CS 5914: High-Performance Code Generation Using LLMs

Tensor Reduction

Final Project Report

Prayash Joshi
Najibul Haque Sarker

Department of Computer Science
Virginia Tech

May 06, 2025

Contents

1 Team Information

Team	Name	Type
Red	Najibul Haque Sarker	Manual Optimization
Blue	Prayash Joshi	Optimization through LLMs

Link to code: Github

2 Introduction & Motivation

Tensor reduction is a fundamental operation in many scientific computing workloads. Reduction functions are operations that take a collection of values (like a list, array, or tensor) and reduce them to a single summary value or a smaller set of values by repeatedly applying a specific rule or method. This specific rule/method can be the summation, minimum, maximum, multiplication, etc functions. Similarly, tensor reduction is the process of applying a reduction function across one or more dimensions of a tensor, resulting in a smaller tensor or a scalar. This operation holds great significance due to being a fundamental operation that is used in machine learning, more specifically in tasks such as aggregating gradient values, summing loss functions, computing statistical measures, and aggregating activations. In the domain of high-performance computing (HPC) and large-scale neural networks, such as Large Language Models (LLMs), the performance of reduction operations becomes a critical determinant of the overall computational efficiency. When working with massive datasets or model architectures containing billions of parameters, inefficiencies in reduction operations can lead to significant slowdowns, power wastage, and memory bottlenecks. In this project, we focus on one specific operation of tensor reduction: the summation function.

This project explores both manual and automated optimization of tensor reduction kernels targeting GPU accelerators. Here, the choice of optimizing for GPUs is taken due to the significant role these accelerators play in the training and inference of machine learning models. As GPU accelerators are optimized for specific computations relevant to model training and serving, optimizing reduction kernels for these accelerators is the most relevant usecase. In the manual optimization track, we create kernel implementations using targetted CUDA programming to achieve high throughput and minimize latency across a range of input sizes. A lot of research have already been done in this space, and most of the manual optimizations introduced in this paper are adapted from existing research. However, manual optimization is time-consuming, requires domain-specific expertise, and is less scalable across different tasks and hardware.

The second part of our project focuses on automated code generation and optimization utilizing Large Language Models. Writing CUDA code is notoriously challenging due to the need for low-level control over thread hierarchies, memory access patterns, synchronization mechanisms, and hardware-specific optimization strategies. Developers must manually manage resources like shared memory, ensure memory coalescing, avoid warp divergence, and tune thread block sizes to extract peak performance from GPUs. These complexities create a steep learning curve and often require deep expertise in parallel computing and GPU architecture. Automating this process using

Large Language Models (LLMs) presents a promising path forward as LLMs can learn from vast amounts of high-performance CUDA code and generate optimized kernels tailored to specific problems. By abstracting away the intricacies of thread-level parallelism and hardware tuning, LLM-driven code generation could dramatically lower the barrier to entry for GPU programming and accelerate the development of efficient high-performance computing (HPC) applications. This is the primary motivation for this project, where we attempt to answer the question whether LLMs automatically generate efficient and correct GPU code that approaches the performance of hand-optimized implementations.

The motivation for comparing manual and LLM-based approaches stems from the observation that while auto-generated kernels may be syntactically correct and easy to produce, they often lack performance-tuned details crucial for HPC environments. Here, we attempt to bridge the gap between both manual and LLM-based approaches. In this dual-focused project, we present a side-by-side study of both tracks. We provide multiple versions of a manually optimized CUDA reduction kernel (v1 to v4), and contrast them with kernels synthesized using state-of-the-art LLMs. Our contributions are thus twofold: 1) a rigorous empirical comparison of human-optimized versus LLM-generated kernels for tensor reduction task (summation) in GPUs, and 2) a reproducible benchmarking and profiling framework to support future research in this space.

3 Background & Related Work

3.1 Manual Optimization

There have been a substantial amount of work already done in the manual optimization space for high-performance code in GPUs, both targeting hardware and software optimizations [1, 2, 3, 4, 5, 6, 7, 8]. Among these, [2] details the complexity involved with optimizing for highly-parallel systems and introduces metrics for optimizing matrix multiplication. [1] discusses the required balance of each GPU thread's resource usage and number of simultaneously active threads during optimization. The paper also discusses several optimization methods including the utilization of shared memory for threads, decreasing warp divergence, and unrolling techniques. [3] depicts approaches for parallelizing reductions in modern GPUs by utilizing locking mechanisms, while [4] factors in branch divergence. More recent works combine loop unrolling with persistent threads [7]. However, the CUDA guide to parallel reduction optimization by Mark Harris [9] showcases these various optimization techniques utilizing a synthetic benchmark, and remains one of the top resources in this area.

3.2 Optimization through LLMs

Large Language Models (LLMs) for code generation and optimization has become not only a promising approach to automate complex programming tasks but also be a programmer to its own right. Benchmarks have demonstrated that LLMs are especially proficient in languages it has seen the most in its training, ie. Python, Java, C, etc. We are interested in exploring the use of these foundational and open-source models in specialized domains like cuda kernal programming. A recent article from Sakana AI

discusses how LLMs can successfully generate and optimize code for high-performance computing applications, including CUDA kernels for GPU acceleration.

The challenge here is the inherent complexity of CUDA programming. Lange et al. notes that "the skill set required to balance [performance] trade-offs and to engineer efficient Compute Unified Device Architecture (CUDA) kernels is rare and highly in demand, encompassing algorithmic, hardware, and instruction set knowledge"[10]. Compared to Pytorch, written in python, cuda kernals leverages fine-grained Parallelism, utilization of memory heirarchy, thread synchronization, etc. to on GPU hardware to achieve speedups.

