Yet running scientific applications efficiently on HPC systems requires more than just *functional* portability, which refers to program correctness. Codes must also perform well on a range of target systems, ideally without incurring the technical debt of system-specific implementations. This is often referred to as *performance* portability.

Application developers would benefit from a deeper understanding of the performance portability provided by different programming models on modern GPU systems before porting their application to a particular model. Choosing a programming model for porting a CPU-only application to GPUs is a major commitment, requiring significant time for developer training and

programming. If a programming model delivers unacceptable performance, then that investment is wasted.

Nevertheless, each programming model's effectiveness at enabling performance portability, as well as the definition of performance portability itself, remain open questions. Although developers' experiences comparing the performance portability of several models on a single application are valuable, we have observed that open-source applications or even proxy applications implemented in a several different programming models are uncommon and difficult to find. Further, a single smaller application or benchmark implemented in most programming models is unlikely to be representative of the diverse and complex production applications typically run on HPC systems. Finally, conducting exhaustive combinatorial studies of programming model, compiler, system, and application combinations is a significant undertaking, as each programming model usually requires unique combinations of compilers flags and libraries for any given system.

In this chapter, we provide a comprehensive empirical study of the performance portability of several programming models on GPU-based leadership-class supercomputers. We use a variety of proxy applications that are representative of production codes, and using them, we enable realistic comparisons of the performance portability of GPU kernels written in several programming models across different architectures. We study five proxy applications from different scientific domains, create implementations where missing, and comprehensively evaluate differences between these programming models.

We present a Spack-based [36] environment and scripting system to significantly lower the barrier for performance portability studies. This system encapsulates our methodology for systematically building, running and benchmarking a suite of applications in several program-

ming models, in a manner which can be adapted for future studies. Our comparative evaluation of model performance includes specific insights into why certain programming models perform well or poorly for particular applications on different target systems. To our knowledge, this is one of the most comprehensive performance portability studies to date, in terms of the breadth of programming models and applications studied and the detail provided in the analysis of results.

4.2 Methodology for Evaluating Performance Portability on GPU Platforms

In this section, we describe our approach to comprehensively compare programming models that provide portability on GPU systems. We also justify for our choices of programming models, proxy applications, systems, and metrics.

4.2.1 Choice of programming models

Our goal in this work is to empirically compare the performance portability provided by popular programming models. In Section 2.1, we describe three categories of programming models with a few examples in each category. We identified those representative models by surveying a broad range of proxy applications in order to determine how common existing implementations in each model were. We surveyed a variety of sources for proxy applications, including the ECP Proxy Apps suite [32], the NERSC Proxy suite [65] and the Mantevo Applications Suite [44]. Armed with that knowledge, we have decided to focus on CUDA, HIP, SYCL, Kokkos, RAJA, OpenACC, and OpenMP, as they were most commonly found in the proxy applications we surveyed. Together, these models cover the three categories of models mentioned earlier.

4.2.2 Choice of proxy applications

Based on the survey of proxy applications mentioned above, we identify five applications that represent the range of typical scientific computing workloads on GPU clusters. These include a pure memory bandwidth benchmark as well as four other proxy applications. They range from highly compute-intensive (miniBUDE) to highly memory-intensive (BabelStream), and also include one representative from each of the three large proxy application suites we surveyed. The scientific domains covered by them include hydrodynamics (CloverLeaf), molecular dynamics (miniBUDE), nuclear physics (XSBench), and particle physics (su3_bench), and computational methods include structured grid (CloverLeaf and su3_bench), dense linear algebra (su3_bench), n-body (miniBUDE) and Monte Carlo (XSBench) methods.

CloverLeaf, miniBUDE, and XSBench are missing implementations in some programming models compared in this work. So, we develop these missing implementations to obtain full coverage of the space of application and model combinations. Table 4.2 summarizes the key details of each proxy application, and identifies the implementations that were either created or modified by us for this study. Our modifications consist of small changes to the memory management library or style to ensure portability and consistency of gathering execution times across implementations. Below, we describe in brief the five proxy applications used in this study:

BabelStream is a memory bandwidth benchmark with five kernels: `copy`, `add`, `mul`, `triad`, and `dot` [20]. The dot kernel includes a reduction operation, known to be a challenging operation for some programming models [18].

XSBench [93] is a proxy for OpenMC, a Monte Carlo transport code [83]. XSBench runs one kernel, OpenMC's macroscopic cross-section lookup kernel, with a large number of lookups. We use the event-based transport method with a hash-based grid as it is preferred for GPUs.

CloverLeaf is a 2D structured compressible Euler equation solver, with 14 kernels [43]. The `advec_mom`, `advec_cell`, `PdV`, and `calc_dt` kernels are typically the most time-intensive, and `calc_dt` contains a reduction.

su3_bench [29] is a proxy application for MILC, a lattice quantum chromodynamics code [9]. It implements the SU(3) matrix-matrix multiply routine in its lone kernel.

miniBUDE is a proxy for Bristol University Docking Engine (BUDE), a molecular dynamics code which simulates molecular docking for drug discovery [62]. miniBUDE computes the energy field for a single configuration of a protein repeatedly.

4.2.3 Choice of systems

Evaluating performance portability requires selecting a range of systems with diverse architectures. One of the main goals of this study is to evaluate performance portability on production GPU-based supercomputers, given the rising prominence of GPUs in new systems [90]. We select five different supercomputers for our experiments: Summit and Frontier at ORNL, Perlmutter at NERSC, Corona at LLNL, and Zaratan at the University of Maryland (UMD) (architectural details in Table 4.1). These systems cover the majority of the GPU architectures in the top ten systems. Frontier and Summit are in the top ten, and Perlmutter is in the top fifteen. We include Corona (AMD MI50) and Zaratan (NVIDIA H100) for context with older AMD and newer NVIDIA hardware, respectively. For Frontier's MI250X GPUs, we run on one Graphics

Compute Die (GCD) which is an independent unit of allocation.

Table 4.1: Architectural details of the GPUs in each system used in this chapter. Flop/s and bandwidth values are theoretical peaks provided by device manufacturers.

System	GPU Model	Tflop/s*	DRAM b/w (GB/s)
Summit† (ORNL)	NVIDIA V100	14.0 (7.0)	900
Perlmutter (LBL)	NVIDIA A100	19.5 (9.7)	1555
Zaratan (UMD)	NVIDIA H100	67.0 (34.0)	3350
Corona (LLNL)	AMD MI50	13.3 (6.6)	1000
Frontier‡ (ORNL)	AMD MI250X	23.9 (23.9)	1600

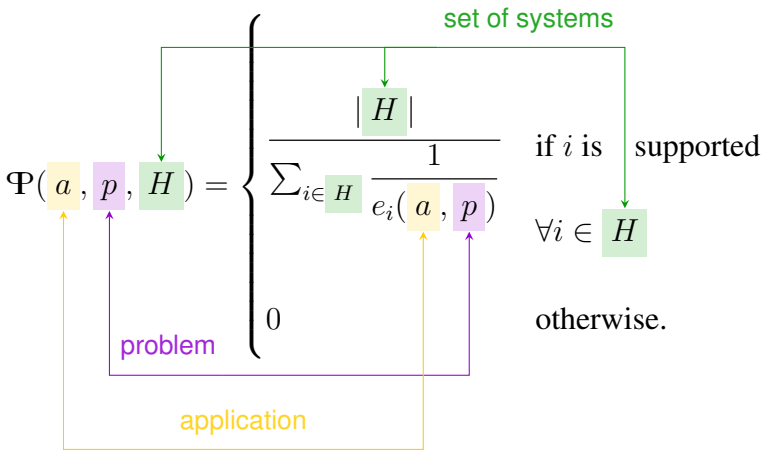
* Single-precision, double-precision in parentheses.
† We use the high-memory GPUs on Summit.
‡ Details for a single GCD of one MI250X.

4.2.4 Measurement and evaluation strategy

In this study, we modify applications where needed to consider both the efficiency of GPU kernel(s) and that of data movement between host and device needed to run the application. However, as will be discussed in Sec. 4.5, the impact of data movement on overall performance is minimal for these applications and not presented in detail. We add a runtime option to all the applications to specify a number of warmup iterations at the start of the simulation which we exclude from timing. XSBench normally runs only for only a single iteration, so we add a loop that repeatedly runs the kernel a user-specified number of times to ensure consistency across applications. As will be mentioned in Sec. 4.4, variability across runs is low, with runs of a given setup differing by at most 3.3%.

Having determined how to consistently define performance for each application, we can also derive additional higher-level metrics about performance portability for each combination of application and programming model. In this work, we use $\mathcal{P}$ **with application efficiency**

proposed by Pennycook et al. [73]. Φ is defined, for some application a, problem p, set of systems H, and measure of application efficiency e, as:



This is the harmonic mean of the efficiencies of an application running the same input problem across a set of systems. The application efficiency $e_i(a, p)$ of an application a solving problem p is the ratio $\frac{t_{min}}{t}$, where t is the runtime of a solving p on the particular hardware i, and t_{min} is the best observed runtime across all variants of a solving p on i. Φ ranges from 0 to 1, where 1.0 indicates the application runs at the best observed performance on all systems.

4.2.5 Automation and reproducibility strategy

In our experiments, we ensure that compilers, dependency versions, and flags are used consistently across applications and systems. We accomplish this with Spack [36], a popular HPC package manager. We create a single Spack environment file for each system which specifies the exact compiler, application, and library dependency versions along with any needed flags. As listed in Table 4.2, we have created or updated Spack package files for each proxy app, and these updates will be provided to the community. Our Spack environments for this project can be easily adapted to any new system, allowing for easy reproduction of our experiments, and significantly

reducing the extremely time-consuming effort of building every combination of application and programming model.

We further employ Spack's Python scripting tools[1] to develop robust automation for our experiments — we can create jobs with a single-line invocation leveraging Spack's spec syntax to adjust which application, models, or compilers are used, and save profile data to disk to be directly read by our plotting scripts. These scripts and environments will be published to allow the community to use our portability study methodology. These infrastructural contributions dramatically reduce the effort required to reproduce our results and create new studies of portable programming models.

[1] `https://spack-tutorial.readthedocs.io/en/latest/tutorial_spack_scripting.html`

Table 4.2: Summary of proxy applications and benchmarks used in this study as well as which programming model ports and Spack packages are updated or created by the authors. Here, E = already exists, **M** = modified by us, **C** = created by us.

Proxy Application	Scientific Domain	Method(s)	Suite	CUDA	HIP	SYCL	Kokkos	RAJA	OpenMP	OpenACC	Spack Pkg.
BabelStream	N/A	Bandwidth benchmark	N/A	E	E	E	E	**M**	E	E	**M**
XSBench	Nuclear physics	Monte Carlo	ECP	E	E	**M**	**C**	**C**	E	**C**	**M**
CloverLeaf	Hydrodynamics	Structured grid	Mantevo	E	E	**M**	E	**C**	E	**C**	**M**
su3_bench	Particle physics	Structured grid, dense lin. alg.	NERSC	E	E	E	E	**C**	E	E	**C**
miniBUDE	Molecular dynamics	N-body	N/A	E	E	**M**	E	**M**	E	E	**C**

Table 4.3: Compilers and versions used for building different programming model implementations on each system.

Prog. Model	Summit	Perlmutter	Corona	Frontier
CUDA	GCC 12.2.0	GCC 12.2.0	N/A	N/A
HIP	N/A	N/A	ROCmCC 5.7.0	ROCmCC 5.7.0
SYCL*	DPC++ 2024.01.20	DPC++ 2024.01.20	DPC++ 2024.01.20	DPC++ 2024.01.20
Kokkos	GCC 12.2.0	GCC 12.2.0	ROCmCC 5.7.0	ROCmCC 5.7.0
RAJA	GCC 12.2.0	GCC 12.2.0	ROCmCC 5.7.0	ROCmCC 5.7.0
OpenMP†	NVHPC 24.1	NVHPC 24.1	LLVM 17.0.6	LLVM 17.0.6
OpenACC	NVHPC 24.1	NVHPC 24.1	Clacc 2023-08-15	Clacc 2023-08-15

* We use AdaptiveCpp 23.10.0 for SYCL CloverLeaf due to performance improvement.

† We use ROCmCC 5.7.0 for OpenMP su3_bench on AMD GPUs due to performance improvement.

4.3 Porting to Unsupported Programming Models

The proxy applications we choose have implementations in most of the evaluated programming models. In these existing ports, we make minor modifications to consistently align timing measurements across different programming models. We also update the RAJA ports of BabelStream and miniBUDE to use Umpire for portable memory allocations.

When creating new ports, we seek to apply the same level of effort for all of them in order to avoid granting an unfair advantage to any particular implementation arising from excess optimization. We spend similar amounts of time implementing each new port, and keep the structure of the code between new and existing ports as similar as possible. Further, we specifically do not tune kernel grid size, block size, and shared memory per block. For programming models that require the user to specify these values (CUDA, HIP, RAJA, SYCL), we use the default values provided by the respective proxy application developers. For programming models that can select their own default parameter values (OpenMP, OpenACC, Kokkos), we allow the model to do so if compatible with the existing application code. Our results reflect "out of the box" performance that a user would encounter with minimal porting effort.

In the following subsections, we discuss our experiences working with the programming models as applicable. Table 4.2 summarizes our development efforts. We plan to merge these contributions to their respective upstream repositories.

4.3.1 Porting to Kokkos

Porting the XSBench code to Kokkos requires converting the existing `for` loop to be a lambda function passed into a `Kokkos::parall-el_for` call and converting the data struc-

tures to be used in Kokkos calls to `Kokkos::Views`. For example, XSBench's `SimulationData` struct contains several dynamic arrays which need to be Views in order to work on the GPU. In this situation, there are two options available to a developer: 1) rewrite all of the application code to use Views from the beginning, including any CPU-side setup or initialization; or 2) avoid rewriting the any setup code by constructing Views out of pointers to any ordinary C++ arrays after initialization but before copying them to the device and launching kernels.

We opted for the second of these methods to minimize changes to existing application code. Listing 4.1 provides an example of this approach as we implemented it. In summary, we construct an unmanaged View in the `HostSpace` called `u_cocns` using the heap memory of the `SD.concs` array, construct a new View in the device space called `SD.d_concs`, and finally `deep_copy` the unmanaged host View to the new device View. While Kokkos requires developers to use its memory abstraction, the View, in order to make use of its portable kernel abstraction, we demonstrate how an application developer looking to work incrementally can minimize changes to application code while gaining the portability benefits of Kokkos.

```
1 View<double*, LayoutLeft, HostSpace, MemoryTraits<Unmanaged>>
2     u_concs(SD.concs, SD.length_concs);
3 SD.d_concs = new View<double*>("d_concs", SD.length_concs);
4 deep_copy(*SD.d_concs, u_concs);
```

Listing 4.1: Example of converting a C++ dynamic array to a device View for incremental development, where SD is a struct containing XSBench simulation data.

4.3.2 Porting to RAJA

In contrast to Kokkos, the RAJA portability ecosystem uses multiple libraries to provide portability. Briefly, the RAJA library itself provides C++ lambda-capturing to allow developers to express portable computation. For memory management, the developer can either write or use a custom portable memory management library, or use the related Umpire [6] library, which provides portable memory allocation primitives and memory pools. This separation of concerns in the RAJA ecosystem provides greater capability for incremental porting of an existing codebase (i.e., portable compute first, then portable data structures), avoiding more extensive refactoring.

In our case, we opt to take advantage of Umpire for CloverLeaf and XSBench, which both have extensive existing code for managing and initializing data structures. However, we encounter several challenges building the RAJA applications. Relying on multiple independent libraries increases the expertise required and frequency of errors in setting up build systems, a process that is already complicated for a single library containing device kernels. Package managers such as Spack [36] can mitigate these problems for end users, although this solution pushes the responsibility of ensuring the libraries build and install correctly onto the package maintainers.

4.3.3 Porting to OpenACC

OpenMP ports already exist for all applications, so creating similar OpenACC ports where needed just requires a one-to-one conversion of the relevant OpenMP pragmas to OpenACC. For example, `omp target teams distribute parallel for` becomes `acc parallel loop`. This rote method makes our experience with porting XSBench and CloverLeaf from

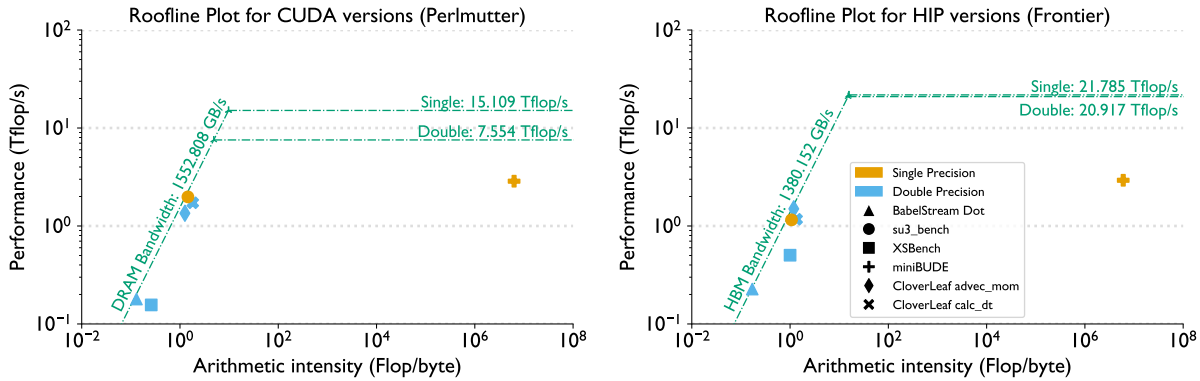


Figure 4.1: Roofline plots for the most time-consuming kernel in the CUDA (left) and HIP (right) versions of each application, from runs on Perlmutter (NVIDIA A100) and Frontier (AMD MI250X GCD) respectively. Orange points are single precision, and blue points are double precision. For each application, we plot the predominant precision used.

OpenMP to OpenACC very productive. In contrast to Kokkos and RAJA, working with existing data structures is highly transparent in OpenACC, so long as the structures are plain old data (POD) and do not contain pointers to CPU memory internally. In those more advanced cases, which we do not encounter in this work, users must write more complex directives to handle such data structures, convert them to simpler formats, or use automatically managed memory if provided by the GPU device [98].

4.4 Experimental Setup

In this section, we describe the setup for the experiments conducted in this work. We run all the applications on all five systems selected (listed in Table 4.1).

Table 4.3 lists the compilers used with each programming model alongside their versions. We use GCC 12.2.0 as the host compiler on NVIDIA systems and ROCmCC 5.7.0 on AMD. We use CUDA version 12.2 on NVIDIA systems, and HIP 5.7.0 on AMD systems, as well as Kokkos version 4.2.00 and RAJA v2023.06.1. OpenACC, OpenMP, and SYCL all have different

implementations provided by multiple compilers on the systems where we perform our experiments. We test all the available compilers for these models[2] and choose the best-performing compiler for each application, model, and system. We perform this compiler-choice tuning to reflect the fact that applications using these programming models will likely test their code with all working compilers, and use in practice the best-performing option. In all models except SYCL and OpenMP, the best-performing compiler is consistent across applications on each system. For the SYCL port of CloverLeaf, AdaptiveCpp is consistently superior, so we present AdaptiveCpp results for that application and DPC++ for all others. For OpenMP, ROCmCC wins on AMD systems for su3_bench and Clang wins for all other applications. Note also that we are unable to build CloverLeaf with Clacc due to lack of support for the `host_data` clause, and hence we cannot run CloverLeaf on AMD systems with OpenACC.

We compile all proxy applications with '-O3' as well as fast math flags and hardware specific instructions for approximate sqrt and division operations. For AMD systems we also add '-munsafe-fp-atomics' as we found this to be broadly beneficial to performance. Finally, for the Clacc compiler, we provide the flag '-fopenacc-implicit-worker=vector-outer' at the recommendation of a developer, as this flag will soon be enabled by default for Clacc.

We select input decks and command line inputs for each proxy application based on recommended settings from their respective developers. When given a choice of problem size, we select the largest representative problem available that fits on all tested GPUs. We also choose the number of iterations for each application to ensure about a minute of execution time, so as to reduce variability. Section 4.2.4 describes how we modify the proxy applications to ensure

[2]OpenMP: Clang, GCC, ROCmCC, NVHPC, CCE; OpenACC: Clacc, GCC, NVHPC; SYCL: DPC++, AdaptiveCpp

consistent timings. We present the final command line arguments in Table 4.4.

Table 4.4: Input parameters to the proxy applications.

Application	Input parameters
BabelStream	`-n 1500 -w 150 -s $((1<<29))`
XSBench	`-s large -m event -G hash -n 150 -w 15`
CloverLeaf	`--in clover_bm64_mid.in -w 52`
su3_bench	`-l 32 -i 100000 -w 10000`
miniBUDE	`--deck bm2 -p 2 --wgsize 128 -i 10 --warmups 1`

Note that for all cases tested the time spent in data movement is negligible (less than 2%) compared to time spent in device kernels, so our result figures present **only** GPU kernel time. For all performance results presented we run the application three times and present the average result. Variability is low; the largest range of times recorded as a percentage of mean runtime for a case is 3.3%, and the mean is 0.1%. We report total runtime for BabelStream kernels rather than memory bandwidth in order to ensure that "lower is better" across all performance results we present. The values collected can be converted to bandwidth (GB/s) by dividing the total data moved by the time.

4.5 Results and Discussion

We first present a roofline analysis of the native port implementations of each application to understand their compute and memory behavior. Next, we present the results of our model comparisons for individual applications and then across all systems and applications.