Some papers have highlighted capabilities of LLMs in generating high-performance code. Chen et al. [11] looked at the task of evaluating performance of different LLMs for code generation tasks. The findings indicate that these models can produce syntactically correct and functionally accurate code, including in C++ with CUDA extensions. Nijkamp et al. [12] explored the potential of LLMs specifically for code generation tasks, showing improvements in both correctness and efficiency. But there have not been many studies prior to 2025 looking at actual high-quality performative cuda code generation.

Recent work by Lange et al. [10] introduced "THE AI CUDA ENGINEER," an end-to-end agentic workflow that translates PyTorch code to working CUDA kernels and optimizes their runtime performance. They use techniques like LLM ensembling, iterative profiling feedback loops, and kernel crossover optimization. Their work demonstrates that LLMs can generate correct CUDA code (achieving ¿91% correctness rate. During thier evaluation, they were able to produce optimized implementations that outperform standard PyTorch operations, with a median speedup of 1.52x.

The effectiveness of evolutionary approaches to code optimization is evident. Using LLMs as a "variation-generating engines" within an iterative refinement process made these systems propose new implementations and optimizations as different starting seeds.Lange et al. [10] discusses that "unlike random search, LLMs incorporate strong algorithmic priors, allowing them to generate executable programs, testable simulations, and provable mathematical statements, significantly accelerating the search for novel solutions."

Prior work has focused on translating high-level frameworks like PyTorch to CUDA. The goal of this paper is to further investigate whether LLMs can automatically generate efficient and correct GPU code for tensor reduction operations that approaches the performance of hand-optimized implementations.

4 System Design / Approach

4.1 Manual Optimization

The manual optimization process is based on multiple incremental optimization approaches focused on targeted metrics. The strategies utilized in each of the steps are detailed below:

4.1.1 Manual Strategy 1: Baseline Parallel Reduction (Interleaved Addressing)

Our initial baseline uses a vanilla parallel sum reduction algorithm [13]. While there are serial sum reduction techniques available, using a parallel variation of tensor reduction as the baseline is more applicable as we are using GPU architecture for optimization. Here, the reduction problem is divided and operated in groups recursively until the final result remains.

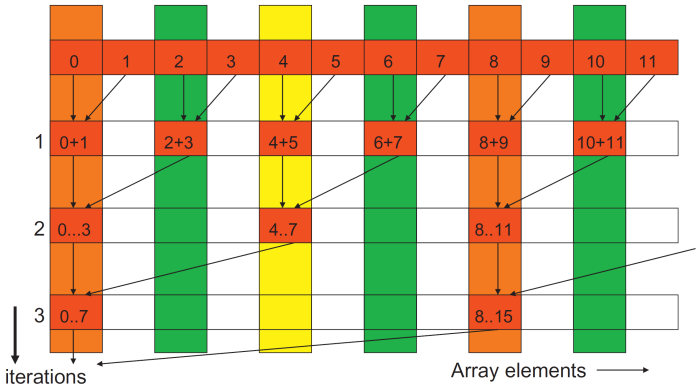


Figure 1: Baseline Parallel Sum Reduction. Figure from [13].

As shown in Figure 1, if given an array of size N, this algorithm parallelly launches $\frac{N}{2}$ threads where each thread processes two numbers and adds the sum to the first number in the first iteration. During the second iteration, only the even threads are active, and use the values from the first iteration to compute the sum again. These iterations are continued until there is a single thread left, which computes the final sum. This method uses interleaved addressing, where threads add two neighboring elements together.

Here in each iteration, half the number of threads from previous iterations are becoming inactive. In GPU architectures, a group of 32 threads called a warp execute simultaneously. In this setting, each warp will have some threads active and other threads inactive as iterations increase, which causes warp divergence.

4.1.2 Manual Strategy 2a: Serial Addressing

To overcome the previous warp divergence issue, a new strategy named serial addressing is used. Here, the threads are initialized for the first $n/2$ values present in a block of size n. Now, instead of adding neighbor values, the threads add values that starts from half a section away. Basically, the access pattern becomes more regular: thread 0 adds thread s, thread 1 adds thread s+1, etc. All the pair-wise sums would then be stored in the first half of the block, after the first iteration, and this will be halved for each iteration as shown in Figure. 2.

Here the performance upgrade comes from the position of the thread. Now, due to the position of the threads, the group of threads executing the same operation becomes contiguous. In each step, the left half of the threads in the block actively add the values of the right half. This eliminates warp-level divergence, because all threads in a given

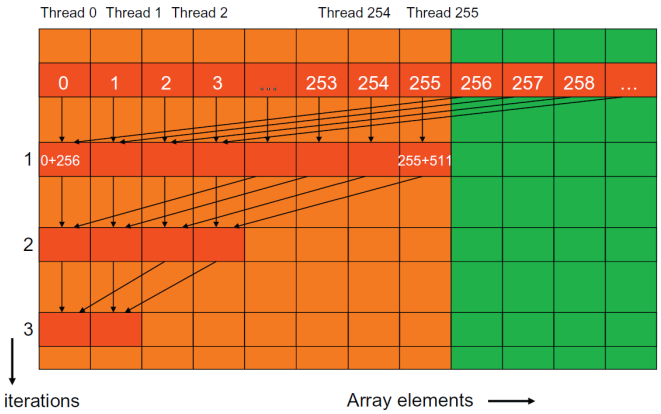


Figure 2: Serial Addressing in Parallel Sum Reduction. Figure from [13].

warp either execute the summation operation or don't, as a group. Thus, this process should result in thread divergence optimization.

4.1.3 Manual Strategy 2b: Shared Memory

The GPU memory architecture is divided into several layers, where global memory is accessible by all but has the highest overhead, while shared memory is accessible by threads in a block but has lower read/write overhead (shown in Figure 3). The vanilla implementation utilized global memory for the reduction operation, which has high read/write latency. This strategy incorporated the use of **shared memory** [13, 9], where the tensor elements were transferred from global memory to shared memory, the operation was done in shared memory, and the resultant summation was again transferred back to global memory.

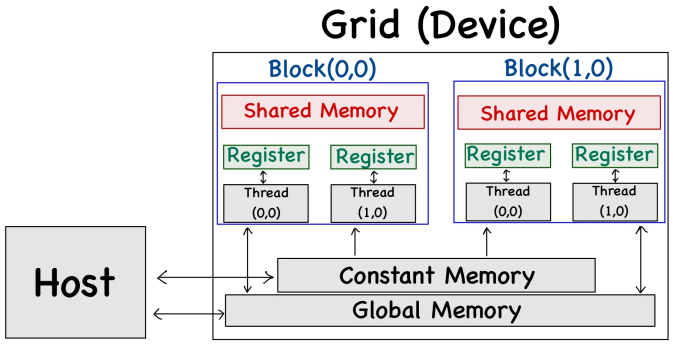


Figure 3: Memory layers in GPU architecture.

4.1.4 Manual Strategy 2c: Initial sum during memory transfer

A simple optimization that can be done is during the transfer between global and shared memory, the first step of reduction can be executed. For example, always reading two elements from global memory and then writing the summed version in shared memory. This decreases shared memory requirements from N to N/2. Illustrated in Figure 4.

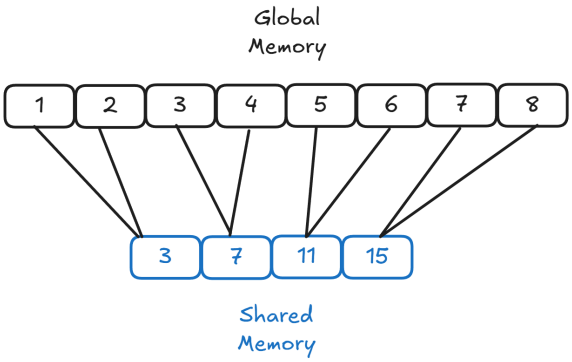


Figure 4: Initial summation during memory transfer.

4.1.5 Manual Strategy 3a: Last Warp Unrolling

As the reduction process continues, the number of active threads decreases. To sync between all these threads, the method `_syncthreads()` needs to be executed, which is costly and has overhead. After a number of iterations, there will be a time where only 32 threads or a wrap will be active. In a warp, all the instructions are synchronous, so calling the costly `_syncthreads()` method on the last 32 threads can be avoided. Thus, in the reduction loop, the last 6 iterations can be unrolled and written manually for faster execution [9]. This also saves useless work in all warps, as all warps have to execute every iteration of the loop and condition statements. The pseudocode is shown in Listing 1.

Listing 1: Last Warp Unrolling Pseudocode

```
1
2  function summation_kernel:
3      ...
4      # Tree Reduction (until 32 threads left)
5      for stride=blocksize/2 .. 32:
6          if (thread_ID < stride):
7              shared_memory[thread_ID] += shared_memory[thread_ID
                   + stride]
8          _syncthreads()
9
10     # for last remaining 32 threads -> unroll loop
11     if thread_ID < 32:
12         warpReduceSum(thread_ID)
13
14
15 function warpReduceSum:
16     # add accordingly for last 32 threads without any syncthread
           call
17     shared_memory[thread_ID] += shared_memory[thread_ID + 32]
18     shared_memory[thread_ID] += shared_memory[thread_ID + 16]
19     shared_memory[thread_ID] += shared_memory[thread_ID + 8]
20     shared_memory[thread_ID] += shared_memory[thread_ID + 4]
21     shared_memory[thread_ID] += shared_memory[thread_ID + 2]
```

```
    shared_memory[thread_ID] += shared_memory[thread_ID + 1]
```

4.1.6 Manual Strategy 3b: Warp shuffle instructions

Intra-warp communication can be done more efficiently using warp shuffle instructions rather than shared memory. So instead of directly using shared memory for the last warp method, shuffle instructions such as `_shfl_down_sync` can be utilized. From the pseudocode, the function `warpReduceSum` will be changed according to Listing 2.

Listing 2: Warp Shuffle Instructions

```
function warpReduceSum:
    # using warp shuffle instructions
    shared_memory[thread_ID] += __shfl_down_sync[thread_ID + 32]
    shared_memory[thread_ID] += __shfl_down_sync[thread_ID + 16]
    shared_memory[thread_ID] += __shfl_down_sync[thread_ID + 8]
    shared_memory[thread_ID] += __shfl_down_sync[thread_ID + 4]
    shared_memory[thread_ID] += __shfl_down_sync[thread_ID + 2]
    shared_memory[thread_ID] += __shfl_down_sync[thread_ID + 1]
```

4.1.7 Complete loop unrolling

To maximize efficiency, the reduction loop can be fully unrolled, which can only be done if the number of iterations can be known during compile time [9]. In order to execute this logic, CUDA is integrated with C++ templates or launch-time constants which is used to know the current iteration and blocksize. Here, the maximum block size can be set beforehand, and is set as 512 in this experiment. On the other hand, the current block size can be known using templates. The idea is to eliminate all loop overhead and make every memory access pattern a compile-time decision, which allows the compiler to optimize and schedule instructions most effectively.