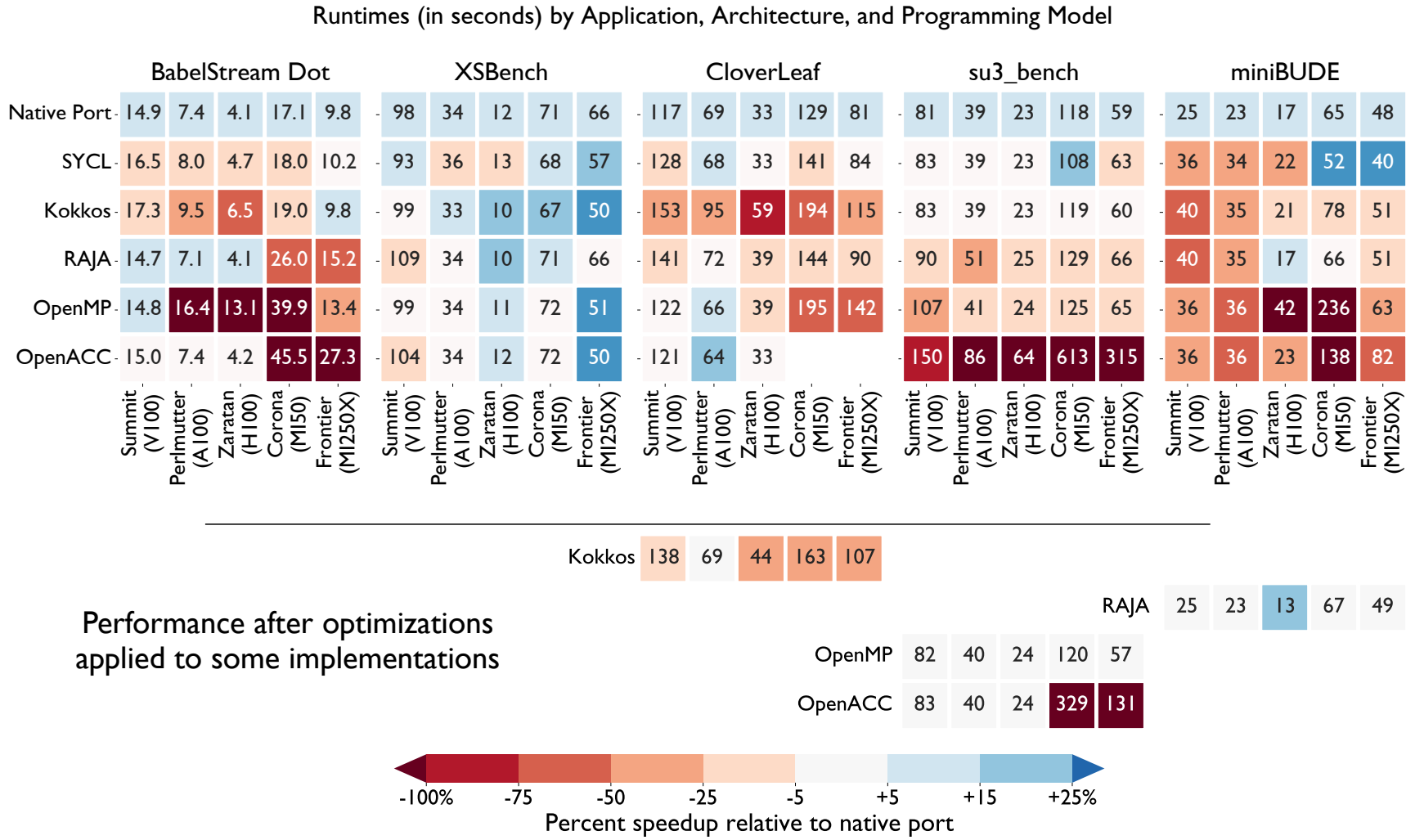


Figure 4.2: Execution time (shown as raw numbers in each cell) over three trials of all proxy applications across all systems and programming models (lower is better). Color of each cell indicates performance improvement or degradation relative to the native port for that system and application (blue is faster and red is slower than the native port). Execution times for optimized versions of the last three applications are shown below the horizontal line.

4.5.1 Roofline analysis

Figure 4.1 provides the empirical rooflines for the NVIDIA A100 GPU on Perlmutter and AMD MI250X GCD on Frontier. It also plots the positions of the most time-consuming kernels in the CUDA and HIP implementations of the five proxy applications. For BabelStream, this is the `dot` kernel, and for CloverLeaf, this is `advec_mom`.[3] We also plot `calc_dt`, as it has the longest per-invocation execution time in CloverLeaf. miniBUDE, XSBench, and su3_bench contain a single computational kernel each. We plot each kernel for the predominant floating-point precision used. We observe that all kernels evaluated are memory-bound except for miniBUDE, which is highly compute-bound, on both architectures. Among the memory-bound apps, on both systems BabelStream `dot` is the most memory-bound (i.e., furthest to the left). This is expected given that BabelStream is a memory bandwidth benchmark. CloverLeaf and su3_bench are much closer to the knee point on both systems, while XSBench has substantially different arithmetic intensity on both systems — 0.26 on Perlmutter, 1.00 on Frontier. It is possible that XSBench heavily utilizes some instruction types that are accounted differently between NVIDIA and AMD's counters used for roofline plotting. Except for XSBench and miniBUDE, all of these kernels are relatively close to the roofline, suggesting these CUDA and HIP versions are relatively close to optimal for the algorithms they implement.

Table 4.5 provides additional details about the kernels compared. We provide cyclomatic complexity (CC) to reflect control flow complexity in the kernels and the number of loops nested in the main loop to reflect dimensionality of potential parallelism. The number of vector arguments to the kernel partially reflects the range of distinct memory locations potentially refer-

[3] `advec_cell` and `PdV` are also highly time-consuming, but have similar positions on the roofline.

enced by the kernel. Static instruction count and registers per thread are both taken from Nsight Compute profiles of each kernel on Perlmutter, and reflect the length of the kernel and register pressure.

Observation 1

Most major kernels examined in this work are memory-bound with the exception of miniBUDE, which is strongly compute-bound.

4.5.2 Analysis of Individual Applications

Next, we present performance results for BabelStream `dot`, XSBench, CloverLeaf, su3_bench, and miniBUDE in Figure 4.2. Each heatmap cell represents the **total execution time** across all kernel invocations in each application. Note that each value is the mean of three separate runs of each case, with maximum difference between any run and the three-run mean at 3.3% (as described in Section 4.2). Also, while we do measure data movement time, we do not report it here, as it is consistently negligible (<2% of runtime) compared to the time spent in the GPU kernels. The "Native Port" row in each plot represents CUDA performance on Summit and Perlmutter (the NVIDIA systems) and HIP performance on Corona and Frontier (the AMD systems). We will discuss observations derived from each mini-app in turn.

4.5.2.1 BabelStream

BabelStream contains five kernels: `copy`, `add`, `mul`, `triad`, and `dot`. All of these kernels are simple one-line memory-bound operations. `dot` is unique in that it utilizes a reduction operation to compute a dot product, making it a useful simple benchmark of reduction operations across programming models. We observe that for all kernels except `dot`, performance across

programming models is highly consistent with the native port. Figure 4.3 shows the performance of the `add` and `copy` kernels across implementations on Frontier and Perlmutter as a demonstration. OpenACC on AMD systems is the only significant outlier, likely arising from overhead of the Clacc compiler translation approach.

Table 4.5: Key details of the major kernels in each proxy application used in the chapter. CC is cyclomatic complexity and LN indicates the level of nesting in the kernel's main loop. Static instruction count and registers per thread are collected from NCU profiles of the CUDA implementations on Perlmutter.

App.	Kernel	CC	LN	# Vector Args.	# Static Insts.	Reg./ thread
BabelStream	copy	1	1	2	12	16
BabelStream	dot	5	2	3	48	16
XSBench	xs_lookup	39	3	9	670	49
CloverLeaf	advec_mom	4	2	5	405	43
CloverLeaf	calc_dt	8	3	15	643	47
su3_bench	k_mat_nn	3	4	3	59	26
miniBUDE	fasten	29	4	10	357	62

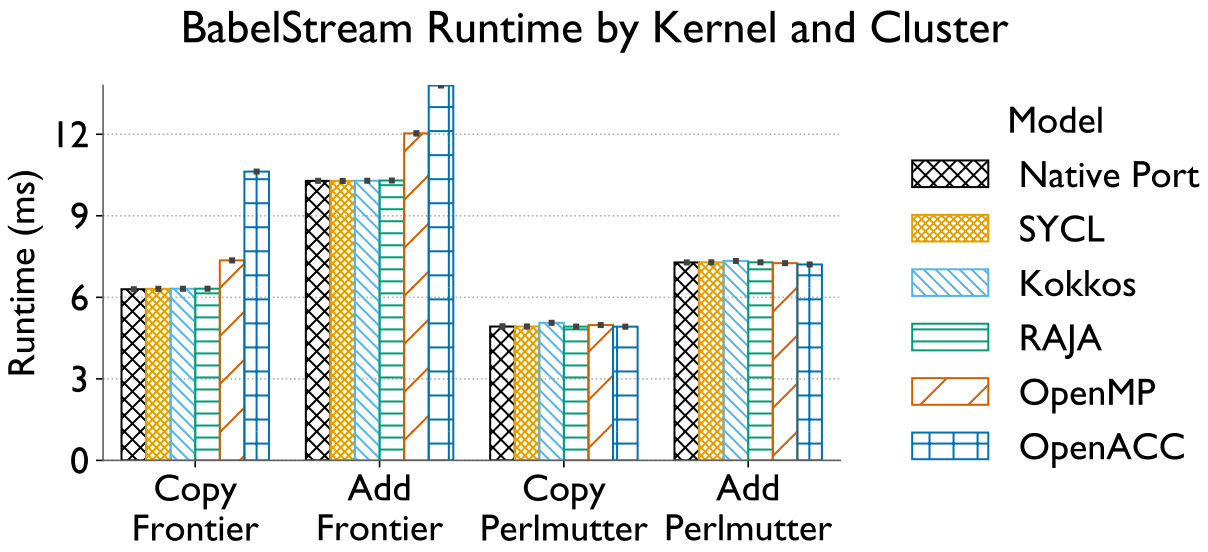


Figure 4.3: Execution time by model for Copy and Add kernels in BabelStream on Frontier and Perlmutter.

Performance in `dot` is in contrast much more variable, which is why it is the only one

reported in Fig. 4.2. Usually, the portable programming models offer similar or worse performance than the native port. Kokkos performs moderately worse than the CUDA on Perlmutter and Zaratan, while RAJA performs moderately worse than HIP on Corona and Frontier. More notably, OpenMP performs significantly worse than the native port on Perlmutter, Zaratan, and Corona, and moderately worse than HIP on Frontier. OpenACC performs significantly worse than HIP on both AMD systems.

In one notable exception, for RAJA BabelStream `dot` on Perlmutter, we observe that RAJA takes advantage of warp-level primitives in addition to shared memory to perform the reduction, maximizing utilization of hardware-specific features for such operations. This allows RAJA to achieve moderately improved performance over CUDA on all NVIDIA systems.

Observation 2

Simple BabelStream kernels (other than `dot`) are dominated by memory-bound operations, and all programming models can provide performance portability for them, except OpenACC on AMD.

Observation 3

For `dot`, a reduction kernel, SYCL comes close to providing performance portability across all systems, and RAJA and OpenACC provide good performance on NVIDIA systems. OpenMP and OpenACC really struggle to provide performance portability for `dot`.

4.5.2.2 XSBench

XSBench runs a single, long kernel which performs a large quantity of binary searches. In this case, all programming models achieve near or moderately better performance than the native port. In some cases, particularly on Frontier for all models except RAJA, the portable

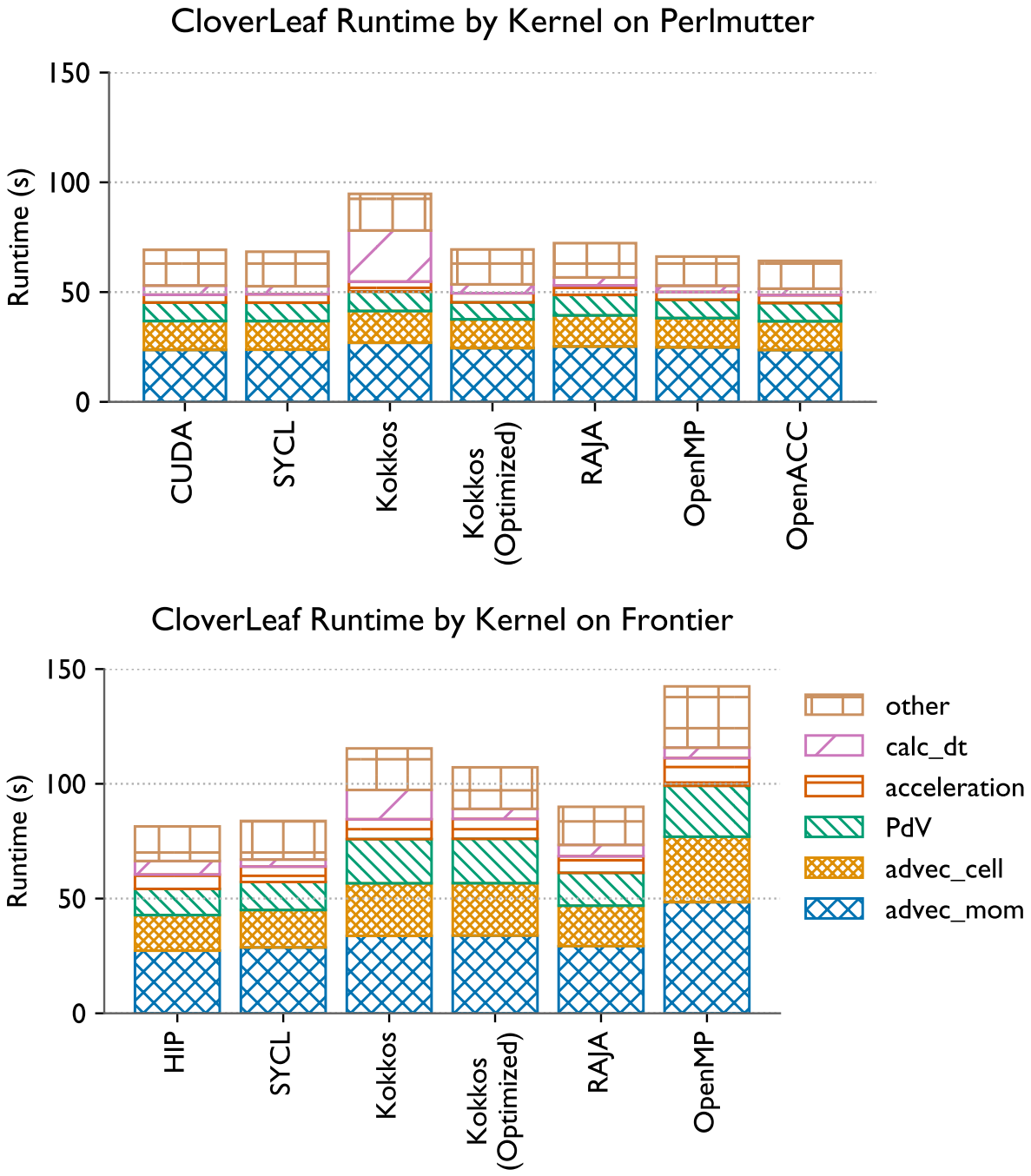


Figure 4.4: Execution time by model broken down by kernel for CloverLeaf runs on (top) Perlmutter and (bottom) Frontier.

programming models outperform HIP. Using Omniperf to profile XSBench, we observe that the HIP port achieves lower Gflop/s and lower L1 cache bandwidth, while Kokkos uses a larger workgroup size and arranges L1 cache read requests in a larger number of smaller requests for

a similar number of bytes. This suggests Kokkos is selecting a more ideal workgroup size and arranges data access patterns more efficiently for AMD GPUs in XSBench. Meanwhile, OpenMP appears to be able to take advantage of Local Data Share (LDS) implicitly, reducing stalls for accesses to memory, while HIP is not able to.

XSBench is a performance test case used in the development of LLVM OpenMP offloading, which Clacc also uses for OpenACC on Frontier, helping explain why both directive-based models perform so well with XSBench. However, given that Kokkos is a C++ abstraction over HIP code, it is surprising that it can outperform HIP. We note that HIP XSBench performance on Frontier is only slightly better than HIP XSBench on Corona, suggesting that the XSBench HIP implementation is not a fully optimized and mature baseline.

Documentation for XSBench indicates that developers used the Hipify tool to create the XSBench HIP port, and in comparing the HIP and CUDA versions it is clear that they are identical aside from simple substitution of CUDA syntax for HIP syntax. The XSBench kernel is also notably more cyclomatically complex and longer than other kernels we examine (Table 4.5). Together, these observations suggest that HIP kernels with more complex control flow translated directly from CUDA without additional optimization may not guarantee optimal performance on AMD GPUs, which have significantly smaller cache capacity per thread workgroup relative to NVIDIA GPUs. More broadly, the case of XSBench demonstrates how portable programming models are able to achieve matching or even superior performance for more complex kernels with a similar level of development effort as compared to vendor programming models.

Observation 4

All programming models can achieve competitive performance portability for XSBench, a kernel with relatively complex control flow where the implementation style between portable and native ports is highly similar.

4.5.2.3 CloverLeaf

As a larger proxy application with many kernels including structured stencil operations as well as a reduction operation in `calc_dt`, CloverLeaf encompasses multiple types of GPU kernels. Nevertheless, several programming models achieve broadly consistent performance compared to native ports on both NVIDIA and AMD GPUs. SYCL in particular achieves slightly better performance than CUDA on Zaratan. OpenACC slightly outperforms CUDA on both Perlmutter and Zaratan, mostly due to improved performance in smaller kernels like `calc_dt`, but unfortunately we are unable to compile CloverLeaf with OpenACC on AMD systems at this time due to lack of support for the `host_data` clause. RAJA performance is slightly worse than native ports across systems.

Fig 4.4 breaks down CloverLeaf performance into major constituent kernels to illustrate where performance differences arise for outlier cases. First, CloverLeaf performance for OpenMP on Frontier is a notable outlier. We find that compared to HIP the OpenMP port spends significantly more time in the `advec_*` and `PdV` kernels. OpenMP achieves less than half the L1 cache bandwidth in this kernel, as well as a roughly 40% lower L2 cache hit rate and 30% higher rate of stalls on L2 cache data, relative to HIP.

Kokkos performance in CloverLeaf is also a notable exception. We observe that the Kokkos

port of CloverLeaf spends longer in the `calc_dt` reduction kernel relative to other ports, particularly on NVIDIA systems, like Perlmutter. In Nsight Compute, we find that the Kokkos port achieves fewer eligible warps on average, mostly due to barrier warp stalls, which we do not observe in the other ports. Comparing the implementations of the `calc_dt` between ports, we find that Kokkos is the only one to use a 2D reduction instead of collapsing the kernel into a 1D reduction. We adjust the Kokkos port to use a 1D scheme, bringing Kokkos `calc_dt` performance closer to the native port on all studied systems, and no longer observe barrier warp stalls in the new profile. As shown in Figure 4.2, Kokkos CloverLeaf performance improves on all systems with this change. The benefit is greater on NVIDIA, where Kokkos's performance on the other three significant kernels compares more favorably with the native ports.

Observation 5

For CloverLeaf's memory-bound kernels, which employ stencil operations, SYCL and RAJA can consistently provide performance portability and OpenMP, Kokkos and OpenACC struggle with providing portable performamce.

4.5.2.4 su3_bench

su3_bench is a single-kernel proxy application that computes a matrix multiplication on complex numbers with a relatively deep nested loop heirarchy (Table 4.5). Generally, performance across programming models is fairly consistent, with the obvious exception of OpenACC before our improvements.

The OpenACC port for su3_bench in particular suffers from insufficient exposed parallelism, even on NVIDIA GPUs. The su3_bench OpenACC port originally generates code with only 36 threads per block, despite iterations being assigned to blocks of size 128. This leads to

fewer active threads per block. We address this issue by collapsing all four loops, exposing more parallelism.

We also find that both OpenMP and OpenACC generated twice as many global loads and stores as CUDA, due to a misaligned complex number struct. [4] We declare this struct aligned to sizeof(T) * 2, resulting in a single load and store for each complex number in the array. On AMD this optimization has no effect. As presented in Figure 4.2, OpenACC benefits strongly from this combination of optimizations, whereas OpenMP achieves modest speedups.

For RAJA, we observe substantially lower arithmetic intensity in L1 and L2 cache compared to native ports, suggesting the RAJA port loads unnecessary data from memory more often, although the overall impact on performance portability of this limitation is relatively low.

Observation 6

For su3_bench, a memory-bound kernel with a deep loop nest, almost all programming models can achieve approximate performance portability as long as sufficient parallelism is exposed and struct declarations are aligned, with the moderate exception of RAJA and stronger exception of OpenACC.

4.5.2.5 miniBUDE

miniBUDE, a single-kernel app, is by the the most challenging proxy application for the programming models we test. miniBUDE is unusual compared to our other proxy applications in that it is highly compute-bound, and leverages thread coarsening to hide memory latencies using instruction-level parallelism within the kernel. It also exerts the highest register pressure in the native CUDA implementation relative to other kernels we examine (Table 4.5). No programming

[4] OpenMP, OpenACC, and SYCL do not provide a native complex type for GPUs.

model is able to acheive consistent performance with the native port, except for RAJA after our improvements. We note that SYCL does achieve superior performance compared to HIP on AMD devices.

In comparing the Kokkos, RAJA, SYCL, and CUDA versions of miniBUDE, we notice that the RAJA version is not making use of shared memory, while the Kokkos, SYCL, and CUDA ports are. RAJA recently added features for dynamically allocating shared memory inside a kernel, a feature needed in miniBUDE since the forcefield data is input-dependent in size, so we modify RAJA miniBUDE to use shared memory for this data.

This optimization improves RAJA performance on NVIDIA systems, with little impact on AMD, leading to an overall increase in portability (see Fig. 4.2). After the change RAJA performance comes very close to the CUDA performance on Perlmutter and 27% faster than CUDA on Zaratan, an impressive gain since other models already using shared memory do not get this close on NVIDIA systems. At the time of writing we are unable to add dynamic shared memory allocation inside the kernel for the OpenMP and OpenACC ports due to lack of support.

The OpenMP port of miniBUDE appears to allocate an order of magnitude more Local Data Share (LDS) bytes than HIP does, limiting the number of active compute units and thus reducing the degree to which memory access latency can be hidden. Comparing the OpenMP port to CUDA, we see a significant increase in registers used per thread (86 vs. 62) and dramatically more static instructions (1463 vs. 357). Other models show similar issues, but less exaggerated: for example, Kokkos uses 69 registers per thread and generates a 797-instruction kernel. For both Kokkos and OpenMP these overheads correspond to a 500% increase in cycles spent in L2 cache activity as well as 50% increases in DRAM and L1 cache cycles.

Observation 7

For miniBUDE, a highly compute-bound kernel relying on shared memory, RAJA (after adding shared memory support) can provide very competitive performance portability where most other models, especially OpenMP and OpenACC, struggle due to register pressure and increased generated instruction counts. SYCL is also highly competitive, but only on AMD systems.

4.5.3 Evaluating performance portability across applications after optimizations

From analysis of Figures 4.2 after our optimizations, we can make several general observations about the performance portability enabled by each programming model.

4.5.3.1 Language extensions

CUDA is the best or within 3% of the best performing model in eleven out of fifteen cases. For these applications, this is a useful validation of the maturity of the CUDA baseline for each application, and confirms our expectation that the low-level vendor model would be the most performant and portable across GPUs from the same vendor.

Meanwhile, for most cases on AMD systems, including CloverLeaf, BabelStream `dot`, and su3_bench on Frontier, AMD's HIP programming model achieves the best performance, as expected. However, in multiple instances, HIP does not achieve the best performance, particularly for XSBench, as discussed under that application.

Observation 8

On NVIDIA systems, CUDA almost always performs at or near the best observed performance, while on AMD systems, while HIP performs best in many scenarios, there are some cases, in particular for XSBench, where other models are faster than HIP.