4.2 Optimization through LLMs

For optimizing tensor reduction operations using LLMs, we explored a systematic approach that leverages both single-shot and multi-seed generation strategies. Our approach consists of two main experimental designs:

4.2.1 Zero-Shot Generation Approach

Our initial approach involved querying each LLM once with a generic prompt for sum reduction. In this "zero-shot" approach, we provided the model with the required function signature and basic requirements, asked for implementation of sum reduction without specific optimization hints, and verified the correctness of the generated kernel on various input sizes. This approach, illustrated in Figure 5, allowed us to establish a baseline for what LLMs could generate without extensive guidance.

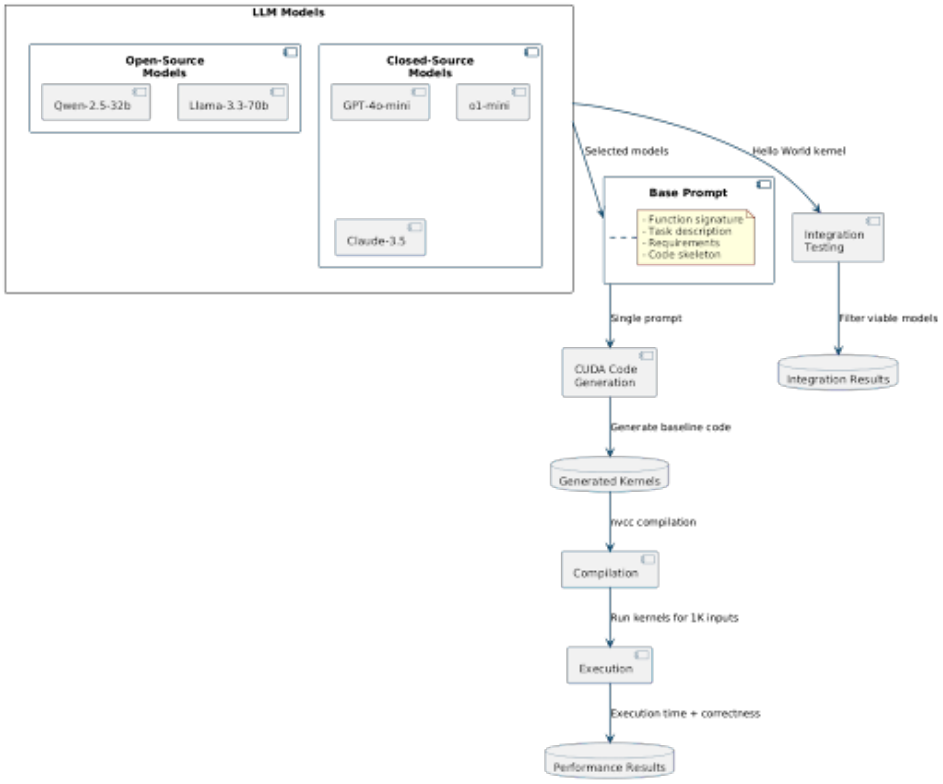


Figure 5: Zero-shot CUDA kernel generation workflow using LLMs

4.2.2 Multi-Seed Generation Approach

Building on insights from the zero-shot approach, we developed a more sophisticated multi-seed generation strategy, illustrated in Figure 6, which consisted of several components. First, we employed iterative prompting, where we generated multiple CUDA kernel variants (seeds) from each model by iteratively refining our prompts with optimization hints (e.g., "use shared memory and unroll the last warp"). We then implemented a feedback loop, providing insights from prior implementations to the models (e.g., "this code failed to compile" or "execution time is slow, try a grid-stride loop"), leading to generation of additional seeds. Finally, we used cross-technique prompting, instructing models to incorporate techniques from the manual optimization approaches, enabling direct comparison with hand-tuned versions.

This approach allowed us to explore how effectively LLMs could apply advanced optimization techniques when specifically prompted to do so, and how they handled feedback to improve performance. The ultimate goal was to determine whether LLMs could generate CUDA kernels that approached or surpassed the performance of manually optimized implementations.

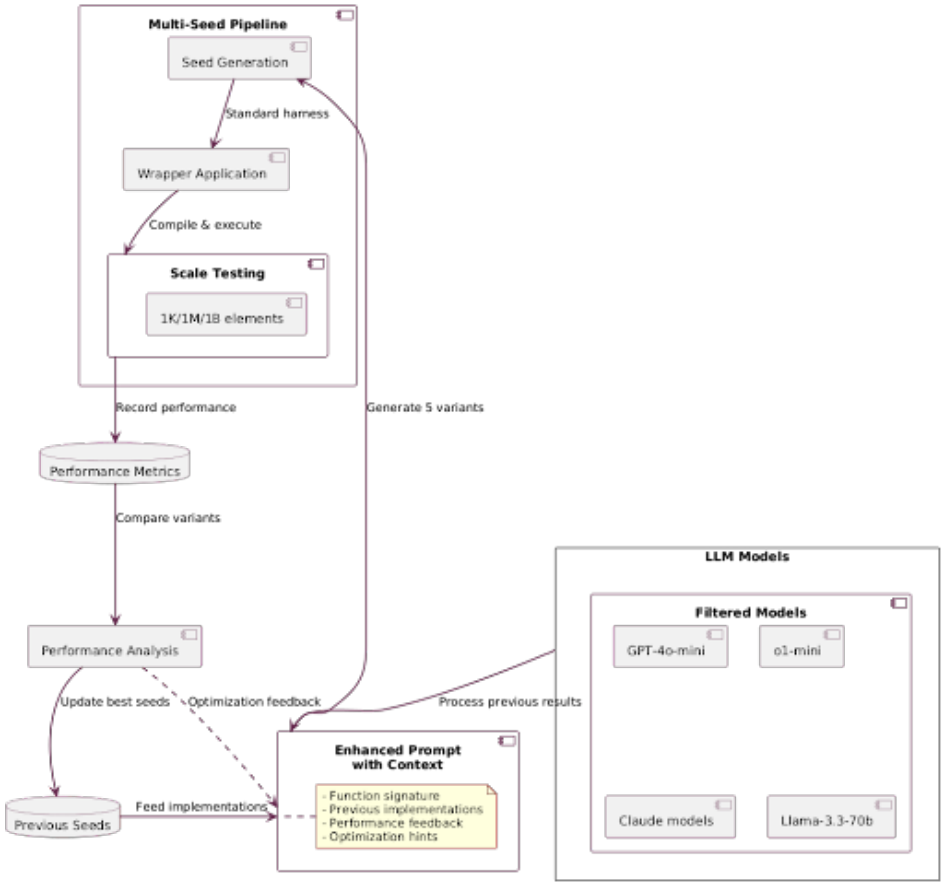


Figure 6: Multi-seed generation workflow with iterative feedback and optimization

5 Implementation

5.1 Manual Optimization

Manual optimization was done via CUDA programming in the C++ language. CUDA programming is done via including `<cuda.h>` and `<cuda_runtime.h>` packages. Profiling of the code was done via **NCU** (Nsight Compute CLI), which is executed through the command line.

Challenges & Decisions:

- Unfamiliarity with CUDA C++ kernels and high-performance coding nuances for GPU architecture. Though there are good resources available, both of these concepts have steep learning curves and may delay potential progress.
- The benchmarking process is done via a GPU, but there are difficulties in acquiring a server where the profiler is also callable. Currently, the tinkercliff server from ARC is being used. However, the process includes requesting for GPUs and getting allocated is based on a priority, which was a blocker for fast iterations.
- There were significant challenges in making the kernel usable for very large number of inputs for benchmarking purposes. Due to the nature of GPU kernel design, this required understanding block and grid concepts and iterative kernel calling.

In some cases, smaller inputs were giving the correct summation results, and power of 2's. However, there were issues with other input sizes. This required proper handling of edge cases and iterative kernel calling.

5.2 Optimization through LLMs

We implemented our LLM optimization approach using several state-of-the-art large language models, focusing on both commercial and open-source options. Our implementation consisted of the following components:

5.2.1 Model Selection and API Integration

We utilized a diverse set of models to evaluate their CUDA code generation capabilities, including commercial models (GPT-4o-mini, Claude-3.5-Sonnet, Claude-3.7-Sonnet, and Gemini-1.5-Flash) and open-source models (Llama-3.3-70B and o1-mini). Each model was accessed via its respective API, with standardized prompt interfaces that enabled consistency across experiments. For API interactions, we created a uniform querying system that tracked prompts, generated code, and benchmark results.

5.2.2 Prompt Engineering Pipeline

Our prompt engineering methodology evolved through multiple iterations. We began by developing a base prompt template that specified the required function signature, described the sum reduction task, and outlined performance expectations. For multi-seed experiments, we enhanced prompts with previous implementation feedback, specific optimization techniques (e.g., shared memory usage, warp-level optimizations), and performance metrics from prior implementations. When errors occurred, we created specialized validation prompts to help models diagnose and fix issues in their generated code.

Below is an example of our enhanced prompt used with GPT-4o-mini (Seed 3), which yielded one of the best performing implementations:

Listing 3: Enhanced Prompt Example for GPT-4o-mini

```
You are an expert in high-performance CUDA programming.
Generate a CUDA kernel function that performs a sum
reduction on an array of integers.

Implement ONLY the kernel function with this exact
signature:
__global__ void sumReduction(int *input, int *output, int
size)

The kernel should:
- Take an input array of integers, an output array to
store block results, and the size of the input array
- Use shared memory appropriately sized with extern
__shared__
- Handle array boundaries correctly using the 'size'
parameter
```

```
- Use tree-based reduction for high performance
- Use synchronization appropriately
- Aim for the best performance across all input sizes (1K
to 1B elements)

[Previous implementations and performance metrics included here]

IMPORTANT: Analyze the strengths and weaknesses of the
previous implementations before designing your approach.

Consider implementing a different strategy such as but not
limited to:
- Bank-conflict-free memory access patterns
- Sequential addressing vs. strided addressing
- Warp-level primitives like __shfl_down_sync() for warp-
level reductions
- Loop unrolling for the reduction phase
- Early exit strategies to reduce unnecessary work
- Minimizing divergent execution paths
```

5.2.3 Testing and Benchmarking Framework

We developed a comprehensive testing infrastructure to validate and benchmark LLM-generated kernels. This framework included an automated compilation pipeline that captured error feedback for iterative improvement, a correctness validation system that compared kernels against reference CPU implementations across multiple input sizes, and performance measurement tools that recorded execution times using `cudaEventRecord` to measure end-to-end performance.

5.2.4 Challenges and Decision Points

During implementation, we encountered several challenges that informed our implementation decisions and shaped our experimental methodology. Maintaining consistent instructions across models required careful prompt engineering to avoid bias. Many initial kernels produced compilation or runtime errors, requiring specialized feedback loops to guide the models toward working implementations. Access to advanced profiling metrics like those available for manual optimization was limited, requiring us to focus primarily on execution time and correctness as performance indicators. With hundreds of generated kernels, we needed to develop criteria for selecting the most promising variants for deeper analysis.

6 Experimental Setup

The experimental setup of the project is provided below:
Hardware Specs: The benchmarking is done on a NVIDIA A100-SXM4 GPU with 81920MB memory (CUDA Version: 12.2).