Finally, SYCL performs better than HIP in five out of ten cases on AMD systems. As a lower-level language extension, similar to CUDA or HIP, this is not necessarily surprising. In some cases, SYCL is able to improve on CUDA or HIP performance, and even where SYCL is more than 3% slower than a native port, it is never the worst-performing port except in XSBench on Perlmutter and Zaratan, where it is is only 5.3% and 10% slower, respectively. SYCL is the fastest non-native programming model in more cases than any other model, at eleven out of twenty-five total application and system pairs, and six of these are on AMD systems.

Observation 9

SYCL performance is often competitive with CUDA and HIP, and relatively stable across system and application pairs, with the exception of miniBUDE on NVIDIA GPUs.

4.5.3.2 C++ abstraction libraries

Kokkos and RAJA compare favorably with CUDA and HIP on NVIDIA and AMD systems, with one of the two ports either nearing or exceeding the native port's performance on every combination of system and app, besides those involving CloverLeaf on any system or miniBUDE on Summit and Perlmutter. While which model is more performant is very application-dependent, we can observe that RAJA tends to perform more competitively for NVIDIA systems, and Kokkos tends to have an advantage on AMD systems.

Observation 10

Kokkos and RAJA are competitive with CUDA and HIP on many system and application pairs, with a slight preference for RAJA on NVIDIA GPUs and Kokkos on AMD GPUs.

4.5.3.3 Directive-based models

OpenMP performance can be slower than the native baseline, achieving significantly better performance than the baseline only for XSBench on Frontier. OpenMP is able to achieve rough parity with the native baseline in twelve out of twenty-five cases.

On NVIDIA systems, OpenACC generally achieves more consistent performance with the baseline, but is often worse than OpenMP and further worse than HIP on AMD systems, likely because it is employing the same LLVM OpenMP offloading runtime through the Clacc compiler. Per Clacc developers, there is some overhead due to suboptimal translation of OpenACC to OpenMP within Clacc which will be addressed in a future release.

Observation 11

OpenMP is slower than other implementations in roughly half our cases, and OpenACC struggles with AMD systems.

4.5.4 Performance portability metric evaluation

Figure 4.5 displays the Φ metric for each programming model and proxy application combination after applying the optimizations described above. The "Native Port" column provides context, indicating what the metric would report if a team decided to maintain both a HIP and CUDA version of the application. We are unable to run CloverLeaf with OpenACC on AMD sys-

tems, so that cell is zero per the official formulation of the metric[5]. According to $\mathcal{P}$, we observe a moderate preference for SYCL, RAJA, and Kokkos as performance portable programming models, in roughly that order, and for OpenMP over OpenACC within directive-based models.

Observation 12

Summarizing our study with $\mathcal{P}$, we find that SYCL most consistently achieves performance portability for our tests, followed closely by RAJA and Kokkos.

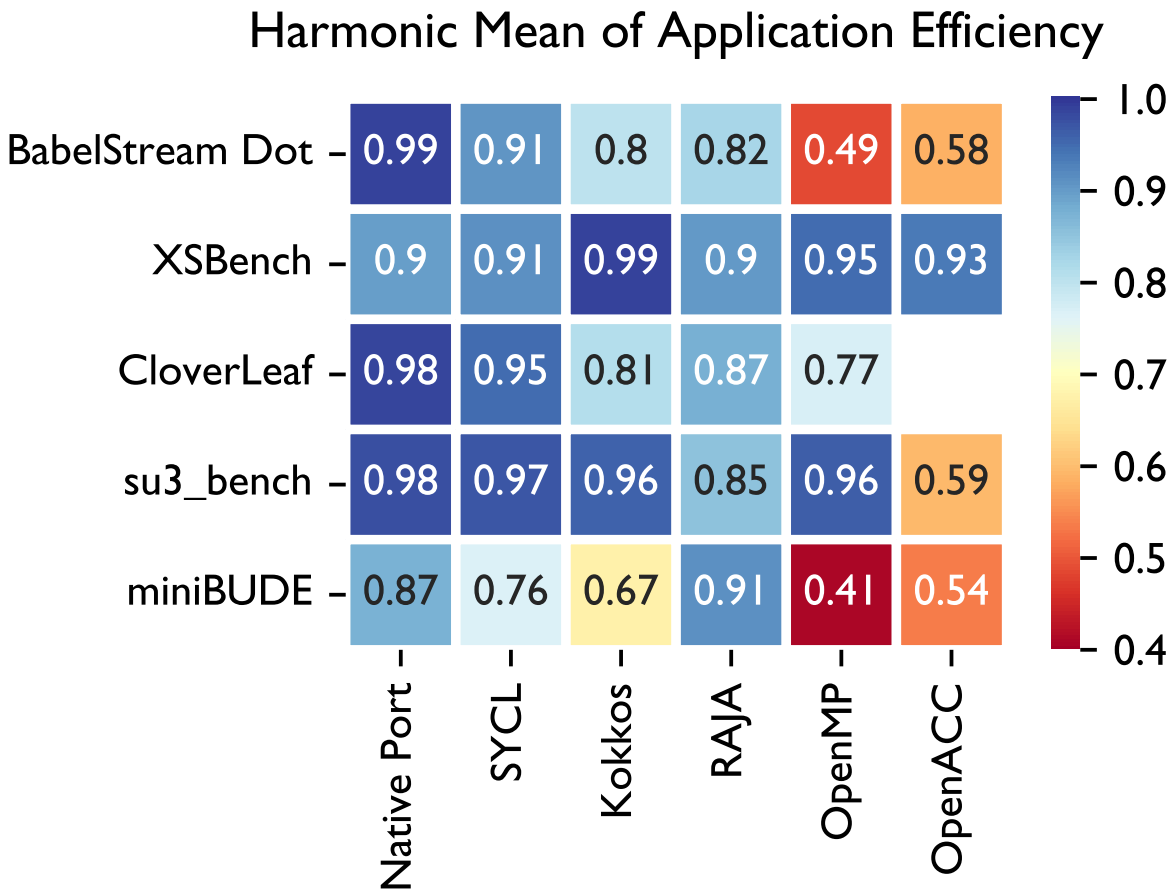


Figure 4.5: $\mathcal{P}$ of GPU kernel performance for each programming model and application combination, after optimizations. Applications are listed in ascending order of arithmetic intensity from top to bottom. Note for OpenACC we are unable to compile CloverLeaf on AMD systems.

[5]For the subset of systems (Summit, Perlmutter, and Zaratan) we are able to run OpenACC on, the value is 0.98.

Chapter 5: PAREVAL-REPO: Benchmarking LLM Repository-Scale Translation Between Parallel Programming Models

5.1 Introduction

High performance computing (HPC) hardware has increasingly diversified over the last decade, now including GPUs from multiple vendors alongside CPU-based systems. Portable programming models like Kokkos [94] and OpenMP [70] provide a solution to develop a single source code for multiple types of hardware, but challenges remain in productively converting existing code repositories to use them. The size and complexity of existing HPC and scientific code repositories means that manually converting all relevant portions of the codebase to use a new programming model is extremely time-consuming, as well as potentially prone to error and the introduction of performance issues.

Meanwhile, large language models (LLMs) have shown significant promise in automating simple programming tasks. While prior work has shown that generating parallel code for a task from scratch proves to be a challenge for current LLMs, they are capable of translating existing serial or parallel code into a particular parallel programming model [66]. However, these benchmark cases are translations of single, small functions or kernels. Scientific applications are often composed of many such functions and kernels, and solve more complex problems. Translating

an entire software repository introduces the complexities of data structure design, object hierarchies, and coordination of work between multiple parallelized functions or kernels. If LLMs can automate some of this work with minimal human intervention, the boon to developer productivity would be immense. Nevertheless, whether LLMs are capable of translating entire software repositories, including build systems, headers, and multiple functions, between parallel programming models, remains an open question.

In this chapter, we attempt to answer the question posed above by proposing PAREVAL-REPO, a suite of benchmarking cases for LLM-based automatic repository-scale translation between parallel programming models. This benchmark suite includes a range of standalone scientific computing and artificial intelligence programs in a range of codebase sizes, ranging from hundreds to thousands of lines. The evaluation tasks include multiple programming model translations – CUDA to OpenMP offloading, CUDA to Kokkos, and OpenMP threading to OpenMP offloading. All but one of these cases are chosen specifically to ensure that no publicly-available translation in the target programming model exists, to prevent the LLM from simply reciting code memorized during training rather than reasoning through and solving an unseen problem.

PAREVAL-REPO enables us to evaluate the capabilities of different LLMs in correctly performing repository-scale translation, and enables future researchers to design and test novel techniques for improved LLM-based translation. We evaluate a non-agentic and a top-down agentic approach that can use a variety of open-source and commercial LLMs underneath, as well as a state-of-the-art SWE-agent [99] agentic tool for LLM-driven software engineering issue resolution. While some combinations of LLM and translation techniques succeed in translation for smaller applications, we show that repository-scale translation between portable programming models poses unique challenges beyond the previously-observed difficulties with parallel code

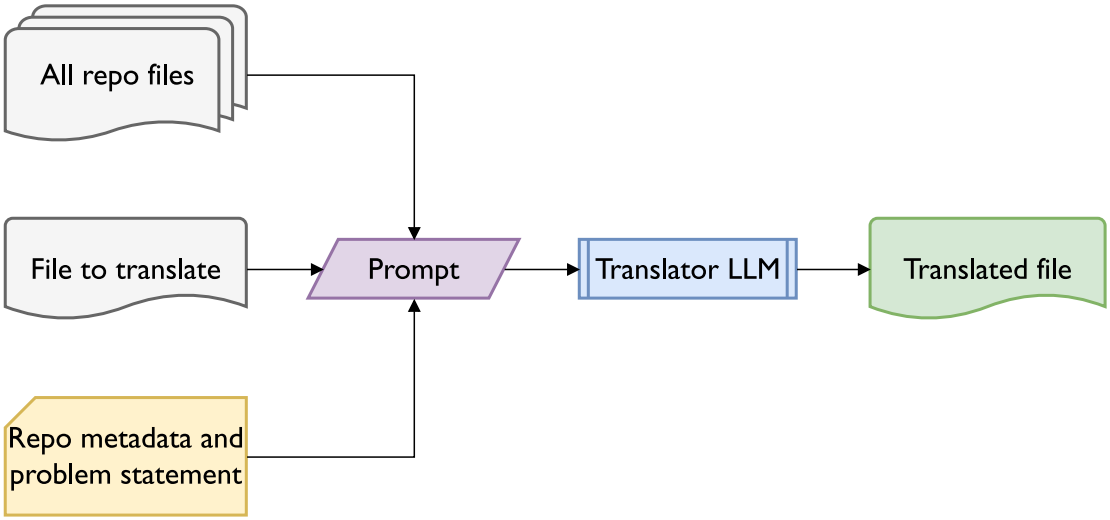


Figure 5.1: Control flow of non-agentic translation method, per file to translate.

generation. Most importantly, regardless of the translation technique or LLM used, creating working build systems and maintaining consistency of interfaces across files remain major hurdles. Finally, we propose a novel derived metric, expected token cost (E_κ), which predicts the total token cost required to achieve a correct translation, and use it to compare inference cost of translation techniques and LLMs.

5.2 Techniques for Repo-level Translation

In this section we detail the techniques we benchmark for translation of whole HPC software repositories.

5.2.1 Non-agentic method

The non-agentic strategy employs an LLM of the user's choice to translate an entire repository file-by-file. Figure 5.1 outlines the control flow for an individual file translation. Note that

as this approach is non-agentic, no context or information can pass between translations of separate files in the same repository. When translating each file, we provide as context the contents of all other untranslated files in the repository. Listing 5.1 shows a sample prompt for the file `main.cpp` in nanoXOR.

````
You are a helpful coding assistant. You are helping a software
developer translate a codebase from the CUDA execution model to the
OpenMP Offload execution model. Writing correct, fast code is
important, so take some time to think before responding to any query,
and ensure that the code you create is enclosed in triple backticks
(```), as used in the query below.

Below is a codebase written in the CUDA execution model. We are
translating it to the OpenMP Offload execution model. Here is the file
tree of the entire repository:

|-- Makefile
|-- README.md
+-- src/
    +-- main.cpp

Here is the code for each file in the codebase:

Makefile
...

src/main.cpp
````

```
...

Translate the src/main.cpp file to the OpenMP offload execution model.
Output the translated files in one code block. Assume .cpp filenames
whenever referring to other files as this will be a C++ code.
```

Listing 5.1: Sample prompt for non-agentic translation method.

To encourage accurate outputs, we begin with a system prompt indicating to the LLM that it is a helpful coding assistant, clearly indicating the overall task, and emphasizing that code should be correct and fast. Next, we provide the file tree for the full repository, and the full text of all the untranslated files in the repository. Finally, we indicate which file the LLM should translate, the execution model to translate into, and the filename extensions to assume where appropriate.

When prompting the LLM to translate files containing main functions and build system files like (Makefile or CMakeLists.txt), we provide an addendum to the prompt. For main function files, this addendum indicates the command line interface the application should respect. For build system files, the addendum indicates the interface the build system should respect, along with the compiler it should be compatible with.

5.2.2 Top-down agentic method

Directly translating an entire repository within the context window of an LLM works well for smaller repositories, but will not scale to larger repositories that do not fit in the context window. This means that the translation will need to split into smaller translation tasks, working with smaller portions of the full repository. Finding the best way to partition the data and work is non-trivial and various coding agents have been proposed that address these two issues with dif-

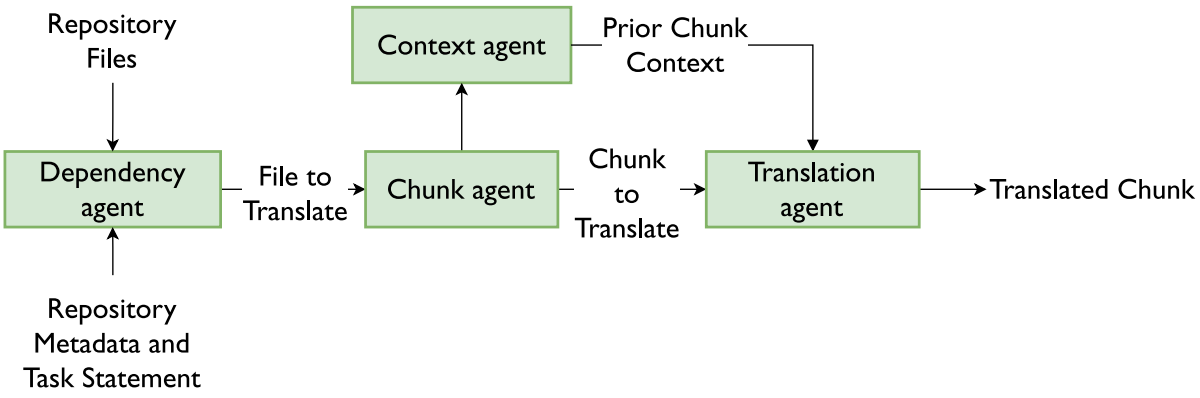


Figure 5.2: Control flow of the entire top-down agentic translation method.

ferent approaches. We employ a top-down agentic approach to translation, which is highlighted in Figure 5.2.

Our top-down agentic approach is composed of four LLM agents: chunk, dependency, context, and translation. Each of the agents is a mix of static code analysis tools and LLMs. The first agent that is run is the dependency agent. This agent is responsible for determining the dependencies between files in the repository. We translate files with no dependencies first, since they do not require any external context. We utilize the clang compiler to determine `#include` dependencies only, precluding the existence of circular dependencies. For non-C/C++ files or C/C++ files where clang fails, we use an LLM to analyze the file contents and repository structure and determine the dependencies.

Files are then translated one by one in the order set by the dependency agent. When translating files with dependencies we provide in the prompt a summary of the changes already made to the dependencies. For example, if the function `computeWithCuda()` is translated to `computeWithOpenMP()`, then any files that call `computeWithCuda()` need to be updated to call the new function. The context agent produces LLM generated summaries of translation

changes to pass down to future dependents.

The chunk agent is responsible for splitting files into smaller pieces that fit within the context window of the translation agent's LLM. The chunk agent is syntax-aware and splits files at the function level as much as possible, to minimize scope splitting across chunks. Finally, the translation agent is an LLM that translates code from one execution model to another.

5.2.3 SWE-agent

We employ SWE-agent, described in Sec. 2.2.3, as a technique for full-repository translation. To use SWE-agent for this translation, we need to slightly reconfigure our translation task into a format SWE-agent expects. SWE-agent addresses GitHub issues in git repositories, so we start by rephrasing the usual translation prompt from the Non-agentic approach as an issue, placed in a dedicated file that is passed to SWE-agent. We also create a simple .git directory in the application codebase to be translated to ensure SWE-agent sees the files it expects. Unfortunately, SWE-agent is designed primarily to work with Python repositories, and automatically replaces tabs with spaces, breaking Makefiles. This limits its usefulness for our benchmark suite.

Listing 5.2 shows the full SWE-agent invocation used for our tasks.

```
sweagent run
  --agent.model.name=gpt-4o-mini
  --agent.model.per_instance_cost_limit=0.25
  --env.repo.path=/path/to/temp/repo
  --env.deployment.image=python
  --prompt_satement.path=/path/to/translation_task.md
```

Listing 5.2: SWE-agent invocation for translation.

5.3 Large Language Models Evaluated

In this section we briefly describe the large language models tested in this study, including commercial, open-source, reasoning, and non-reasoning LLMs. To keep total inference costs and node hours down, we use smaller and more efficient versions of commercial API LLMs and quantized versions of open-source LLMs.

Gemini 1.5 Flash

Google AI released the multimodal Gemini 1.5 Flash in May 2024. Gemini model parameter counts are not published [88]. We also tested gemini-2.0-flash, but found performance to be severely worse compared to 1.5, and we do not show those results here. We select Gemini 1.5 Flash to represent state-of-the-art free but closed-source LLMs.

GPT-4o mini

OpenAI's GPT-4o mini was released in July 2024. It is a smaller, lower-cost version of GPT-4o, itself a multimodal variant of GPT-4 [69]. OpenAI models are closed-source and model parameter counts are not published. We select GPT-4o mini as an affordable paid LLM, from a popular commercial provider, but without the reasoning capabilities that o4 mini provides.

o4 mini

Another OpenAI offering, o4 mini is a more recent product from April 2025. It is a smaller, more cost-efficient version of the o4 reasoning model [68]. Inference API costs for this model

are higher than 4o mini, and as a reasoning model o4 produces significantly more output tokens per request.

Llama 3.3 70B Instruct

The Llama model family is an open-source series of LLMs developed by Meta AI [30]. Version 3.3 was released in December 2024 [57]. We use the instruction fine-tuned version of the model, a top performer in code tasks among open-source LLMs. To fit the 70 billion parameter model on a single Delta node, we employ a 4-bit GGUF quantized version of the model. Llama 3.3 represents the state-of-the-art in open-source non-reasoning LLMs.

QwQ-32B

QwQ-32B is a medium-sized 32-billion-parameter reasoning model based on Qwen-2.5 released in March 2025 by Alibaba Cloud [76]. It can achieve competitive performance with other reasoning models, including DeepSeek-R1, but at smaller size. We select QwQ to represent state-of-the-art open-source reasoning LLMs. We use an 8-bit GGUF quantized version.

5.4 The ParEval-Repo Suite of Translation Tasks

In this section we describe the translation tasks we select for inclusion in ParEval-Repo.[1] Table 5.1 summarizes all the benchmarking tasks, including for each application the source lines of code (SLoC), cyclomatic complexity (CC), number of files, and the programming models translated between. Cyclomatic complexity serves as a general measure of software complexity,

[1]The full ParEval-Repo is available at `https://github.com/parallelcodefoundry/ParEval-Repo`

specifically measuring the number of linearly independent paths through the program. We collect it using the `pmccabe` tool. For all applications, we leverage the correctness validation test cases provided by the developers to ensure the translation preserves correctness.

	SLoC	CC	# Files	OMP Th.	OMP Of.	CUDA	Kokkos
nanoXOR	109	33	2	✓	?	✓	?
microXORh	127	33	3	✓	?	✓	?
microXOR	133	33	4	✓	?	✓	?
SimpleMOC-kernel	780	59	6		?	✓	?
XSBench*	2307	261	8	✓	✓?	✓	✓?
llm.c	3039	360	6		?	✓	?

Table 5.1: The set of applications used as translation tasks in PAREVAL-REPO, including number of source lines of code (SLoC), cyclomatic complexity (CC), and number of files. On the right side, we indicate the programming models already implemented in the application in green and with a checkmark. Programming models we will attempt to port to are indicated with a question mark and colored yellow. For XSBench, we are porting to an existing model to examine whether the existence of a public port improves translation ability.

5.4.1 Applications selected

The primary constraint on application selection is the existence of public ports to the programming models we are attempting to translate into. If a public port to that model does exist, data contamination, or the presence of the exact code we want the LLM to generate in its training dataset, becomes likely. In order to evaluate the extent to which possible data contamination is relevant to translation accuracy, we do include one case, XSBench, which is known to have publicly-available versions in the programming models we target. While it would certainly be possible to extend our PAREVAL-REPO to more complex applications, as we will demonstrate in Section 5.7, applications at this level of complexity are sufficient to expose significant weaknesses in current state-of-the-art approaches to full application translation.

Below we briefly describe each application in PAREVAL-REPO. Each application, in addition to differening in size and file count, differs in key characteristics that affect the difficulty of its associated translation tasks.

nanoXOR

nanoXOR is a custom micro-application written specifically for PAREVAL-REPO. It consists of a single kernel and driver function in a single source file. The nanoXOR kernel performs a simple four-point stencil with the XOR operator over a 2D grid.

microXORh

microXORh is also a custom micro-application, one step in complexity above nanoXOR. It differs only in that the GPU kernel is written in a separate header file that is included in the main function file, introducing a simple compile-time dependency.

microXOR

microXOR is the last of our custom micro-applications. It is one more step in complexity up from microXORh, with the kernel and main driver function in two separate source files, introducing a simple link-time dependency to the translation problem.

SimpleMOC-kernel

SimpleMOC-kernel is a proxy application for SimpleMOC [92], representing neutron flux attenuation. It exists only as a CUDA app in public repositories, but has an external library

dependency on cuRAND, posing an additional challenge for translation tools.

XSBench

XSBench is a proxy application for OpenMC [93], representing macroscopic cross-section lookup. XSBench is a substantial step up in complexity from SimpleMOC-kernel, but does have publicly-available implementations of the translations we attempt. This case will provide insight into whether possible data contimination can affect translation success rate.

llm.c

llm.c is a CUDA implementation of LLM pretraining. We have slightly reduced the size of the llm.c application to focus on critical application components. llm.c provides a useful case study for making AI applications portable, as well as a more complex case that does not have publicly-available ports to other programming models as XSBench does.

5.4.2 Programming model translation pairs selected

We further subdivide our translation tasks into pairs of programming models, the source programming model and the destination programming model for translation.