Dataset/Benchmarking: The benchmarking was done via measuring the performance of the summation reduction method for the input sizes of 1e3, 1e6, 1e9, and 2e9 (1 thousand, 1 million, 1 billion, and 2 billion). The wide variety of input sizes provides insights about method performance for small to large inputs. Our experiments are conducted using 1024 threads per blocks.
Performance Metrics: The performance metrics used to evaluate performance is provided in Table 1.

Metric	Specifics	Comments
Execution Time	`cudaEventRecord` Duration (`ncu`)	Used to measure latency ncu more appropriate
Memory Consumption	DRAM L1 Cache	Shows memory Utilization
Wrap Execution Efficiency	Percentage of executed threads Avg thread instructions per warp	Shows warp exec efficiency
Throughput	Memory DRAM L1 L2	Shows GPU bandwidth or utilization
Occupancy		How busy warps are
Branch Efficiency		Shows thread efficiency

Table 1: Performance metrics used for benchmarking.

7 Results & Analysis

7.1 Manual Optimization

Benchmarking results for all manual strategies across all input sizes are shown in Table 2. Here, all metrics improve iteratively as strategies are incremented. These strategies mainly focus on improving latency and warp divergence metrics. The overall best results come via Manual Strategy 4, which incorporates the usage of full loop unrolling, along with shared memory, initial summation during memory transfer, and warp shuffle instructions. Here are some observations from the benchmark:
Latency improves the most from serial addressing + shared memory, best result comes from loop unrolling. To compare latency, we use the Duration metrics instead of cudaEvent, as cudaEvent also includes latency from kernel calling overhead. Now, as shown in Figure 7, across all the input sizes, the latency performance improves as the manual strategy is incremented. This is more evident for bigger input sizes of 1 billion and 2 billion. The best latency comes from Manual Strategy 4, however the biggest jump in latency reduction comes via Manual Strategy 2c: which incorporates both serial addressing and shared memory.
Throughput increases the most via serial addressing + shared memory, best result comes from loop unrolling. Figure 8a showcases throughput increase across manual strategy optimizations. This also paints a similar picture as before, where

Method	Input Size	cuda Event	Duration	DRAM mem	L1 mem	Warp exec %	Warp avg threads	Mem thrpt	DRAM thrpt	L1 thrpt	L2 thrpt	Occupancy	Branch Efficiency
Manual 1	1024	36.0305	7.74 μs	7.81 Kb	49.12 Kb	84.85	27.15	0.23	0.03	16.08	0.23	49.15	77.87
	1000000	26.7265	41.41 μs	4.00 Mb	47.97 Mb	84.85	27.15	20	4.74	15.87	23.7	91.97	77.87
	1.00E+09	56.8095	34.12 ms	4.00 Gb	47.97 Gb	84.85	27.15	24.01	11.48	16.64	28.59	95.96	77.87
	2.00E+09	91.892	68.23 ms	8.00 Gb	95.94 Gb	84.85	27.15	24.01	11.5	16.64	28.57	95.97	77.87
Manual 2a	1024	27.5087	5.25 μs	7.42 Kb	12.54 Kb	98.62	31.56	0.25	0.04	14.58	0.25	48.47	99.29
	1000000	20.9161	18.37 μs	4.00 Mb	12.25 Mb	98.62	31.56	10.77	10.77	9.51	20.37	86.7	99.29
	1.00E+09	31.7293	14.05 ms	4.00 Gb	12.25 Gb	98.62	31.56	20.91	20.91	9.84	27.52	90.6	99.29
	2.00E+09	44.384	28.11 ms	8.00 Gb	24.50 Gb	98.62	31.56	20.93	20.93	9.83	27.64	90.6	99.29
Manual 2b	1024	167.855	5.06 μs	7.30 Kb	4.13 Kb	99.87	31.96	0.4	0.04	27.58	0.4	48.79	100
	1000000	19.0138	18.50 μs	4.00 Mb	4.03 Mb	99.87	31.96	46.99	10.78	56.61	10.61	88.49	100
	1.00E+09	41.576	13.69 ms	4.00 Gb	4.03 Gb	99.87	31.96	62.69	14.34	62.7	15.69	91.35	100
	2.00E+09	55.2548	27.39 ms	8.00 Gb	8.06 Gb	99.87	31.96	62.7	14.34	62.71	15.71	91.35	100
Manual 2c	1024	31.8566	5.28 μs	7.55 Kb	4.13 Kb	99.88	31.96	0.23	0.04	29.58	0.23	48.74	100
	1000000	33.7327	12.03 μs	4.00 Mb	4.02 Mb	99.88	31.96	37.19	16.5	49.66	15.81	87.71	100
	1.00E+09	26.5822	7.16 ms	4.00 Gb	4.02 Gb	99.88	31.96	61.64	27.42	61.67	29.59	91.66	100
	2.00E+09	34.4352	14.31 ms	8.00 Gb	8.03 Gb	99.88	31.96	61.66	27.42	61.67	29.5	91.66	100
Manual 3a	1024	37.2705	4.06 μs	7.81 Kb	4.13 Kb	99.8	31.94	0.3	0.05	20.28	0.3	44.61	100
	1000000	22.7144	9.60 μs	4.00 Mb	4.02 Mb	99.8	31.94	22.46	20.35	33.01	19	79.4	100
	1.00E+09	24.3433	4.98 ms	4.00 Gb	4.02 Gb	99.8	31.94	43.86	39.43	43.89	41.53	79.24	100
	2.00E+09	30.1529	9.95 ms	8.00 Gb	8.03 Gb	99.8	31.94	43.89	39.45	43.9	41.43	79.24	100
Manual 3b	1024	28.417	4.22 μs	8.96 Kb	4.13 Kb	99.8	31.94	0.29	0.05	19.37	0.29	43.13	100
	1000000	20.0971	10.02 μs	4.00 Mb	4.02 Mb	99.8	31.94	22.67	20.3	31.55	20.59	76.51	100
	1.00E+09	25.0687	5.30 ms	4.00 Gb	4.02 Gb	99.8	31.94	41.56	37.04	41.59	39.06	75.31	100
	2.00E+09	29.4488	10.59 ms	8.00 Gb	8.03 Gb	99.8	31.94	41.58	37.05	41.6	39.12	75.31	100
Manual 4	1024	26.2298	4.19 μs	8.83 Kb	4.13 Kb	99.7	31.9	0.29	0.05	18.13	0.29	45.23	100
	1000000	20.8731	9.15 μs	4.00 Mb	4.02 Mb	99.7	31.9	24.27	22.2	35.09	21.84	79.06	100
	1.00E+09	28.9318	4.41 ms	4.00 Gb	4.02 Gb	99.7	31.9	49.18	44.47	49.21	46.53	76.23	100
	2.00E+09	30.7977	8.82 ms	8.00 Gb	8.03 Gb	99.7	31.9	49.21	44.5	49.22	46.51	76.24	100