5.4.2.1 CUDA to OpenMP Offload

For all applications we examine translation from CUDA to OpenMP GPU Offload. OpenMP has offered GPU programming support (called "offloading") since specification version 4.0 [70]. OpenMP directives are relatively unobtrusive code modifications, but compared to the much more

obtrusive and explicit CUDA model, we should expect translations to require a significant degree of code changes. Converting to OpenMP offload also poses compilation challenges, as we require the LLM to produce a working Makefile that provides correct compile flags to use OpenMP offload for the specified compiler and system.

5.4.2.2 CUDA to Kokkos

For all applications we also test translation from CUDA to Kokkos [94]. Kokkos is a portable programming model that can execute on NVIDIA GPUs by acting as an abstraction layer over CUDA code. As such, Kokkos code is often structurally relatively similar to CUDA code, including kernels and explicit memory management. However, it poses greater challenge with successful compilation, with the addition of a library dependency, advanced C++ features, and the CMake build system generator.

5.4.2.3 OpenMP Threads to OpenMP Offload

For most applications, excluding those that lack a CPU OpenMP (i.e., threads) implementation, we evaluate translation from OpenMP threads to offload. We would expect this task to be easiest, largely requiring modification of existing OpenMP directives. However, this is the only translation task that requires migration from CPU to GPU parallelization.

In total, we evaluate translation of six applications for two translation pairs, and four applications for the third translation pair, for a total of sixteen unique translation tasks.

5.5 Metrics for Repo-level Translation Correctness and Performance

In this section, we define the metrics we employ to assess the quality of translations of full-scale applications between parallel programming models. To clarify the terminology used here onwards, we define the translation task. To complete an individual translation task, the LLM tool being assessed must take as input an application, currently written using some *source* programming model, and return an updated version of the input application that is functionally equivalent but implemented using some *destination* programming model. We consider the quality of the translation tool and its output, the translated application, in the following categories:

1. **Correctness**: the translated application must compile and run correctly, according to the same unit tests used to assess the original application. In addition, the translated application must execute using the desired destination programming model, in the manner specified in the prompt.

2. **Token economy**: We measure the number of input and output tokens used in completing inference for the prompts required by the tool to generate the translated application.

In the remainder of this section we define our metrics for each of these categories of quality.

5.5.1 Metrics for Correctness

For correctness, we adopt the definition of pass@k from [66], as described in 2.2.2 by Equation (2.1), which estimates the likelihood that the LLM will generate a correct solution for a given task over k attempts at that task. We report both the average of pass@k over all tasks in PAREVAL-REPO as well as per-task pass@k.

Each application in our benchmark suite provides test cases with correct expected output, and a correct solution must generate expected outputs for those tests. In addition, to be considered correct we also require that the application be implemented using the desired target programming model and actually execute on the hardware that the prompt requests the translation to be executable on. For example, if a translation to Kokkos compatible with running on an NVIDIA GPU is requested, the translated application must actually execute on the GPU, not fallback to CPU (host) execution.

The typical translation task for a full application can take some time, on the order of ten to thirty minutes. As such, pass@1, or the likelihood of correctness for a single translation attempt, provides the best insight into user experience among possible k values. We will follow this for all @k metrics employed in this chapter.

To separately consider compilation success and failure this in our evaluation, we employ build@k, which measures the probability that the model will generate a compilable solution given k attempts. This metric is similar to pass@k except that it considers all samples that compile, not just those which are correct. The number of correct samples, c_t, is replaced with the number of buildable samples, b_t. The typical translation task for a full application can take some time, on the order of ten to thirty minutes. As such, $pass@1$ and $build@1$, or the likelihood of correctness and build success for a single translation attempt, provide the best insight into user experience among possible k values.

5.5.2 Metric for Token Economy

We also measure the token economy of LLM inference prompts generated by the translation tool, to understand how tool choice affects cost of translation. Most LLMs inference serving APIs charge users by the number of input tokens and output tokens used, and output tokens are typically more expensive than input tokens. Self-hosted LLM inference does not directly incur a financial cost, but does incur a compute time cost in node hours, charged to an allocation if using a cloud or supercomputing center. In such cases, the number of tokens consumed is proportional to the compute time of the inference prompt. We report for each combination of translation tool and translation task the average number of tokens consumed. The tokenization process differs across LLMs, so these values should be compared primarily between different LLM-based tools sending prompts to the same underlying LLM. To enable comparison between LLMs, we provide estimated costs in dollars or node hours as appropriate in Section 5.7.4 for two top-performing LLMs.

To assess the impact of accuracy on token economy, we propose the *expected token cost* (E_κ) metric. This is defined as the expected number of generations required to produce a correct translation (i.e., the multiplicative inverse of the pass@1 score) multiplied by the average number of tokens required to produce a single translation. E_κ is defined explicitly in Equation (5.1).

Average tokens used per generation for the task

$$E_\kappa = \left(\frac{1}{\text{pass@1}} \right) \cdot \kappa \tag{5.1}$$

Single generation correctness chance

5.5.3 Identifying and Classifying Translation Errors

To identify specific opportunities for improvement, we analyze mistakes made by LLMs in their translation attempts. Failures in the translation process can come from many sources and are only recorded in unstructured build and run logs. To evaluate errors from the thousands of builds and runs we perform, we develop a semi-automated method to classify errors in the translation process. We cluster the build and run logs of the translated applications into sets of similar errors, and then name these clusters based on the error class they reflect.

We first convert the build and run logs of each translation and run to vector embeddings using the `word2vec` [63] model. This yields for each translation a single vector that captures the semantics of its output logs. We then cluster these vectors using the `DBSCAN` [34] algorithm, a density-based clustering algorithm that can identify clusters of arbitrary shapes, is robust to noise, and has only two hyperparameters that need to be tuned. We manually inspect the quality of resulting clusters to tune DBSCAN's hyperparameters.

While the clustering is helpful, it produces many clusters and fails to join some similar errors. We make a manual pass over the algorithmically-generated clusters to merge them and reassign samples to different clusters where appropriate. During the manual pass we also assign a short label to each cluster that describes the category of error it contains. This combination of automated clustering, followed by manual correction and labeling, allows us to productively and scalably classify the most common errors in the translation process across the translation tasks in PAREVAL-REPO.

5.6 Experimental Setup

In this section we describe the experimental setup for translation generation (inference) and testing.

5.6.1 Inference setup

For commercial API models, we set a total budget of 200 dollars, with 170 dollars for o4-mini and 30 dollars for GPT-4o-mini. We complete Gemini inference using the Google's API free tier.

For locally-hosted open-source models, we employ the vLLM inference engine version 0.8.2 configured in server mode. We run inference on the Delta system at NCSA[2], specifically the single-socket A100 nodes, which have four 40 GB NVIDIA A100 GPUs, one 64-core AMD Milan CPU, and 256 GB of RAM. We configure vLLM to use all four GPUs with tensor parallelism with prompt prefix caching enabled, and batch requests where possible.

5.6.2 Evaluation setup

All translation tasks in PAREVAL-REPO require translation to a GPU programming model. We test translations on a single NVIDIA 40GB A100 GPU, hosted on the Zaratan system at the University of Maryland[3] on a single node with a 128-core AMD EPYC 7763 CPU and 512 GB of RAM. We use CUDA 12.3.0 universally and compile OpenMP offload codes with LLVM 19.1.7 and Kokkos codes with Kokkos 4.5.01 and GCC 11.3.0.

[2] https://docs.ncsa.illinois.edu/systems/delta/en/latest/index.html
[3] http://hpcc.umd.edu

5.7 Results

In this section we present example translations, results for correctness of generated code translations across LLMs, tasks, and techniques, our analysis of the clustering of errors encountered in attempting to compile LLM translations, and our analysis of the token efficiency of cases where the LLM translation is successful.

5.7.1 Examples of successful and unsuccessful translations

Listings 5.3- 5.5 provide the original nanoXOR CUDA kernel, and examples of correct and incorrect translations to OpenMP Offload generated by QwQ with the non-agentic and agentic translation techniques, respectively. The agentic translation is incorrect, as it omits the `target` and `parallel for` directives, which are required to run the loop on the GPU and parallelize the loop across threads.

```
__global__ void cellsXOR(const int *input, int *output, size_t N) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N && j < N) {
    int count = 0;
    if (i > 0 && input[(i-1)*N + j] == 1) count++;
    if (i < N-1 && input[(i+1)*N + j] == 1) count++;
    if (j > 0 && input[i*N + (j-1)] == 1) count++;
    if (j < N-1 && input[i*N + (j+1)] == 1) count++;
    output[i*N + j] = (count == 1) ? 1 : 0;
  }
```

```
}
```

Listing 5.3: Original nanoXOR CUDA kernel.

```
#pragma omp target data map(to: input[0:N*N]) map(from: output[0:N*N])
{
  #pragma omp target teams distribute parallel for collapse(2)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      // Kernel body identical, omitted for brevity
```

Listing 5.4: Correct nanoXOR OpenMP Offload translation generated by the non-agentic translation method with `qwq-32b-q8_0`.

```
#pragma omp target data map(to: input[0:N*N]) map(from: output[0:N*N])
{
  #pragma omp teams distribute collapse(2) num_threads(blockEdge * blockEdge)
  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < N; j++) {
      // Kernel body identical, omitted for brevity
```

Listing 5.5: Incorrect nanoXOR OpenMP Offload translation generated by the agentic translation method with `qwq-32b-q8_0`.

5.7.2 Translation correctness

In Figure 5.3 we present build@1 and pass@1 results for our three programming model translation pairs. Within each row of subfigures, the left subfigure indicates build@1 scores, the likelihood of generating a compilable translation given one attempt, while the right subfigure

indicates pass@1, the likelihood of generating a translation that passes correctness tests given one attempt.

Within each subfigure, the heatmaps are organized into rows and columns. Each column of heatmaps corresponds to a translation technique, non-agentic, top-down, and SWE-agent (where applicable). The upper row of heatmaps in each subfigure corresponds to "Code-only" score, while the lower row of heatmaps corresponds to "Overall" score. "Code-only" score considers only the correctness of the generated source code, using a pre-written ground truth Makefile or CMakeLists.txt manually translated by the authors to compile the LLM-translated source code. "Overall" score reflects the results of using both the LLM-translated source code and build system.

Note that an experiment configuration with no value in its heatmap cell indicates that we do not run that case, and results of 0 indicate that we run the configuration but it does not generate any correct translations. In several cases, we were not able to generate translations for a combination of LLM, translation technique, application, and programming model pair. The non-agentic approach, due to the requirement that the LLM generate each file in its entirely in one response, cannot scale to some application sizes due to exceeding LLM output context limits. This is the case for Gemini and GPT-4o when translating llm.c, and for Gemini when translating XSBench from CUDA to OpenMP offload. In some cases with the top-down agentic approach using local models, we did not complete translation due to exceeding our per-experiment budget of 8 node hours. This is the case for qwq translating XSBench and llm.c with all programming model pairs as well as Llama-3.3 translating XSBench and llm.c from CUDA to Kokkos. Finally, we present SWE-agent results for a subset of the cases due primarily to the SWE-agent's incompatibility with Makefile mentioned in Sec. 5.2.3, but also omit XSBench and llm.c to remain within our

OpenAI API budget.

Observation 13

Without the file chunking enabled by the agentic approach, non-agentic translation encounters a hard upper limit in application size supported, depending on the output context limit of the LLM used.

Across all cases, we observe that Overall score is consistently significantly lower than Code-only score. This indicates a substantial capability gap between source code translation and build system generation. To further explore the reasons why LLMs fail to generate functional build systems, we examine the types of build errors encountered in this study in Sec. 5.7.3.

Observation 14

Across applications, translation techniques, and LLMs, generating working build systems is more challenging than generating correct source code.

Examining all results along the axis of translation technique, we observe that the non-agentic approach, where it can carry out a translation, tends to achieve the highest scores in both build and pass@1, although this varies across LLMs used. SWE-agent, while only tested with one LLM and programming model pair, achieves moderate success, particularly in overall build@1 score, but does not manage to generate any code that passes correctness tests. The non-agentic likely outperforms the top-down agentic approach due to the greater quantity of repository context provided, highlighting the need for more sophisticated approaches in future work looking to develop full application translation agents. Along the programming model translation pair axis, we observe that CUDA to Kokkos translation is significantly more challenging that the translations involving OpenMP.

Observation 15

The non-agentic approach generally outscores the top-down agentic technique and SWE-agent, particularly in pass@1 scores for the OpenMP Threads to Offload task.

Observation 16

CUDA to Kokkos translation proves signficantly more difficult than CUDA to OpenMP offload and OpenMP threads to offload for all LLMs and techniques tested.

Turning next to the application axis, we find that more complex applications generally pose greater challenge for all LLMs and translation techniques, as expected. One notable exception can be found with non-agentic Llama-3.3 translating CUDA to OpenMP offload, where code-only pass@k is 0.76 for microXORh but only 0.2 for nanoXOR. Additionally, non-agentic Llama-3.3 translating OpenMP threads to OpenMP offload achieves a pass@k of 0.68 for microXOR and 0 for microXORh and nanoXOR. Overall, no combination of translation technique and LLM achieves a pass@k above 0 for any application larger than microXOR. This key finding indicates that LLM-based translation cannot yet produce working code in a fully automated manner. In Sec. 5.7.3, we explore the key obstacles to success in translating larger applications in terms of most frequent compilation errors in translated code.

Observation 17

No translation technique and LLM combination can successfully translate any application larger than microXOR.

5.7.3 Error clustering

We conduct a clustering analysis on the error messages recorded by ParEval-Repo's translation testing code when attempting to build the generated translations, as described in

Sec. 5.5.3. Figure 5.4 displays the results of this analysis, after manually combining highly similar clusters and removing clusters of less interest, including errors related to missing files and build timeouts as well as successful build outputs. Across applications and LLMs, errors relating to CMake configuration, as well as undeclared identifiers and function argument or type mismatches for apps besides nanoXOR, are broadly common. This suggests that coordination of function interfaces and variable names across files and successful configuration of CMake for Kokkos builds are common points of difficulty across LLMs. These issues must be addressed by improved LLM translation techniques and prompting, rather than tuning the choice of LLM.

Observation 18

Difficulty with CMake configuration and managing cross-file dependencies for applications larger than a single file occur consistently across LLMs.

However, we also observe that some categories of errors are only highly prevalent for some LLMs and applications. For example, Gemini appears most likely to struggle with Makefile syntax and compiler flags, particularly for SimpleMOC-kernel, Llama-3.3 is particularly susceptible to source code syntax mistakes, and GPT-4o mini produces translation that fail to link especially often for microXOR. We observe that LLMs can still encounter unique pitfalls that may be avoided by tuning the choice of LLM.

Observation 19

Many other categories of errors, especially those involving Makefile and source code syntax, compiler flags, and linker problems, occur much more frequently for specific LLM and application combinations.

5.7.4 Inference token economy

We also analyze the cost of translation in terms of token economy, or the number of inference tokens required to complete a translation. Fig. 5.5 displays total inference tokens used for translation for each technique, LLM, and application combination, averaged across programming model translation pairs and individual generations. Note that in Figure 5.5 some additional heatmap cells are empty compared to Figure 5.3, because we do not include the total tokens consumed for any cases where zero total correct translations are generated. Among commercial API LLMs, the non-agentic translation technique consumes more inference tokens than top-down, but among open-source locally-hosted LLMs, the top-down agentic approach is more expensive, largely because the commercial API LLMs are more conservative in selecting translation context in the top-down approach. In the non-agentic approach, QwQ in particular consumes a significant quantity of tokens, due to the size of its reasoning output. o4-mini is also a reasoning model but consumes significantly fewer additional tokens.

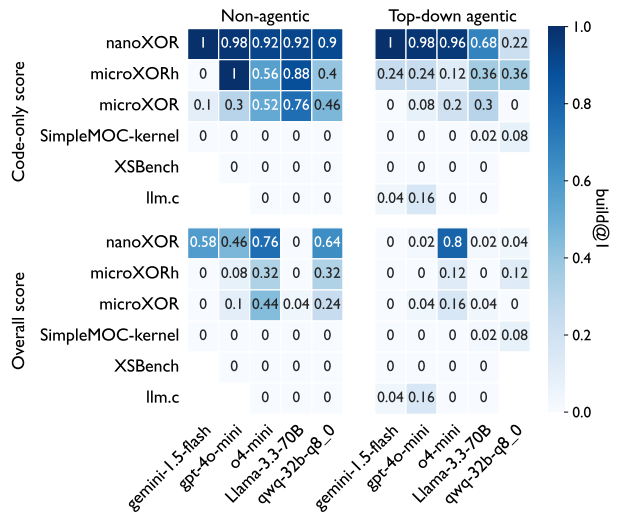
Figure 5.6 lists E_κ, the expected number of tokens needed to produce a successful translation, as defined in Sec. 5.5.2. We aggregate this metric only over cases where the pass@1 is greater than 0. We can conclude from Fig. 5.6 that among commercial API models, non-agentic o4-mini produces correct translations at the lowest token cost, while among open-source locally-hosted models, non-agentic Llama-3.3 is cheapest.

Given the expected token cost provided in Fig. 5.6, we can estimate the total cost in US dollars or in node-hours for the least expensive API and open-source models, respectively. These estimates are listed for the three applications that can be successfully translated in Table 5.2, and are calculated using public OpenAI API costs for o4-mini as well as our observed average

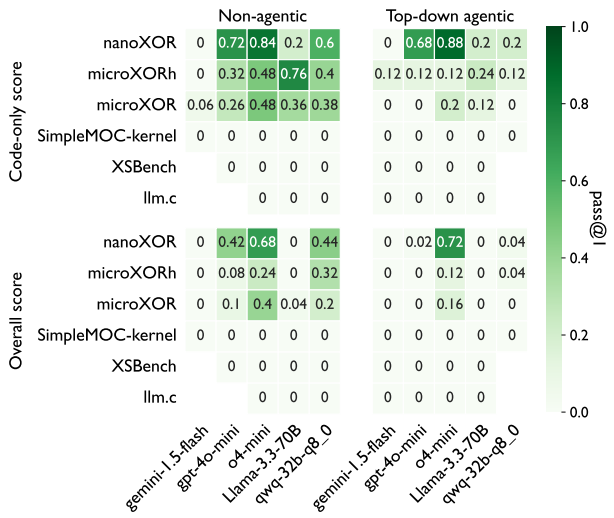
generation throughput of 187 tokens per second on a single node of the Delta system with vLLM, as described in 5.6. Note that Llama-3.3 nanoXOR cost is particularly high due to Llama's unexpected difficulty with that simple application, as described in Sec. 5.7.2.

	nanoXOR	microXORh	microXOR
Non-agentic o4-mini	$0.03	$0.04	$0.05
Non-agentic Llama-3.3	0.6 n.h.	0.06 n.h.	0.08 n.h.

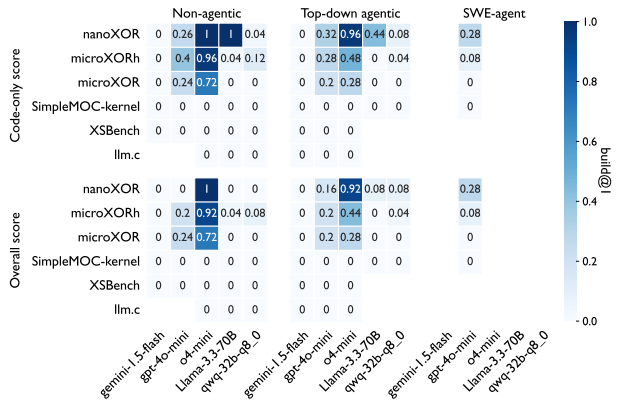
Table 5.2: Estimated cost in dollars or node-hours for successful translation, based on E_κ, published OpenAI API costs, and our observed average generation throughput on a single Delta node with Llama-3.3 in vLLM (187 tokens per second).



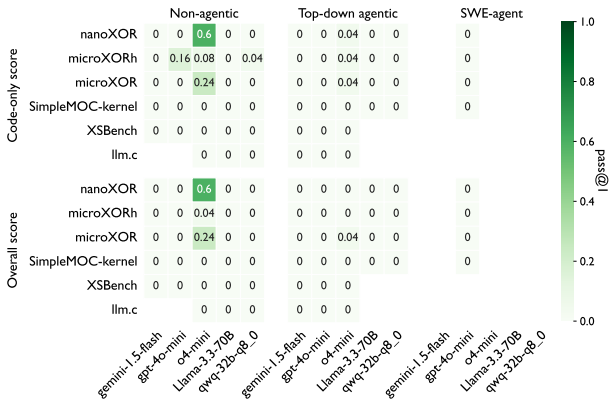
(a) build@1 for CUDA to OpenMP Offload



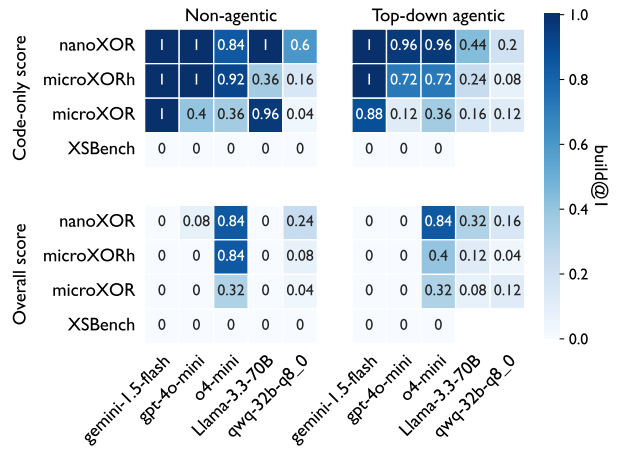
(b) pass@1 for CUDA to OpenMP Offload



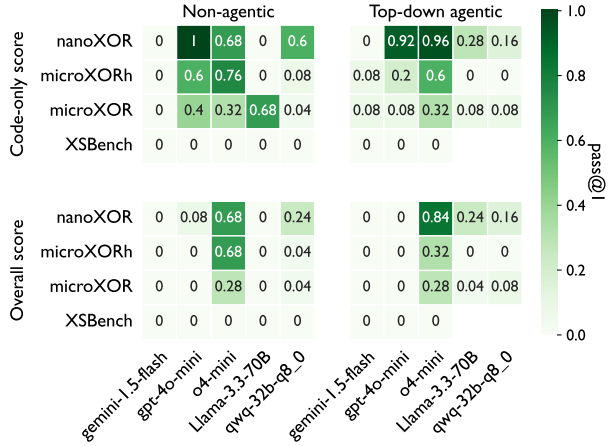
(c) build@1 for CUDA to Kokkos



(d) pass@1 for CUDA to Kokkos



(e) build@1 for OpenMP Threads to OpenMP Offload



(f) pass@1 for OpenMP Threads to OpenMP Offload

Figure 5.3: Correctness metrics for all translation tasks.

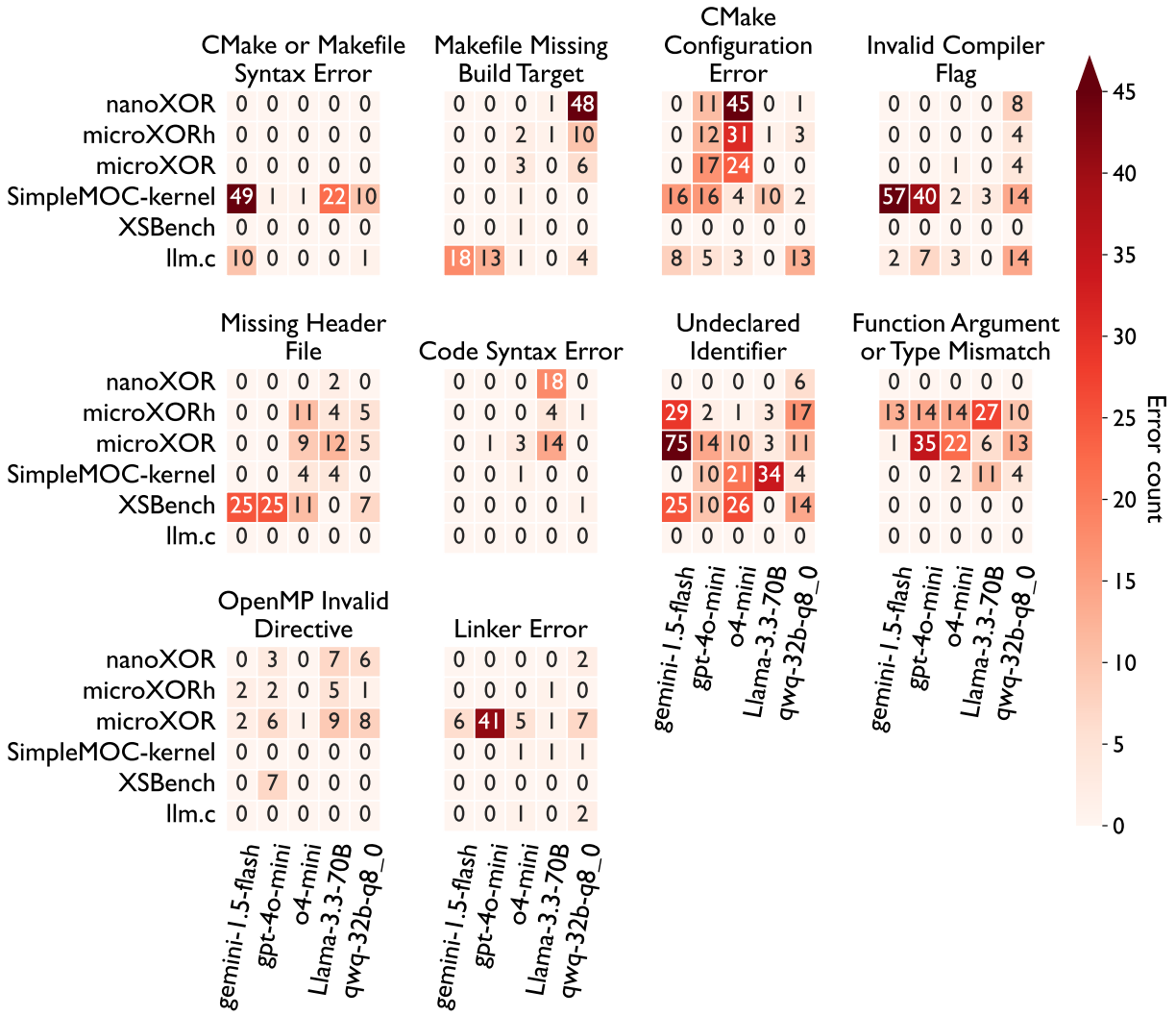


Figure 5.4: Count of categories of errors encountered when across combinations of large language models and application.

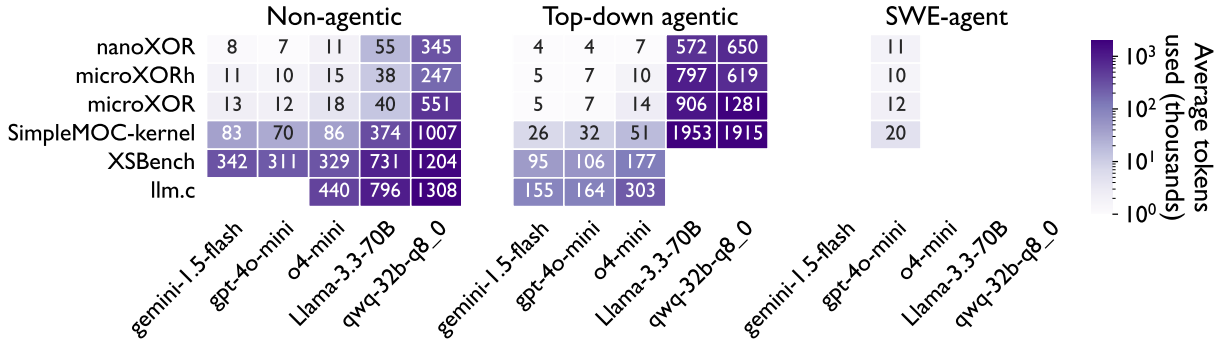


Figure 5.5: Total inference tokens used in translation, averaged across individual generations and programming model translation pairs.

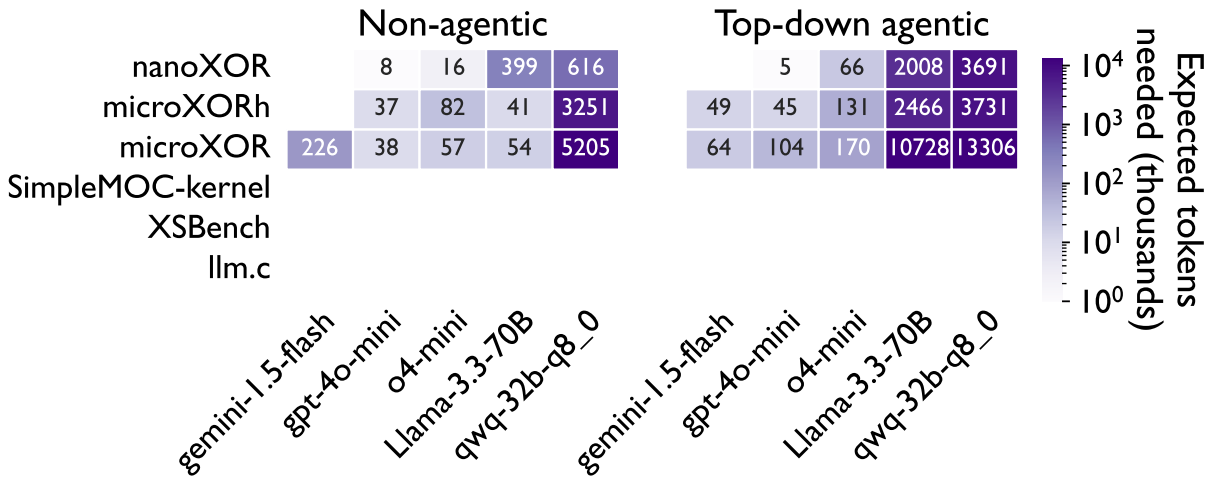


Figure 5.6: Expected tokens needed for successful translation (E_κ), averaged across individual generations and programming model translation pairs where at least one generation passed correctness testis (i.e., pass@1 was greater than 0).

Chapter 6: KEET: Explaining GPU Kernel Performance to Enable Performance Optimization

6.1 Introduction

Graphics Processing Units (GPUs) have become ubiquitous in modern supercomputers and clusters. The Top500 list [91], as of November 2025, includes 255 systems (51% of the full list) that use GPUs or other accelerators. Further, nine out of the top ten systems use GPUs. Scientific applications must utilize GPUs as efficiently as possible to achieve maximum performance on these systems. Writing and optimizing code to achieve this goal on a given GPU architecture requires careful consideration of the hardware architecture and the requirements of the particular application. Meanwhile, modern GPU architectures are becoming increasingly sophisticated. Each new generation adds not just additional streaming multiprocessors and memory bandwidth, but also new features, such as specialized cores and more complex memory hierarchies. In the worst-case scenario, every GPU architecture release requires a new profiling and optimization effort to achieve peak performance, which is an unacceptable drain on developer time and resources.

GPU software developers and performance engineers employ profiling tools such as NVIDIA's Nsight Compute (NCU) [67] to understand and optimize the performance of their code on a given

GPU architecture. In particular, developers must identify which hardware components and code regions are causing performance bottlenecks. However, despite the wealth of information these tools provide, extracting insight from GPU profiling tools remains a complex and largely manual task. While NCU provides a graphical interface to visualize metrics and a rule engine to automatically highlight potential problems, it remains challenging to easily and quickly identify key insights, especially from large sets of profiles. In this chapter, we address the challenge of extracting actionable and informative optimization insights from individual and multiple GPU kernel profiles.

Tools such as GPA [105] and DrGPU [41] have proposed automated approaches to assist with analyzing GPU kernel performance on NVIDIA GPUs, but both these tools take a rule-based approach that can be somewhat rigid. DrGPU, for example, decomposes stall reasons into categories in a tree structure and offers suggestions to address the most frequent stall reasons. However, these suggestions are statically defined and do not consider how specific characteristics of the kernel algorithm might interact with performance bottlenecks. Further, these existing tools require manual intervention to incorporate new GPU hardware features or performance metrics when new GPUs are released. Most importantly, they do not suggest code changes tailored to the existing kernel code.

We propose a new approach to address these limitations by interpreting GPU kernel performance profiles using Large Language Models (LLMs). Our Kernel Execution Explanation Tool (KEET) is an LLM-based agentic framework that automatically generates a natural language report explaining the performance of a GPU kernel using NCU profiles. It identifies key performance bottlenecks and suggests specific code changes to address them. Further, it scales easily to larger sets of profiles and can be used to understand the relationship of kernel tuning

knobs such as block size or algorithm parameters to performance and hardware behavior.

By using LLMs to generate these reports, KEET provides natural language explanations grounded in the measured profiling data, rather than reciting data or results of static rules. We leverage public knowledge of GPU architectures, NCU performance metrics, and optimization strategies embedded in pre-trained LLMs to generate useful insights from profiling metrics and kernel code. Because the approach does not rely on statically defined rules, and instead uses pre-trained LLMs as the core reasoning system, it has the potential to be easily updated to understand newer GPU architectures by updating to a model with a more recent knowledge frontier. In this work, we present and describe the design of KEET, evaluate its effectiveness against prior work, and study its sensitivity to input profile dataset size as well as the contribution of individual LLM agent roles to overall output quality.

6.2 Design of KEET

KEET is an LLM-based agentic framework for explaining the performance of GPU kernels by analyzing their Nsight Compute (NCU) performance profiles. Its design emphasizes scalability to large numbers of profiles, interpretability of intermediate analysis steps, and integration of algorithmic understanding. KEET takes as input a single NCU profile, or a set of multiple profiles, along with the kernel source code and descriptions of the run configuration(s) used to generate the profile(s). It produces a natural language report that explains kernel performance, identifies key code locations and hardware bottlenecks, analyzes the relationship between tuning knobs and performance, and suggests specific optimizations.

6.2.1 Overview of KEET

We present the overall structure of KEET in Figure 6.1. Each colored box represents an LLM agent role – a prompt template populated with appropriate context from prior agents as indicated by arrows. As indicated in the legend, some agent roles are optional, and others are bypassed depending on whether handling for multiple source files or multiple profiles is required. The tool takes as input the kernel source code files, NCU profile(s), and descriptive metadata. It outputs the final performance explanation report and an accompanying review.

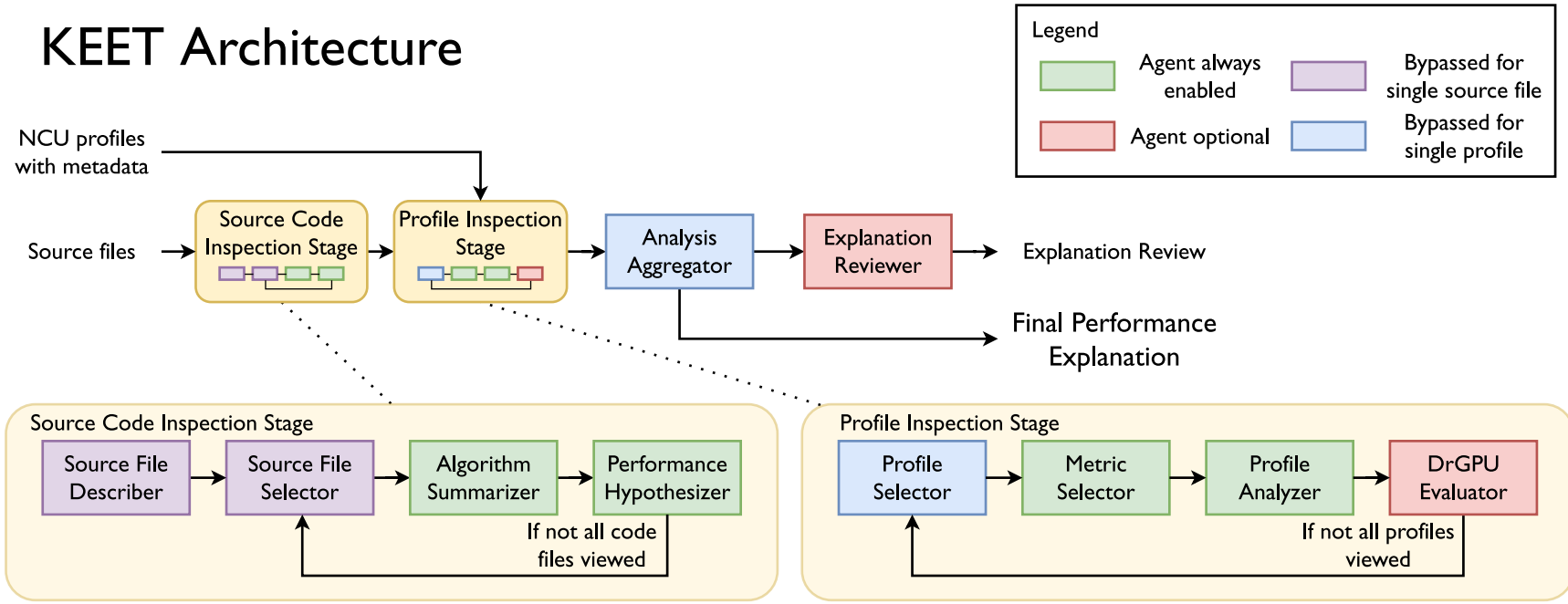


Figure 6.1: The architecture of KEET, where each green box represents an agent role and the arrows represent the data flow between them. The Source Code Inspection and Profile Inspection stages are shown in detail below the overall architecture diagram.

Rather than relying on a single monolithic LLM prompt, KEET decomposes analysis into multiple specialized agent roles. Together, these agents implement an iterative analysis workflow that begins with code understanding and transitions to empirical performance analysis. This architecture offers several advantages: it enables the use of a wider range of data sources than a single prompt would allow, supports iterative refinement using outputs from prior roles, and produces interpretable intermediate results. These intermediate outputs can be reviewed to trace the origins of analysis claims. In Section 6.5, we present the performance of KEET against baseline approaches as described in Sec. 6.3.1.

Internally, the tool is organized into two iterative stages: the Source Code Inspection Stage and the Profile Inspection Stage, each indicated by a yellow box in the center of the diagram and expanded in detail above and below the central diagram. These two stages are followed by an Analysis Aggregator step and finally an Explanation Reviewer step. Below we explain and motivate each stage, starting with the Source Code Inspection Stage and highlighting key agent roles.

6.2.2 Source Code Inspection Stage

The Source Code Inspection Stage is responsible for extracting insight from the kernel source code. It first prepares descriptions of the source files to support selection, and then iteratively selects from the available code files and refines a summary of the algorithm and a set of performance hypotheses based on each code file. This stage repeats until all files have been reviewed at least once, after which KEET execution proceeds to the Profile Inspection Stage.

A key component of this stage is the Performance Hypothesizer role, which reviews the

current selected code file and the current algorithm summary to update a running set of performance hypotheses. These performance hypotheses are the tool's heuristic predictions for how the kernel should perform based on the code alone. They serve as a plan for later exploration of empirical performance data.

6.2.3 Profile Inspection Stage

The Profile Inspection Stage is responsible for extracting insight from the NCU profiles. It iteratively selects one or more profiles and a subset of metrics, and produces a performance analysis for each selected group. It also optionally leverages the DrGPU tool to update the analysis based on its suggestions. This stage repeats until all profiles have been analyzed at least once, after which KEET execution continues to the Analysis Aggregator and Explanation Reviewer steps. Several roles are notable in this stage, including:

- **Profile Selector:** selects the next NCU profile(s) to analyze based on the algorithm summary, performance hypotheses, and any previous performance analyses. It attempts to choose profiles that will maximize the information gained from the next analysis pass, particularly as related to the performance hypotheses.

- **Metric Selector:** selects a subset of NCU metrics to analyze next from those available across all chosen profiles. This role reduces the noise in the analysis by focusing on the most relevant metrics for the current pass.

- **Profile Analyzer:** examines the profile metrics, algorithm summary, source code, and analysis guidelines (Section 6.2.5) to produce a detailed performance analysis. This role's

prompt requests analysis of performance bottlenecks, how tuning knobs impact performance, and suggestions to improve performance. It also requests inline citations to the Nsight Compute profiles referenced in the analysis to mitigate hallucination risk and support human review. Listing 6.1 shows the prompt template for the Profile Analyzer role.

- **DrGPU Evaluator:** invokes DrGPU on the profile data and incorporates any suggestions from DrGPU it determines are useful. This role is optional, and we evaluate KEET with and without it in Section 6.5.

```
You are an expert CUDA programmer, assisting with
performance analysis of a CUDA kernel,
{kernel_name}.

Here are the descriptions of each Nsight Compute profile
collected for the kernel:
...

Here is a table of the Nsight Compute data for each
profile mentioned above. Each row corresponds to a
particular metric as indicated in the first column, and
each column corresponds to a particular profile as
indicated in the first row:
...

Here is a table of descriptions of the highest utilized
memory and compute resources for each profile based on
NCU's speed of light throughput breakdown section:
```

```
...

Please construct your analysis using the kernel code as
context. Make reference to the kernel code where relevant
to explain performance behaviors. Here is the kernel CUDA
code:
...

Here is an explanation of the algorithm of the kernel for
reference:
...

Consider the following performance analysis guidelines
when carrying out your analysis:
...

Provide a comprehensive analysis of the performance of
{kernel_name} using the Nsight Compute profiles across
all configurations. Your analysis should include sections
discussing the kernel algorithm; the main performance
bottleneck(s) of the kernel in each configuration, if
any; which GPU hardware components contribute to
performance bottlenecks, along with associated
performance metric values that indicate the bottleneck;
specific code lines related to performance bottlenecks;
discussion of the relationship of configuration knobs and
settings to performance bottlenecks; and suggestions to
```

```
address the identified performance bottlenecks and
optimize the kernel's performance.

Insert inline citations to the Nsight Compute profiles
referenced in the performance analysis, to make the
performance analysis easier to review...

Output the complete, comprehensive, and detailed
performance analysis including inline citations where
needed, in markdown format to ensure readability.
```

Listing 6.1: Abridged version of the NCU Profile Analyzer prompt.

6.2.4 Aggregation and Review

The Analysis Aggregator and Explanation Reviewer roles are responsible for preparing the two outputs of KEET. The Aggregator role combines all the performance analyses generated for each Profile Inspection Stage pass into a final performance explanation report. The Reviewer role compares the final performance explanation against the performance hypotheses generated before viewing any profile data to provide a final assessment of each hypothesis – whether it is confirmed, refuted, or inconclusive given the generated performance analysis. The reviewer role is optional and provides additional interpretability for the explanation generation process.

6.2.5 Performance Analysis Guidelines

The Performance Analysis Guidelines are a set of general human-written GPU performance analysis guidelines included as context for the Profile Analyzer and Aggregator agents. We con-

structed these guidelines as a concise summary of expert wisdom about GPU performance analysis. They are distilled from NVIDIA documentation, existing literature on best practices, and consultation with expert practitioners. Table 6.1 provides a summary of all the guidelines, including the guideline name, which metrics to examine, and key optimization cues.

In contrast to prior tools [41, 105], KEET leverages LLMs to combine hypothesis-driven analysis, adaptive metric and profile selection, and multi-pass analysis aggregation to support scalable, interpretable, and algorithm- and architecture-aware interpretation of GPU performance data. As with all LLM-based tools, KEET may produce lower-quality output depending on the quality of the LLM used and the quantity of profiling data provided. We evaluate these trade-offs empirically in our ablation study described in Section 6.4.4.

Guideline	Metrics to Examine	Optimization Cues
Grid Size	CTAs vs. SM count	Grid $\geq$ #SMs; integer multiple for regular kernels; minimize tail effect; use concurrent kernels if grid is small
Overall Throughput	Achieved FLOPS or BW	$>$70% of peak; lower execution time and elapsed cycles indicate better kernel variant
Performance Limiter	Unit utilization	Single unit $>$70% implies bottleneck; low utilization everywhere implies latency-bound kernel
Occupancy Analysis	SM resource limits	Balance latency hiding and resource use; higher occupancy for memory-bound kernels; identify limiting resource via NCU
Global Memory Coalescing	Memory transaction efficiency	Low sectors/request; contiguous warp accesses; LDG/STG size effects
Cache Performance	L1/L2 cache hit rates	Low hit rate implies poor locality or large working set; long scoreboard stalls; excessive global writes/atomics
Shared Memory Performance	Bank conflicts	Few conflicts; wavefronts per access $\approx$ 1; short scoreboard stalls indicate serialization
Register Spilling	Local memory usage	High local memory implies spilling; long scoreboard stalls; tune registers and unrolling
Instruction Cache	Instruction fetch stalls	High no-instruction stalls; excessive unrolling or large basic blocks
Synchronization	Barrier stalls	High barrier stalls indicate load imbalance or excessive `__syncthreads()`
Thread Divergence	Warp execution efficiency	Active threads/warp near 32; low divergent branches; algorithm-dependent
Instruction Mix	Pipeline utilization	Pipeline utilization $>$70% is bottleneck; rebalance pipelines; consider fast math
Clock Rate	GPU frequency settings	Lock core and memory clocks unless studying frequency as a tuning parameter

Table 6.1: Summary of GPU performance analysis guidelines, metrics, and optimization cues.

In contrast to prior tools [41, 105], KEET leverages LLMs to combine hypothesis-driven analysis, adaptive metric and profile selection, and multi-pass analysis aggregation to support scalable, interpretable, and algorithm- and architecture-aware interpretation of GPU performance data. As with all LLM-based tools, KEET may produce lower-quality output depending on the quality of the LLM used and the quantity of profiling data provided. Furthermore, inference time scales with the number of profiles and analysis passes utilized. We evaluate these trade-offs empirically in our ablation study in Section 6.4.4.

6.3 Evaluation Methodology

In this section, we describe our methodology for evaluating KEET, including other methods we compare with, chosen test cases, downstream tasks and metrics, and the ablation study design. We also describe the LLMs and hardware used.