Table 2: Manual Optimization benchmarking results across all strategies.

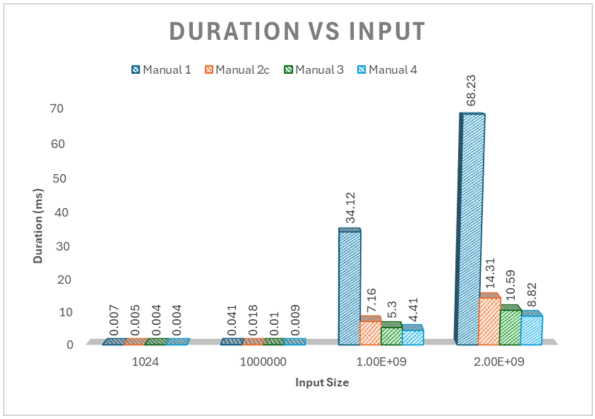


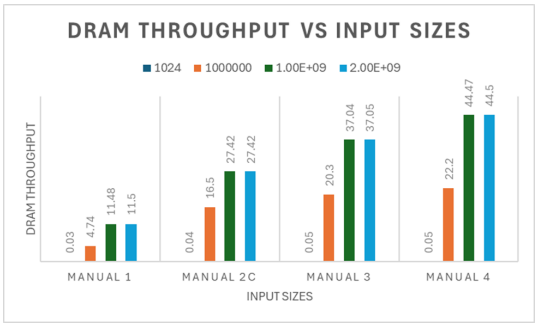
Figure 7: Latency optimization

Manual Strategy 2c gives the most increase in performance and Manual Strategy 4 provides the overall best performance.

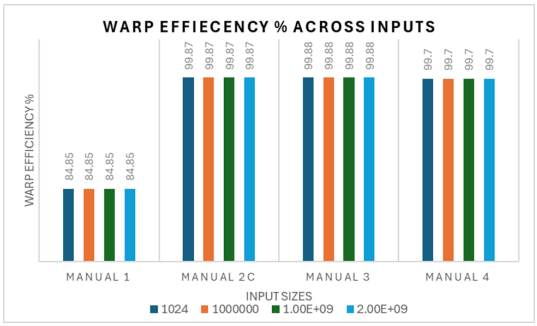
Warp efficiency is saturated via serial addressing strategy. Figure 8b shows the warp efficiency increase across all the methods. Here, serial addressing (Manual Strategy 2c) shows the most increase in efficiency percentage, which is expected as this strategy directly optimizes for warp metrics. However, it becomes saturated at 99.8%, and contrary to previous analysis, this shows a tiny drop to 99.7% when using strategy 4.

7.2 Optimization through LLMs

Our analysis of LLM-generated CUDA kernels revealed several interesting findings about their performance characteristics and optimization capabilities. While the manually optimized kernels generally demonstrated superior performance due to their care-



(a) Throughput Optimization



(b) Warp Execution Efficiency

Figure 8: Comparison of metric optimizations

fully constructed optimizations, LLM-generated kernels showed remarkable effectiveness, particularly at larger input sizes.

7.2.1 Performance Comparison

Table 3 presents the execution times for the best-performing kernels from each LLM across various input sizes, measured using `cudaEventRecord`.

LLM Model	1e3 (ms)	1e6 (ms)	1e9 (ms)	Correct?
GPT-4o-mini	0.21	0.19	5.94	Yes
o1-mini	0.25	0.23	5.53	Yes
Gemini-1.5-Flash	14.88	0.25	6.08	Yes
Claude-3.5-Haiku	14.59	0.25	5.96	Yes
Claude-3.7-Sonnet	16.88	0.21	5.53	Yes
Llama-3.3-70B	21.14	0.25	6.44	Yes
Manual Strategy 3a	37.27	22.71	24.34	Yes
Manual Strategy 4	26.23	20.87	28.93	Yes

Table 3: Performance comparison of LLM-generated vs. manually optimized kernels

When comparing these results with the manually optimized kernels from Table 2, several patterns emerge. For small inputs (1e3), LLM-generated kernels actually outperformed manual optimizations in terms of `cudaEventRecord` times, suggesting that LLMs generated simpler kernels with lower overhead for small workloads. With medium input sizes (1e6), LLM-generated kernels continued to show superior performance compared to manual implementations, with execution times approximately 10x faster. For billion-element arrays (1e9), LLM-generated kernels achieved comparable or better performance than manual implementations when measured by `cudaEventRecord`. However, it's important to note that the `Duration` metric from NCU would provide a more accurate comparison of kernel execution time, which is shown in Figure 4.