6.3.1 KEET and Other Methods used for Comparison

We compare two variations of KEET against four other methods for understanding GPU kernel performance data:

- **Code Only**: The method does not generate a performance explanation using profile data. We simply provide the kernel source code to the LLM for the downstream task.
- **Code+Data**: A simple report containing only the raw metric data table extracted from NCU metric data and the kernel source code is provided to the LLM for the downstream task.

- **DrGPU Only**: We use DrGPU to generate a report of the kernel performance, statically converting the suggestions and tree structure into a basic natural language report without using LLMs.

- **LLM+DrGPU**: We pass the textified DrGPU report to a basic LLM prompt including the raw NCU data and kernel source code to generate a performance report.

- **KEET Only**: We use KEET to generate a performance explanation report of the kernel, with the DrGPU Suggestion Reviewer agent disabled.

- **KEET+DrGPU**: We use KEET with the DrGPU Suggestion Reviewer agent enabled, meaning that KEET evaluates the suggestions generated by DrGPU and incorporates them into the report if deemed helpful.

Each of these methods generates output, whether basic code, or data tables, or a natural language report, which we then provide as context to an LLM to complete one of two downstream tasks described below in Section 6.3.3.

Table 6.2: Evaluation cases used to evaluate KEET and baseline approaches.

Application	**Kernel**	**Algorithmic Motif**	**Scientific Domain**	**SLoC**	**DrGPU?**	**GPA?**
b+tree (r)	findRangeK	Graph Traversal	Search	62	•	•
backprop (r)	bpnn_layerforward_CUDA	Unstructured Grid	Machine Learning	67	•	•
pathfinder (r)	dynproc_kernel	Dynamic Programming	Grid Traversal	81	•	•
nw (r)	needle_cuda_shared_1	Dynamic Programming	Bioinformatics	99	•	•
hotspot (r)	calculate_temp	Structured Grid	Material Science	113	•	•
huffman (r)	vlc_encode_kernel_sm64huff	Finite State Machine	Lossless Compression	124	•	•
lavaMD (r)	kernel_gpu_cuda	N-body Simulation	Molecular Dynamics	210	•	•
heartwall (r)	kernel	Structured Grid	Medical Imaging	1327	•	•
LULESH	ApplyMaterialPropertiesAnd-UpdateVolume_kernel	Unstructured Grid	Hydrodynamics	392	•	
XSBench	xs_lookup_kernel_baseline	Monte Carlo	Nuclear Physics	380		
gaussian (r)	Fan2	Dense Linear Algebra	Linear Algebra	17		•

6.3.2 Benchmarks and Applications used for Evaluation

We select a suite of benchmarks and proxy applications to evaluate the quality of KEET's analysis of GPU kernels, including some reused from prior work [105, 41], as well as others of our own selection. Below, we briefly describe the benchmark suite and proxy applications used in this study:

- **Rodinia** is a benchmark suite of GPU kernels covering a wide range of algorithms and applications. We select nine kernels from Rodinia.
- **LULESH** is a hydrodynamics proxy application, which solves a Sedov blast wave problem using unstructured mesh data structures. We use version 2.0 of LULESH.
- **XSBench** is a proxy application for OpenMC, a Monte Carlo neutron transport code. Its single kernel represents the macroscopic cross-section lookup calculation, which we test with a variety of kernel launch parameter settings.

Table 6.2 summarizes the applications and their respective kernels selected for this study. For each application, we list the name of the kernel profiled, whether the kernel was studied in the DrGPU or GPA efforts [105, 41], as well as the algorithmic motif and scientific domain represented by the application and kernel. As listed in Table 6.2, these kernels cover a range of algorithmic motifs and scientific domains while maintaining overlap with prior work for ease of comparison. For each application, we select the kernel that the application spends the most time in. When a kernel is invoked multiple times by the application, we select the longest-running instance to profile. When multiple instances take up the most time, we select the instance with the median invocation time.

6.3.3 Downstream Tasks and Metrics

We quantitatively evaluate the quality of KEET outputs against the other methods described above using two downstream tasks. For each task, we provide the tool or comparison method's report as context to an LLM and ask it to solve a related problem using the provided report as context. If an LLM achieves higher downstream task performance when using one method's report as context compared to another, we conclude that the former method's report is more helpful.

Task 1: Multiple-Choice Question Answering (MCQ): The multiple-choice question answering (MCQ) task asks the downstream LLM to answer a set of multiple-choice questions about the performance of the kernel, using the generated report (if provided) as context. These questions, twenty per kernel, are manually written to assess understanding of the kernel code and NCU profile data. For MCQ we estimate score@1, the expected percentage score on the MCQ test given one attempt, based on scores achieved with twenty attempts for each report generated. The score@1 is equivalent to the average percentage score across all twenty attempts. We provide an example of an MCQ question in Listing 6.2. We evaluate MCQ on a subset of applications due to the manual effort required to write the questions.

```
"question": "Which of these hardware limits is most heavily saturated by this
    kernel?",
"correct_choices": ["SM issue rate"],
"incorrect_choices": ["ADU pipeline throughput",
                      "ALU pipeline throughput",
```

```
5                       "CBU pipeline throughput"]
```

Listing 6.2: Example MCQ question

Task 2: Code Optimization (OPT): The code optimization (OPT) task asks the downstream LLM to implement the suggestions for code optimization provided in the report to maximize the performance of the kernel. If the report does not provide suggestions for code optimization, we prompt the LLM to identify optimizations itself as part of the OPT task. When updated kernel code does not compile or pass test cases, we provide this feedback to the LLM and ask it to try again, up to a fixed limit on retries. More details are provided in Section 6.4.3. To consider the rate of valid and invalid code generation, we also report pass@1, the expected percentage of attempts that generate correct code.

6.3.4 Ablation Studies

To motivate the design of KEET and choices made in the final downstream task evaluation, we perform several ablation studies. Specifically, we study the impact of the two key agent roles (metric selector and profile selector), as well as the impact of number of profiles provided to the tool and the choice of LLM used in KEET. These studies are described in detail in Section 6.4.4.

6.4 Evaluation Setup

In this section, we describe the setup for our experiments evaluating KEET outputs.

6.4.1 Hardware Used

We use the NVIDIA H100 SXM5 as the primary hardware for this study. For the LULESH multi-profile ablation study (Sec. 6.4.4), we also include NVIDIA V100 and A100 GPUs, to represent a range of GPU architecture generations.

6.4.2 Application Configuration and Profile Collection

For all applications, we use default input parameters to check correctness, generate profiles to analyze, and profile after optimizations. All profiles are collected with all NCU metric sections enabled along with metrics needed for DrGPU analysis. We also manually export the line-level data to support DrGPU analysis in both baseline and KEET evaluations.

For the main evaluation, we collect and provide to the tool only one representative profile for each kernel, collected using the default settings and input. For ablation studies (Section 6.4.4), we collect and provide to the tool multiple profiles for each kernel, collected using a range of performance knob settings and in some cases multiple GPU architectures.

6.4.3 Report Generation and OPT Task Setup

For LLM-based report generation methods, we generate three independent reports per experimental setting to account for stochasticity in LLM output. For deterministic methods (Code Only, Code+Data, DrGPU Only), we generate a single report. Each generated report is evaluated using the downstream task LLM, gpt-oss-120b, over twenty independent attempts to estimate expected performance with a single attempt using speedup@1, pass@1, and score@1 metrics. We use the same LLM (gpt-oss-120b) to complete all downstream tasks to ensure fair comparisons.

For the OPT task, we allow up to three retries maximum per OPT solving attempt to balance solution recovery and inference costs. We measure kernel performance using Nsight Systems (NSYS) to extract exact kernel execution time, recording the average execution time over three runs per OPT solving attempt to minimize the impact of any performance variability. We use NSYS rather than Nsight Compute for this measurement to minimize profiler overhead in speedup measurement, and we configure it to collect only minimal kernel timing data. For OPT we estimate speedup@1, the expected speedup in performance over the unoptimized kernel code achieved with one attempt, based on speedups achieved with twenty attempts for each report generated. Attempts that fail to generate valid code after three retries are excluded from the speedup@1 estimate.

6.4.4 Setup for Ablation Studies

We perform four ablation studies to understand the impact of configuration choices and number of profiles provided on KEET's OPT performance. These are described in detail below.

In multi-profile configurations, we study ablation settings using only the XSBench and LULESH applications, both to constrain inference costs and because these applications have readily available tuning knobs. For LULESH we use up to 48 profiles, varying GPU architecture, block size, and maximum number of registers per thread; for XSBench we use up to 75 profiles, varying grid type (unionized, hash, or nuclide), block size, and maximum number of registers per thread. Single-profile settings use the same profiles and applications as the main downstream task evaluations.

6.4.4.1 Ablation Study 1: Metric Selector Agent

We first study the impact of the use of the Metric Selector agent on KEET output quality, comparing OPT performance in two settings: (1) Metric Selector agent enabled, (2) Metric Selector agent disabled. All other agent roles are enabled for this study. We present results for both single-profile KEET runs and additionally discuss the results for multi-profile KEET runs.

6.4.4.2 Ablation Study 2: Profile Selector Agent

We next study the impact of the use of the Profile Selector agent on KEET output quality, comparing OPT performance in two settings: (1) Profile Selector agent enabled, (2) Profile Selector agent disabled. All other agent roles are enabled for this study, and we present results only for multi-profile KEET runs, as the profile selector is not used in single-profile runs.

6.4.4.3 Ablation Study 3: Profile Count

We further study the impact of the number of profiles provided to the tool on OPT performance. We generate KEET outputs with a range of profile counts from one profile to all profiles available. All other agent roles are enabled for this study. We select the profiles to use in each setting by prioritizing profiles with knob settings closer to the defaults, and for LULESH profiles from older GPUs. We rank profiles by a distance score from default configurations. For each profile with a different configuration setting, we compute a normalized distance between its setting and the default setting, breaking ties lexicographically by filename. For XSBench, the default configuration is the unionized grid type, 128 threads per block, and 64 registers per thread. For LULESH, the default configuration is the V100 GPU architecture, 128 threads per block, and 64

registers per thread.

6.4.4.4 Ablation Study 4: LLM Choice

Finally, we study the impact of the choice of LLM used in KEET on OPT performance in the single-profile setting. Table 6.3 summarizes the LLMs used. We test our tool with two open source LLMs as well as one paid API LLM. These LLMs represent the state of the art in paid API models and open source models while having reasonable inference costs. We include the larger 120B parameter gpt-oss and the smaller 30B parameter Nemotron-3 to evaluate the impact of model size. We set reasoning effort to the highest setting for all experiments.

Table 6.3: LLMs used to evaluate KEET and baselines. Note that parameter counts for the closed-source GPT 5.1 are not published by OpenAI.

LLM	**Open Source?**	**Model Size**
GPT-5.1	No	Unpublished
gpt-oss-120b	Yes	120B (5.1B active)
Nemotron-3-Nano	Yes	30B (3B active)

6.5 Results

In this section, we present the results of the evaluation of KEET. We begin by presenting a brief snippet of KEET's output for a specific kernel. After describing the layout of results figures, we present results from the four ablation studies, followed by evaluation of the tool's performance on the downstream tasks of LLM multiple-choice question answering (MCQ) and code optimization (OPT) for single-profile cases. Finally, we present the best baseline and KEET optimization techniques applied for each application.

6.5.1 Sample KEET Output

We present a brief snippet of KEET's output for the gaussian Fan2 kernel in Listing 6.3. Some metric names are abridged for brevity. After presenting this summary section, this particular report continues with suggestions to address the latency and coalescing bottlenecks by increasing the block size, refactoring the memory layout, and reducing grid size where appropriate, enabling the downstream LLM to generate a kernel with a 14.0x speedup over the original code.

```
## 7. Summary of main bottlenecks

1. **Memory-latency bound, not bandwidth bound**
    - Long Scoreboard stalls dominate (`21.39` per issue).
    - L2 hit rate is high; `dram__throughput` is only 1.77% of peak.

2. **Very poor global memory coalescing**
    - `smsp__..._bytes_per_sector_mem_global_op_ld = 8.34 B/sector` (ideal is
     32 B/sector).
    - `derived__..._sectors_global_excessive = 374,514` sectors.

3. **Low warp and thread-level utilization**
    - Blocks have 16 threads (half-warp).
    - Average active threads per issued inst ~= 15.15 (~50% of a full warp).
    - `sm__warps_active` only 10.4% of peak.

4. **Algorithmic tail and wasted threads**
```

```
17    - Fixed grid size independent of pivot index `t` leads to many threads/
    blocks doing no useful work for larger `t`.
```

Listing 6.3: Example KEET output for the gaussian Fan2 kernel

6.5.2 Layout of Figures

In all figures we refer to approaches by shortened names listed in Section 6.3.1. For speedup figures, we plot both the mean speedup@1 score and the maximum speedup observed across all twenty attempts. These are indicated by an X and a dash, respectively. We focus primarily on the maximum speedup in our discussion, as it more closely reflects the speedup a user would use in practice after generating twenty attempts. We include the average speedup@1 to represent the overall performance tendency across all attempts. All figures presenting per-application quantities are sorted by the number of source lines of code in the kernel (with gaussian placed last to allow for a separate y-axis). For brevity, we only present the best of the Code Only and Code+Data methods as "Code-based" in all figures. For some ablation study figures, we also present the harmonic mean [33] of the speedup@1 scores across applications to understand the overall impact of ablated roles.

6.5.3 Ablation Study 1: Metric Selector Agent

We present mean speedup@1 and maximum speedup observed in single-profile cases with the Metric Selector agent enabled and disabled in Figure 6.2. We observe that the Metric Selector agent increases the maximum speedup observed for all applications except b+tree, pathfinder, and LULESH, and similarly affects the mean speedup@1. Overall, as reflected in the harmonic mean

across applications, the Metric Selector agent improves both mean and maximum speedups.

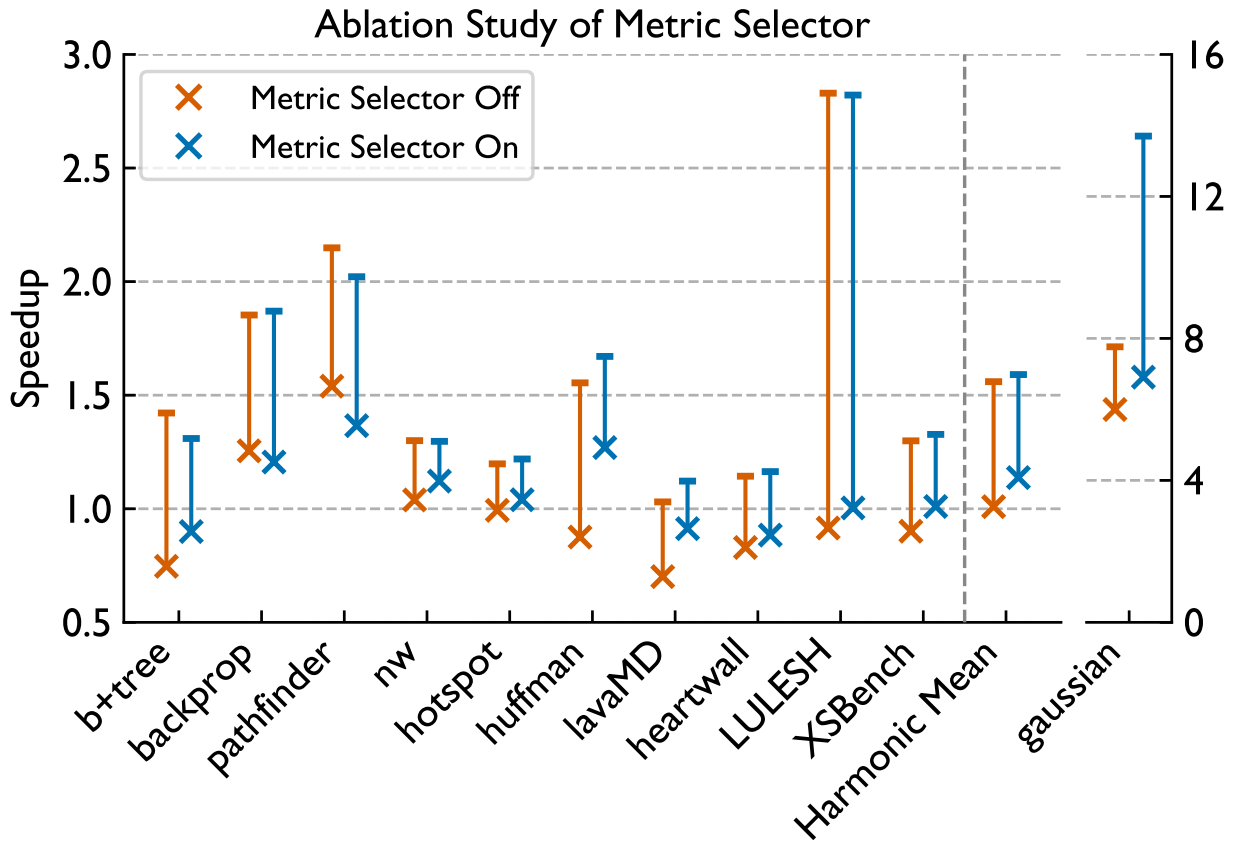


Figure 6.2: Average speedup@1 score and maximum speedup observed in single-profile cases with the Metric Selector agent enabled and disabled.

We also separately evaluate the impact of the Metric Selector agent in multi-profile cases, finding that the Metric Selector agent has minimal impact in those cases. Overall, given these results, we conclude that the Metric Selector agent is more often than not beneficial, and we default to enabling it.

6.5.4 Ablation Study 2: Profile Selector Agent

We present mean speedup@1 and maximum speedup in multi-profile cases with the Profile Selector agent enabled and disabled in Figure 6.3 (left). We observe that the Profile Selector

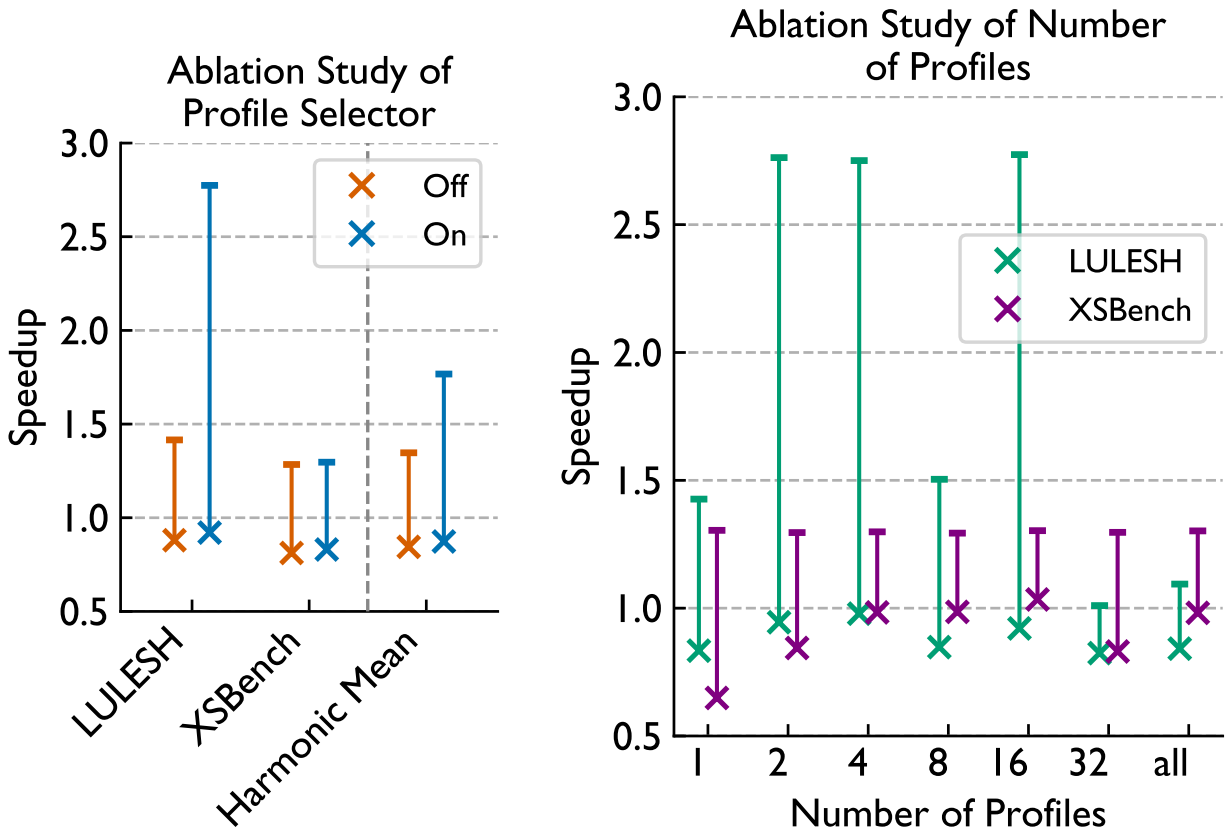


Figure 6.3: Left: Average speedup@1 score and maximum speedup observed in multi-profile cases with the Profile Selector agent enabled and disabled. Right: Average speedup@1 score and maximum speedup observed in multi-profile cases with varying profile counts from one profile to all profiles.

agent increases both mean and maximum speedup, particularly maximum speedup observed for LULESH. As such, we default to enabling the Profile Selector agent in KEET.

6.5.5 Ablation Study 3: Profile Count

We present average speedup@1 score and maximum speedup observed in multi-profile cases with varying profile counts from one profile to all profiles in Figure 6.3 (right). We observe a sweet spot effect in the number of profiles, with the optimal number around four profiles for LULESH and sixteen profiles for XSBench. KEET benefits from additional profiles, but at significantly higher profile counts, we infer that the additional profiles do not provide useful insight

and degrade performance by adding noise to the analysis context.

6.5.6 Ablation Study 4: LLM Choice

We break down speedup@1 averages and maximum speedups by LLM in Figure 6.4, focusing on KEET (without DrGPU) experiments. Overall, GPT 5.1, the largest model, most often achieves the highest average speedup@1 scores, followed by gpt-oss-120b. We note that KEET is able to generate useful reports across LLMs tested, using large and small open-source LLMs as well as closed-source LLMs. However, the most effective LLM is hard to predict for a particular application. For KEET+DrGPU experiments (plots not shown), we find that nemotron-3-nano, the smallest model, most often achieves the highest mean speedup@1 scores, suggesting that DrGPU suggestions are particularly effective in increasing the performance of smaller LLMs. Given these results, we present GPT 5.1 results for KEET experiments and nemotron-3-nano results for KEET+DrGPU experiments in our final downstream task evaluation.

6.5.7 Downstream Task 1: Multiple-Choice Questions (MCQ)

The first downstream task we evaluate is multiple-choice question answering. We present average score@1 values for the MCQ task grouped by application and context types in Figure 6.5. Score@1 indicates the estimated score out of 100 that the LLM should achieve on the MCQ task with one attempt.

We observe that both KEET and KEET+DrGPU consistently achieve higher score@1 scores than baseline methods across applications tested. The exceptions are nw, where the Code-based method is able to achieve a higher score@1 than both KEET-based methods, and XSBench,

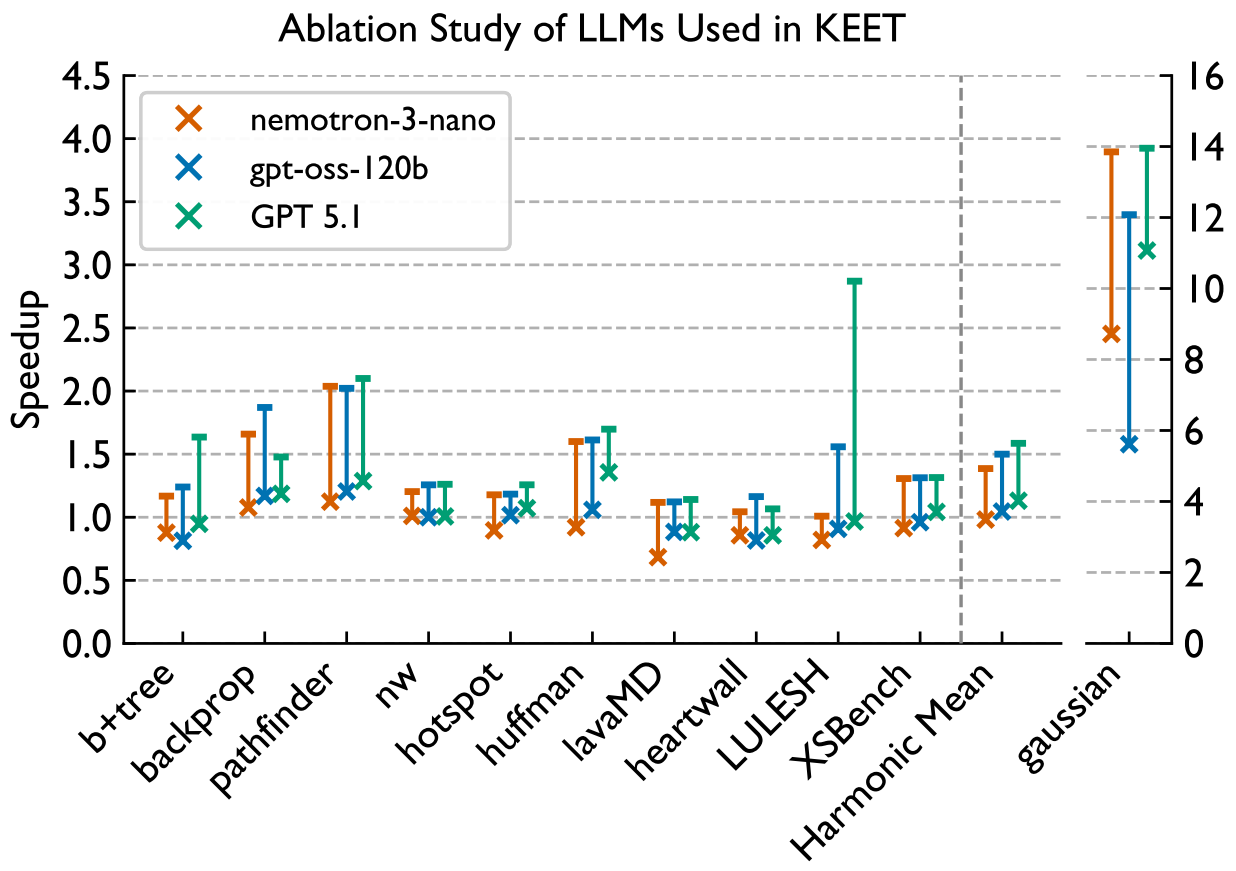


Figure 6.4: Speedup@1 by application and LLM for KEET.

where DrGPU ties with KEET. The backprop and LULESH application MCQ sets are particularly difficult across all context types, and are areas where the KEET methods provide the greatest improvement over the baseline methods. Across applications, KEET and KEET+DrGPU both achieve an average score@1 of 83%, while the nearest baseline, Code+Data, achieves 78%.

6.5.8 Downstream Task 2: Code Optimization (OPT)

For OPT, we first present pass@1 scores grouped by application and context configuration in Figure 6.6 to assess the difficulty of maintaining correctness in the optimization task. Pass@1 indicates the estimated likelihood of the LLM generating valid (compilable and test case-passing) code in its optimization attempt. These scores vary significantly across applica-

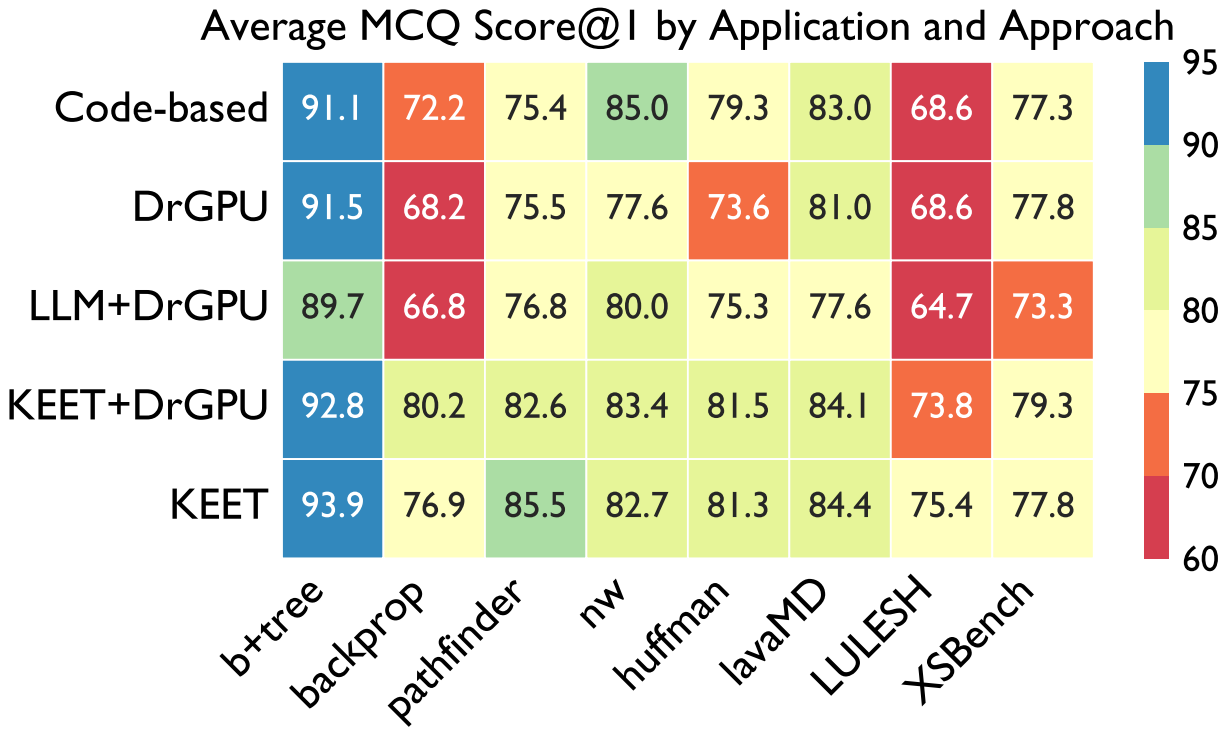


Figure 6.5: Average score@1 (out of 100) for the MCQ task grouped by application and context types.

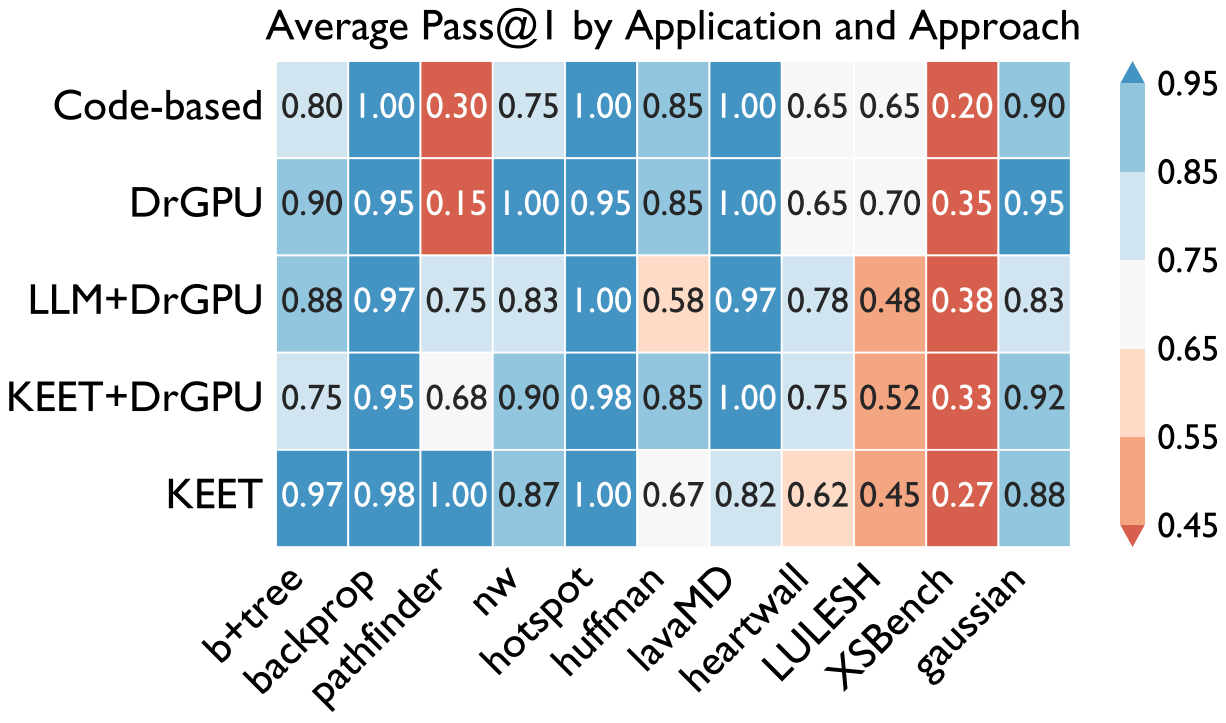


Figure 6.6: Pass@1 by application and approach.

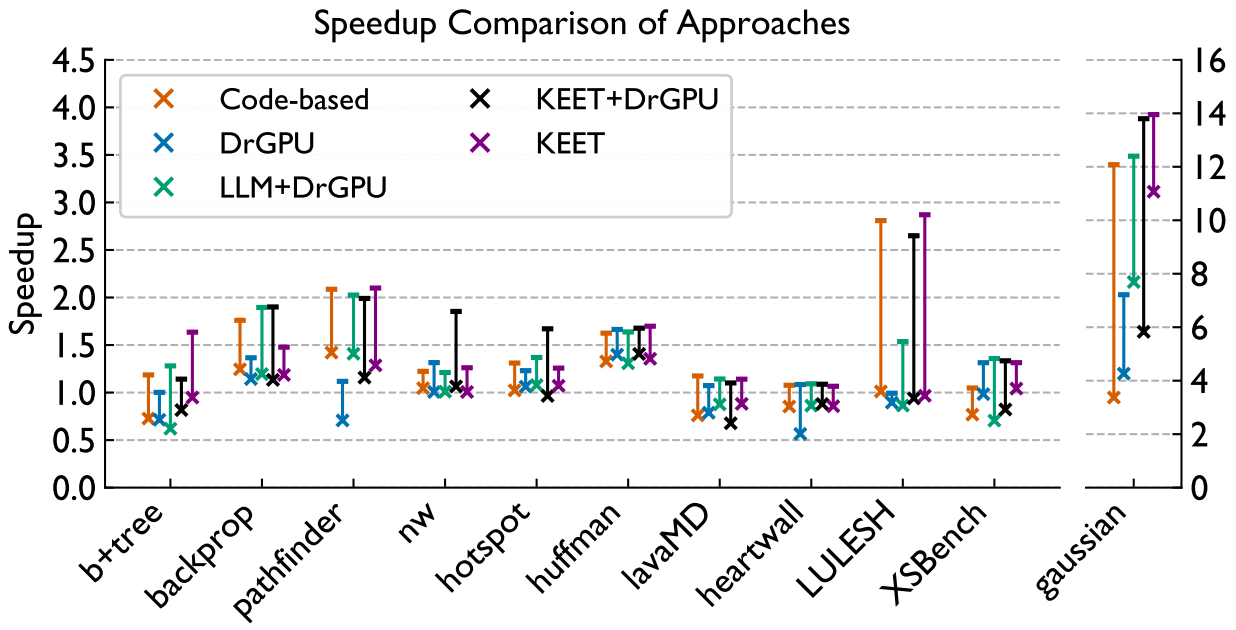


Figure 6.7: Speedup@1 by application and approach.

tions. The pathfinder, heartwall, LULESH, and XSBench applications are particularly difficult for most methods to generate code for.

Application	Best baseline technique	S/up	Best KEET technique	S/up
b+tree	Use warp primitives, reduce block size, stride loop	1.28	**Use warp primitives, reduce block size**	1.64
backprop	Remove barriers, strength reduction, use shared memory	1.90	Remove barriers, remove shared memory	1.90
pathfinder	Add __restrict__, double-buffer shared memory, remove redundant calculation, use __ldg	2.09	Add __restrict__, double-buffer shared memory, remove redundant calculation	2.10
nw	Inline helper, pad shared memory, add __restrict__, use syncwarp, unroll loop, set cache config	1.32	**Inline helper, reduce block size, pad shared memory, reduce barriers, use __ldg**	1.85
hotspot	Pad shared memory, reduce shared memory usage, add __restrict__	1.37	**Reduce shared memory usage, remove barriers, hoist invariants**	1.67
huffman	Use warp primitives, use shared memory, remove barriers	1.66	Use CUB scan, remove barriers, add macro helper	1.70
lavaMD	Add __restrict__, increase parallelism, use register accumulators, unroll loop	1.18	Hoist invariant, use register accumulators, remove barriers, unroll loop	1.14
heartwall	Add __restrict__, unroll loop	1.09	Add __restrict__, add const, unroll loop, set cache config	1.09
LULESH	Remove redundant calculation, hoist loads	2.81	**Remove redundant calculation, use __ldg**	2.87
XSBench	Replace division with fast reciprocal, use __ldg	1.36	Hoist loads to registers, force inline helpers, unroll loops	1.33
gaussian	Increase block size, change memory layout, use shared memory	12.4	**Increase block size, change memory layout, reduce grid size, hoist global load**	14.0

Table 6.4: Best baseline and KEET optimization techniques applied for each application. Techniques marked in bold are best by a significant margin (five percentage points or more) over the other technique listed for that application.

Figure 6.7 presents mean speedup@1 scores and maximum speedup by application and approach. Speedup@1 indicates the expected speedup in performance over the original (unoptimized) code achieved with one attempt. As with pass@1, these scores vary significantly across applications.

Across applications, KEET achieves the highest maximum speedup in five cases, and KEET+DrGPU achieves the highest maximum speedup in three cases. For lavaMD, the code-based method achieves the highest maximum speedup, while for XSBench, the LLM+DrGPU method achieves the highest maximum speedup. For heartwall, there is no significant difference in maximum speedup across approaches. In all cases where a KEET-based method does not achieve the highest maximum speedup, the margin between the highest baseline and highest KEET-based method is less than five percentage points. Given these results, we conclude that the most effective overall approach is KEET, with KEET+DrGPU offering additional benefits for nw and hotspot.

6.5.9 Reviewing Optimizations Applied

To understand more specifically what techniques the best optimization attempts apply to achieve strong speedups, we present the best baseline and KEET optimization techniques applied for each application in Table 6.4. Techniques that achieve at least a five percentage point greater speedup than the next best technique are marked in bold. Frequently, the best baseline and KEET technique sets have some overlap, but the KEET techniques that are most successful are more likely to use `__ldg` instructions and less likely to introduce new shared memory usage. In some cases, the speedup improvement from KEET techniques may be due to a performance

regression introduced by the baseline technique, as is the case for b+tree. For nw and hotspot, where KEET+DrGPU is most effective, we observe that both technique sets include removing barriers as an improvement over the baseline, suggesting that DrGPU is particularly helpful where KEET needs encouragement to identify unnecessary barriers. Overall, we argue that KEET is able to more effectively tune its optimization suggestions to the specific performance profile data observed for the kernel being optimized.

We also systematically analyze the optimization techniques applied against attempt outcomes across all applications and approaches, presenting the results in Figure 6.8. These optimization technique labels are automatically generated by post-processing the optimization attempt code diffs using an LLM. We observe that loop optimizations, memory qualifiers and hints, algorithmic restructuring, and strength reduction and math changes are most associated with build and run failures. Notably, thread and block configuration changes have significantly greater slowdown rates than any other optimization technique – most likely because it is generally difficult to tune block and grid size parameters based on a single profile.

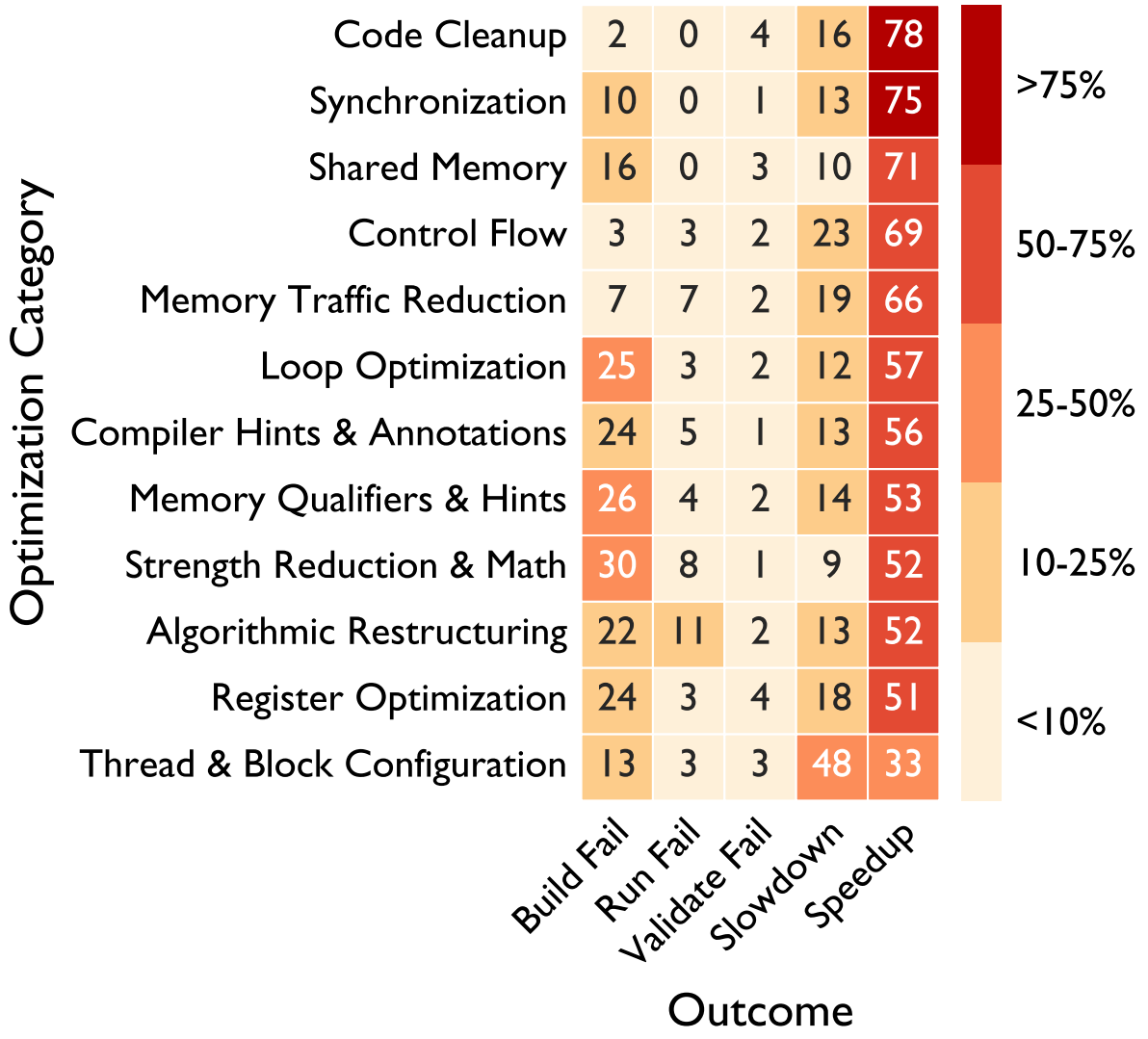


Figure 6.8: Heatmap of percentage of attempts in each outcome category that use each optimization technique.

Chapter 7: A Combined LLM-Driven Workflow for Enabling Performance Portability

In this chapter, we briefly present a workflow for enabling performance portability in a hypothetical application, combining the lessons learned and tools developed in the proceeding chapters.

7.1 Portable Programming Model Evaluation Lessons Learned

In Chapter 4, we presented a comprehensive study of the performance portability of several programming models on GPU-based leadership-class supercomputers. We used five proxy applications from a range of scientific domains, and where we observed significant performance gaps between native and portable models, we provided optimizations that improved application performance portability.

After our optimizations, a few broad outliers remain in the performance results we studied which may be of interest to developers looking to choose a programming model. We highlight the frequent gap between OpenACC and OpenMP performance on AMD systems, generally poor reduction performance in OpenACC and OpenMP, poor reductions on AMD systems with RAJA, and consistent difficulty with the compute-bound and register-intensive miniBUDE for all programming models and systems. For application, compiler, and programming model developers,

we present several insights from our experiences as well as suggestions for future investment of effort towards performance portability:

- For many application, programming model, and system combinations, we observe that performance portability is achievable, especially after applying some optimizations. We expect that as compilers and programming models continue to mature, this gap will continue to shrink in most cases.

- Successfully building all of these applications across systems is not trivial, especially for a multi-library portability suite like RAJA. Additional robustness in – and documentation for – this build process may enable app developers to more easily test competing programming models.

- Our ability to identify bottlenecks depended heavily on profiling tools. Improving the quality of these tools for new programming models and hardware architectures will be critical to enabling performance portability. Line-level stall attribution is a crucial capability missing from Omniperf at the time of writing.

- Reduction operations continue to be a major bottleneck, as observed in prior studies, and work on improving compiler handling of reductions would close some major remaining performance gaps between portable models and native baselines.

- The example of miniBUDE demonstrates that performance in a compute-bound kernel with high register pressure can be highly sensitive to the choice of a portable programming model. Identifying techniques to reduce spilling of registers to memory and instruction count bloat when adopting a programming abstraction may help users maximize arithmetic

bandwidth.

- The ability to separate correctness and performance concerns in these models was critical in identifying the optimizations we describe, as it allowed us to tune ports without invalidating scientific results. Exposing and documenting more semantic-preserving performance "knobs" within each model may provide developers with a wider range of options to improve the performance portability of their applications.

7.2 Proposed LLM-Driven Workflow

Combining the lessons learned from Chapter 4 and the tools developed in Chapters 5 and 6, we propose a workflow for enabling performance portability in a hypothetical application looking to enable performance portability across GPU architectures.

1. **Identify the current state of *functional* portability in the application and establish correctness ground truth.** The application might be CPU-only, designed for shared-memory or distributed-memory parallelism only, or provide GPU support via a vendor-specific programming model like CUDA or HIP. In the upcoming translation step, the starting state of the application may be relevant to selecting a translation tool. Furthermore, throughout the workflow, maintaining robust techniques for verifying the correctness of the application through translation and optimization is crucial.

2. **Identify the hardware platforms of interest for the application.** An application's users might be interested in running GPUs only, or GPUs and CPUs. They might be interested achieving peak performance on NVIDIA GPUs only, or achieve the best possible

performance across GPUs from NVIDIA, AMD, and possibly other GPU vendors. These requirements can impose additional constraints on the programming model selection process.

3. **Choose a programming model to convert the application to.** Chapter 4 provides a comprehensive study of the performance portability of several popular programming models. Based on the application characteristics and goals, consider the lessons learned from Chapter 4 to choose a programming model to convert the application to, aiming to maximize performance portability. For example, if the application makes heavy use of reductions and will be used on AMD GPUs, Kokkos or SYCL may be better choices than RAJA or OpenACC. To assist in making this decision, one might choose to use LLMs to create preliminary translations of the application (in whole or in part) to candidate programming models, and use the performance results to finalize the choice.

4. **Use an agentic LLM-based tool to translate the application to the chosen programming model.** To avoid the manual tedium of translation by hand, we suggest in Chapter 5 that LLM-based approaches show promise in automating the translation process.

   (a) **Select an LLM-based translation tool.** Chapter 5 presents PAREVAL-REPO, a suite of benchmarking cases for LLM-based automatic repository-scale translation between parallel programming models. As LLM-based translation tools continue to improve, we expect newer approaches to achieve increasingly impressive results on PAREVAL-REPO. Select the latest state-of-the-art method based on PAREVAL-REPO's results, ideally on a programming model translation pair relevant to the application.

(b) **Translate the application to the chosen programming model using the selected LLM-based technique.** Depending on the complexity of the application and capabilities of the translation tool, this may be more feasible in an iterative or partitioned manner. Even if the LLM-generated translation is incomplete, we argue it can act as a valuable starting point for manual or LLM-based refinement.

5. **Optimize key kernels in the translated application for performance across key target platforms.** As observed in Chapter 4, optimization can help to bridge the performance gap between portable models and native baselines. KEET, proposed in Chapter 6, can help to identify performance bottlenecks and automatically generate code changes to address them, including when given profiles across multiple architectures and configurations.

   (a) **Identify key kernels for performance optimization in the translated application on key target platforms.** After the application has been translated, we can move to improving performance (portability) on key target platforms. To employ our tool, KEET, developed in Chapter 6, we need to identify which kernel(s) to provide to KEET for analysis.

   (b) **Collect profiles of the key kernels across target platforms with a range of input parameters.** These profiles will act as input to KEET for analysis. As of the time of writing, KEET only supports Nsight Compute (NCU) profiles for NVIDIA GPUs, but work to support other GPU vendors is ongoing.

   (c) **For each kernel, use KEET to analyze collected profiles and a downstream LLM to generate improved code for the kernel.** KEET can assess performance bottlenecks and suggest code changes to address systematic performance issue that appear

across profiles. As preferred, KEET's suggestions can be implemented manually or by an LLM-based tool, as we demonstrated in Chapter 6.

(d) **Evaluate state of performance portability in the translated before and after KEET-based optimization.** If overall portability regresses, revert the optimizations and consider trying the KEET workflow again with a different underlying LLM or with DrGPU enabled, as we have observed in some cases that these settings can affect the quality of the optimization suggestions. Otherwise, the KEET-based optimization process can be repeated until improvements plateau.

6. **Document the changes made to the application and the final performance portability results.** Provide feedback on any key successes or failures in the workflow to the community, and share any best practices or lessons learned that can be applied to future similar efforts.

Chapter 8: Conclusion

In this dissertation, we have studied the problem of performance portability of GPU programming models and the use of LLMs to automate the development of portable and performant GPU codes. Chapter 4 presents our study of performance portability enabled by different programming models. Inspired by the capability of portable programming models to enable performance after the manual effort of porting, Chapter 5 presents our study of LLM-driven code translation. This includes ParEval-Repo, a proposed benchmark suite to evaluate the ability of LLMs to translate entire software repositories between parallel programming models. Considering the tedium of ensuring GPU codes continue to achieve high performance on new GPU architectures, Chapter 6 presents our study of GPU performance analysis using LLMs. This study focused on a new LLM-based agentic framework, KEET, to analyze GPU kernel performance and provide suggestions that help LLMs generate performance-optimized code. Bringing together all the proceeding chapters, Chapter 7 presents a workflow for enabling performance portability in a hypothetical application, employing the lessons learned and tools developed in the proceeding chapters.

This dissertation builds a foundation for future work in enabling performance portability in scientific applications with the assistance of Large Language Models. ParEval-Repo provides goalposts to guide future improvements to techniques for LLM code translation. The

results of our study comparing portable programming models and tools developed to maximize scores on PAREVAL-REPO can achieve a synergistic effect. Application developers can improve their own application's performance portability and contribute additional insight into the broader understanding of performance portability of different programming models by easily generating new ports of their application to multiple portable programming models. Meanwhile, continued development of KEET can help maintain a high degree of performance in these applications on new GPU architectures by providing actionable insights from performance profiles of their application's kernels. As LLM technologies continue to improve, we expect that the scope of KEET and tools like it can expand to include support for not just additional GPU architectures and programming models, but also broader performance analysis tasks beyond individual GPU kernels. Finally, the workflow presented in Chapter 7 can serve as a roadmap for future case studies examining how well these techniques work in real-world application development.

Bibliography

[1] Laksono Adhianto et al. “HPCToolkit: Tools for performance analysis of optimized parallel programs”. In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[2] Laksono Adhianto et al. “Refining HPCToolkit for application performance analysis at exascale”. In: *The International Journal of High Performance Computing Applications* 38.6 (2024), pp. 612–632.

[3] Waseem Ahmed et al. “A Framework for Faster Porting of Scientific Applications Between Heterogeneous Clouds”. In: *Smart Societies, Infrastructure, Technologies and Applications*. Ed. by Rashid Mehmood et al. Cham: Springer International Publishing, 2018, pp. 27–43. ISBN: 978-3-319-94180-6.

[4] Victor Artigues et al. “Evaluation of performance portability frameworks for the implementation of a particle-in-cell code”. In: *Concurrency and Computation: Practice and Experience* 32.11 (2020), e5640.

[5] Seonmyeong Bak et al. “OpenMP application experiences: Porting to accelerated nodes”. In: *Parallel Computing* 109 (2022), p. 102856.

[6] D. A. Beckingsale et al. “Umpire: Application-focused management and coordination of complex hierarchical memory”. In: *IBM Journal of Research and Development* 64.3/4 (2020), 00:1–00:10. DOI: 10.1147/JRD.2019.2954403.

[7] David Beckingsale et al. “Performance portable C++ programming with RAJA”. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019, pp. 455–456.

[8] Tal Ben-Nun et al. “Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–14.

[9] Claude Bernard et al. “Studying quarks and gluons on MIMD parallel computers”. In: *The International Journal of Supercomputing Applications* 5.4 (1991), pp. 61–70.

[10] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. “Hatchet: Pruning the Overgrowth in Parallel Profiles”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. LLNL-CONF-772402. Nov. 2019. URL: http://doi.acm.org/10.1145/3295500.3356219.

[11] Abhinav Bhatele et al. *Pipit: Scripting the analysis of parallel execution traces*. 2023. arXiv: 2306.11177 [cs.DC].

[12] Swen Boehm et al. “Evaluating performance portability of accelerator programming models using SPEC ACCEL 1.2 benchmarks”. In: *High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers 33*. Springer. 2018, pp. 711–723.

[13] Holger Brunst et al. “First experiences in performance benchmarking with the new SPEChpc 2021 suites”. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. 2022, pp. 675–684.

[14] Bowen Cui et al. *Do Large Language Models Understand Performance Optimization?* 2025. arXiv: 2503.13772 [cs.DC]. URL: https://doi.org/10.48550/arXiv.2503.13772.

[15] Weinan Dai et al. “CUDA Agent: Large-Scale Agentic RL for High-Performance CUDA Kernel Generation”. In: *arXiv preprint arXiv:2602.24286* (2026). URL: https://doi.org/10.48550/arXiv.2602.24286.

[16] Daniela F. Daniel and Jairo Panetta. “On Applying Performance Portability Metrics”. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, pp. 50–59. DOI: 10.1109/P3HPC49587.2019.00010.

[17] Joshua H Davis et al. “Porting a computational fluid dynamics code with amr to large-scale gpu platforms”. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2023, pp. 602–612.

[18] Joshua Hoke Davis et al. “Performance Assessment of OpenMP Compilers Targeting NVIDIA V100 GPUs”. In: *Accelerator Programming Using Directives*. Ed. by Sridutt Bhalachandra et al. Cham: Springer International Publishing, 2021, pp. 25–44. ISBN: 978-3-030-74224-9.

[19] Tom Deakin et al. “Analyzing Reduction Abstraction Capabilities”. In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2021, pp. 33–44.

[20] Tom Deakin et al. “Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream”. In: *Int. J. Comput. Sci. Eng.* 17.3 (Jan. 2018), 247–262. ISSN: 1742-7185.

[21] Tom Deakin et al. “Heterogeneous Programming for the Homogeneous Majority”. In: *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2022, pp. 1–13. DOI: 10.1109/P3HPC56579.2022.00006.

[22] Tom Deakin et al. “Performance Portability across Diverse Computer Architectures”. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, pp. 1–13. DOI: 10.1109/P3HPC49587.2019.00006.

[23] Tom Deakin et al. “Tracking Performance Portability on the Yellow Brick Road to Exascale”. In: *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2020, pp. 1–13. DOI: 10.1109/P3HPC51967.2020.00006.

[24] Matthew T Dearing et al. “LASSI: An LLM-based Automated Self-Correcting Pipeline for Translating Parallel Scientific Codes”. In: *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. IEEE. 2024, pp. 136–143.

[25] DeepSeek-AI. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[26] Javier Delgado et al. “A case study on porting scientific applications to GPU/CUDA”. In: *Journal of Computational Interdisciplinary Sciences* 2.1 (2011), pp. 3–11.

[27] Akash Dhruv and Anshu Dubey. *Leveraging Large Language Models for Code Translation and Software Development in Scientific Computing*. 2025. arXiv: 2410.24119 [cs.SE]. URL: https://arxiv.org/abs/2410.24119.

[28] Wei Ding et al. “KLONOS: Similarity-based planning tool support for porting scientific applications”. In: *Concurrency and Computation: Practice and Experience* 25.8 (2013), pp. 1072–1088.

[29] Douglas Doerfler and Christopher Daley. *su3_bench: Lattice QCD SU (3) matrix-matrix multiply microbenchmark (su3_bench) v1. 0*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2020.

[30] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. Tech. rep. 2024.

[31] Amanda S Dufek et al. “Case study of using Kokkos and SYCL as performance-portable frameworks for Milc-Dslash benchmark on NVIDIA, AMD and Intel GPUs”. In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2021, pp. 57–67.

[32] *ECP Proxy Applications*. https://proxyapps.exascaleproject.org/. Accessed: 2023-09-30.

[33] Lieven Eeckhout. “R.I.P. Geomean Speedup Use Equal-Work (Or Equal-Time) Harmonic Mean Speedup Instead”. In: *IEEE Computer Architecture Letters* 23.1 (2024), pp. 78–82.

[34] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD’96. Portland, Oregon: AAAI Press, 1996, 226–231.

[35] Robert D Falgout et al. “Porting hypre to heterogeneous computer architectures: Strategies and experiences”. In: *Parallel Computing* 108 (2021), p. 102840.

[36] T. Gamblin et al. “The Spack package manager: bringing order to HPC software chaos”. In: *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2015. DOI: 10.1145/2807591.2807623. URL: https://doi.ieeecomputersociety.org/10.1145/2807591.2807623.

[37] Rahulkumar Gayatri et al. “A case study for performance portability using OpenMP 4.5”. In: *Accelerator Programming Using Directives: 5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings 5*. Springer. 2019, pp. 75–95.

[38] Markus Geimer et al. “The Scalasca performance toolset architecture”. In: *Concurrency and computation: Practice and experience* 22.6 (2010), pp. 702–719.

[39] Wolfgang Gentzsch. “Porting applications to grids and clouds”. In: *International Journal of Grid and High Performance Computing (IJGHPC)* 1.1 (2009), pp. 55–77.

[40] William F Godoy et al. “Large language model evaluation for high-performance computing software development”. In: *Concurrency and Computation: Practice and Experience* 36.26 (2024), e8269. URL: https://doi.org/10.1002/cpe.8269.

[41] Yueming Hao et al. “Drgpu: A top-down profiler for gpu applications”. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 43–53. URL: https://doi.org/10.1145/3578244.3583736.

[42] Stephen Lien Harrell et al. “Effective performance portability”. In: *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2018, pp. 24–36.

[43] JA Herdman et al. “Accelerating hydrocodes with OpenACC, OpenCL and CUDA”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 465–471.

[44] Michael Allen Heroux et al. *Mantevo Suite 1.0*. Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2013.

[45] Jose Herrera et al. “Porting of scientific applications to grid computing on gridway”. In: *Scientific Programming* 13.4 (2005), pp. 317–331.

[46] *Holistic Trace Analysis (HTA): A library to analyze PyTorch traces*. 2023. URL: https://github.com/facebookresearch/HolisticTraceAnalysis.

[47] Rich D. Hornung and Jeff A. Keasler. *The RAJA Portability Layer: Overview and Status*. Tech. rep. LLNL-TR-661403. Lawrence Livermore National Laboratory, Sept. 2014.

[48] Ali Reza Ibrahimzada et al. *Repository-level compositional code translation and validation*. 2024. arXiv: 2410.24117 [cs.SE]. URL: https://arxiv.org/abs/2410.24117.

[49] Ian Karlin et al. “Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application”. In: *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’13. IEEE Computer Society, May 2013.

[50] Maximilian Keiff et al. “Automated performance analysis tools framework for HPC programs”. In: *Procedia Computer Science* 207 (2022), pp. 1067–1076.

[51] Tuomas Koskela et al. “Principles for automated and reproducible benchmarking”. In: *Proceedings of the SC’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 609–618.

[52] Jens Krüger and Rüdiger Westermann. “Linear algebra operators for GPU implementation of numerical algorithms”. In: *ACM SIGGRAPH 2005 Courses*. 2005, 234–es.

[53] JaeHyuk Kwack et al. “Evaluation of Performance Portability of Applications and Mini-Apps across AMD, Intel and NVIDIA GPUs”. In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2021, pp. 45–56. DOI: 10.1109/P3HPC54578.2021.00008.

[54] Robert Tjarko Lange et al. “Towards robust agentic cuda kernel benchmarking, verification, and optimization”. In: *arXiv preprint arXiv:2509.14279* (2025). URL: https://doi.org/10.48550/arXiv.2509.14279.

[55] Bin Lei et al. *Creating a Dataset for High-Performance Computing Code Translation using LLMs: A Bridge Between OpenMP Fortran and C++*. 2023. arXiv: 2307.07686 [cs.SE]. URL: https://arxiv.org/abs/2307.07686.

[56] Wei-Chen Lin, Simon McIntosh-Smith, and Tom Deakin. “Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC”. In: *arXiv preprint arXiv:2401.02680* (2024).

[57] *Llama 3.3 Model Card and Prompt Format*. 2024. URL: https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/.

[58] Nicholas Malaya et al. “Experiences readying applications for exascale”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023, pp. 1–13.

[59] Ami Marowka. “A comparison of two performance portability metrics”. In: *Concurrency and Computation: Practice and Experience* (2023), e7868.

[60] Ami Marowka. “Toward a better performance portability metric”. In: *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2021, pp. 181–184.

[61] Matthew Martineau, Simon McIntosh-Smith, and Wayne Gaudin. “Assessing the performance portability of modern parallel programming models using TeaLeaf”. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017), e4117.

[62] Simon McIntosh-Smith et al. “High performance in silico virtual drug screening on many-core processors”. In: *The international journal of high performance computing applications* 29.2 (2015), pp. 119–134.

[63] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781.

[64] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. *LLM4VV: Developing LLM-Driven Testsuite for Compiler Validation*. 2023. arXiv: 2310.04963 [cs.AI].

[65] *NERSC Proxy Suite*. https://www.nersc.gov/research-and-development/nersc-proxy-suite/.

[66] Daniel Nichols et al. “Can Large Language Models Write Parallel Code?” In: *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’24. New York, NY, USA: Association for Computing Machinery, 2024. URL: https://doi.org/10.1145/3625549.3658689.

[67] NVIDIA. *NVIDIA Nsight Compute*. https://developer.nvidia.com/nsight-compute.

[68] *o3 and o4-mini System Card*. 2025. URL: https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf.

[69] OpenAI, Aaron Hurst, and et al. *GPT-4o System Card*. 2024. arXiv: 2410.21276 [cs.CL]. URL: https://arxiv.org/abs/2410.21276.

[70] *OpenMP Application Program Interface. Version 4.0. July 2013*. 2013.

[71] S. John Pennycook and Jason D. Sewall. "Revisiting a Metric for Performance Portability". In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2021, pp. 1–9. DOI: 10.1109/P3HPC54578.2021.00004.

[72] S John Pennycook et al. "Navigating performance, portability, and productivity". In: *Computing in Science & Engineering* 23.5 (2021), pp. 28–38.

[73] Simon J Pennycook, Jason D Sewall, and Victor W Lee. "A metric for performance portability". In: *Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. 2016. URL: https://arxiv.org/abs/1611.07409.

[74] Simon J Pennycook, Jason D Sewall, and Victor W Lee. "Implications of a metric for performance portability". In: *Future Generation Computer Systems* 92 (2019), pp. 947–958.

[75] Vincent Pillet et al. "PARAVER: A tool to visualize and analyze parallel code". In: *WoTUG-18* 44 (Mar. 1995).

[76] *QwQ: Reflect Deeply on the Boundaries of the Unknown*. 2024. URL: https://qwenlm.github.io/blog/qwq-32b-preview/.

[77] Esteban Miguel Rangel et al. "A Performance-Portable SYCL Implementation of CRK-HACC for Exascale". In: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 1114–1125.

[78] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. "Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622813.

[79] István Z Reguly. "Performance portability of multi-material kernels". In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2019, pp. 26–35.

[80] István Z Reguly and Gihan R Mudalige. "Productivity, performance, and portability for computational fluid dynamics applications". In: *Computers & Fluids* 199 (2020), p. 104425.

[81] Ruyman Reyes and Victor Lomüller. "SYCL: Single-source C++ accelerator programming". In: *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[82] *ROCm Compute Profiler*. Accessed: 2026-04-09. URL: https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/index.html.

[83] Paul K Romano et al. “OpenMC: A state-of-the-art Monte Carlo code for research and development”. In: *Annals of Nuclear Energy* 82 (2015), pp. 90–97.

[84] Amit Sabne et al. “Evaluating performance portability of OpenACC”. In: *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers 27*. Springer. 2015, pp. 51–66.

[85] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. “CATS: Memory and Control Flow Tracing for Whole-Program Performance Analysis”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2025, pp. 331–348.

[86] Ada Sedova et al. “High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability”. In: *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2018, pp. 1–13.

[87] Jason Sewall et al. “Interpreting and Visualizing Performance Portability Metrics”. In: *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2020, pp. 14–24. DOI: 10.1109/P3HPC51967.2020.00007.

[88] Gemini Team. *Gemini: A Family of Highly Capable Multimodal Models*. 2023. arXiv: 2312.11805 [cs.CL].

[89] Ali Tehrani et al. “CodeRosetta: Pushing the Boundaries of Unsupervised Code Translation for Parallel Programming”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 100965–100999.

[90] TOP500.org. *June 2024 TOP500*. 2024. URL: https://www.top500.org/lists/top500/2024/06/ (visited on 08/13/2024).

[91] TOP500.org. *November 2025 TOP500*. 2025. URL: https://www.top500.org/lists/top500/2025/11/ (visited on 11/17/2025).

[92] John R. Tramm et al. “A task-based parallelism and vectorized approach to 3D Method of Characteristics (MOC) reactor simulation for high performance computing architectures”. In: *Computer Physics Communications* 202 (2016), pp. 141 –150. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2016.01.007. URL: http://www.sciencedirect.com/science/article/pii/S0010465516000266.

[93] John R Tramm et al. “XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis”. In: *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[94] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[95] Pedro Valero-Lara et al. “ChatBLAS: The First AI-Generated and Portable BLAS Library”. In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2024, pp. 19–24. URL: https://doi.org/10.1109/SCW63240.2024.00010.

[96] Ben Van Werkhoven and Pieter Hijma. “An integrated approach to porting large scientific applications to GPUs”. In: *2015 IEEE 11th International Conference on e-Science*. IEEE. 2015, pp. 57–66.

[97] Yanli Wang et al. *RepoTransBench: A Real-World Benchmark for Repository-Level Code Translation*. 2024. arXiv: 2412.17744 [cs.SE]. URL: https://arxiv.org/abs/2412.17744.

[98] Michael Wolfe et al. “The OpenACC data model: Preliminary study on its major challenges and implementations”. In: *Parallel Computing* 78 (2018), pp. 15–27.

[99] John Yang et al. “Swe-agent: Agent-computer interfaces enable automated software engineering”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 50528–50652.

[100] Yang Yu et al. “Towards Automated Kernel Generation in the Era of LLMs”. In: *arXiv preprint arXiv:2601.15727* (2026). URL: https://doi.org/10.48550/arXiv.2601.15727v1.

[101] Mohammad Zaeed, Tanzima Z Islam, and Vladimir Inđić. “Opal: A Modular Framework for Optimizing Performance using Analytics and LLMs”. In: *arXiv preprint arXiv:2510.00932* (2025). URL: https://doi.org/10.48550/arXiv.2510.00932.

[102] Zijian Zhang et al. “Cudaforge: An agent framework with hardware feedback for cuda kernel optimization”. In: *arXiv preprint arXiv:2511.01884* (2025). URL: https://doi.org/10.48550/arXiv.2511.01884.

[103] Qidong Zhao, Milind Chabbi, and Xu Liu. “EasyView: Bringing Performance Profiles into Integrated Development Environments”. In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2024, pp. 386–398.

[104] Keren Zhou et al. “An automated tool for analysis and tuning of gpu-accelerated code in hpc applications”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2021), pp. 854–865. URL: https://doi.org/10.1109/TPDS.2021.3094169.

[105] Keren Zhou et al. “Gpa: A gpu performance advisor based on instruction sampling”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 115–125. URL: https://doi.org/10.1109/CGO51591.2021.9370339.