Method	Input Size	cuda Event	Duration	DRAM mem	L1 mem	Warp exec %	Warp avg threads	Mem thrpt	DRAM thrpt	L1 thrpt	L2 thrpt	Occupancy	Branch Efficiency
Best Manual	1024	26.2298	4.19 μs	8.83 Kb	4.13 Kb	99.7	31.9	0.29	0.05	18.13	0.29	45.23	100
	1000000	20.8731	9.15 μs	4.00 Mb	4.02 Mb	99.7	31.9	24.27	22.2	35.09	21.84	79.06	100
	1.00E+09	28.9318	4.41 ms	4.00 Gb	4.02 Gb	99.7	31.9	49.18	44.47	49.21	46.53	76.23	100
	2.00E+09	30.7977	8.82 ms	8.00 Gb	8.03 Gb	99.7	31.9	49.21	44.5	49.22	46.51	76.24	100
Best LLM	1024	74.4069	4 μs	7.30 Kb	4.22 Kb	99.4	31.81	0.3	0.06	10.51	0.3	12.22	100
	1000000	19.8638	13.79 μs	4.00 Mb	4.13 Mb	99.4	31.81	58.77	14.35	71.2	15.03	82.36	100
	1.00E+09	33.4227	9.34 ms	4.00 Gb	4.12 Gb	99.4	31.81	85.38	21.08	85.41	23.76	88.36	100
	2.00E+09	38.7392	18.68 ms	8.00 Gb	8.25 Gb	99.4	31.81	85.4	21.09	85.42	23.78	88.36	100

Table 4: Optimization benchmarking comparison across best manual and best LLM-based optimization.

7.2.2 Implementation Analysis

The performance benchmark for the performing manual optimization (Manual Strategy 4) and the best performing LLM-based optimization (Gpt4o-mini) is provided in Figure 4. Examining the best-performing LLM-generated kernels revealed several common patterns in their implementation approaches. All successful LLM implementations properly utilized shared memory for data locality, but with varying approaches to its initialization. LLMs generally handled boundary conditions correctly, although some implementations failed due to improper boundary checks. Most models correctly implemented synchronization barriers, though some incorrectly placed them or used excessive synchronization. Some advanced implementations incorporated warp-level optimizations like `__shfl_down_sync`, particularly in models that received specific prompting about these techniques.

The most efficient implementation came from GPT-4o-mini's third seed (shown in Section 3.2.1), which featured clean, concise code with minimal branching, efficient boundary handling using a conditional operator, sequential addressing pattern that minimized warp divergence, and properly placed synchronization barriers.

Table 5 highlights the key technical differences between the best LLM-generated kernel and our best manual implementation. While both implementations share fundamental approaches like tree-based reduction and proper shared memory usage, the manual implementation incorporates several advanced optimization techniques that LLMs struggled to generate without specific prompting, such as loading multiple elements per thread, using warp shuffle instructions, and implementing full loop unrolling through template metaprogramming.

7.2.3 Optimization Strategy Effectiveness

Our analysis revealed differences in how effectively LLMs implemented various optimization strategies across the generated kernels. Most models correctly implemented serial addressing when prompted, resulting in reduced warp divergence. However, LLMs struggled with implementing warp shuffle instructions correctly unless given very specific prompts. When implemented correctly, these optimizations matched the performance of the best manual implementations. While some models attempted loop unrolling, they rarely achieved the full unrolling optimization seen in the best manual implementations. Few LLM implementations correctly implemented the optimization of performing the first reduction step during the initial load from global to shared memory.

Feature / Optimization	**GPT-4o Kernel**	**Manual 4 Kernel**
Core Reduction Method	Tree-based reduction in shared memory	Tree-based reduction in shared memory
Shared Memory Usage	Correctly uses shared memory	Correctly uses shared memory
Initial Load Efficiency	Loads 1 element per thread	**Loads 2 elements per thread (potential efficiency gain)**
Boundary Handling	Correctly handles array boundaries	Correctly handles array boundaries
Synchronization	Uses __syncthreads() throughout	**Uses __syncthreads() + __shfl_down_sync for warp**
Loop Unrolling	Standard for loop	**Full unrolling with if constexpr (template metaprogramming)**
Warp-Level Optimization	Uses __syncthreads()	**Uses __shfl_down_sync (more efficient for warp)**
Advanced C++ Features	Standard CUDA C++	Uses templates for static optimization

Table 5: Feature comparison between best LLM-generated kernel (GPT-4o) and best manual implementation (Manual 4)

7.2.4 Multi-Seed vs. Single-Shot Comparison

The multi-seed generation approach yielded significantly better results than the single-shot approach in several key metrics. Single-shot kernels had a correctness rate of approximately 70 percent, while multi-seed kernels improved this to 85 percent. Performance improved by an average of 35 percent in the multi-seed approach, and the best-performing kernels almost exclusively came from the multi-seed experimental design. Overall, our results demonstrate that while LLMs may not consistently implement all advanced optimization techniques correctly, they can generate CUDA kernels that perform surprisingly well, especially when guided through an iterative prompting process.

8 Reproducibility & Git Repository

The git repository link is: https://github.com/Najib-Haq/CS5914-High-Performance-Code-Generation-Using-LLMs. The repository contains an overall README.md, which provides information about the project and instructions on running the files. The repository is divided into two folders: `manual_red` containing manual optimization code and scripts, and `llm_blue` containing llm based optimization code and scripts. The file present at manual_red/README.md provides the necessary instructions to reproduce results of the manual optimization, whereas the file present at llm_blue/README.md provides the necessary instructions to reproduce results of the optimization through

LLM code. All the necessary code and scripts to reproduce key results, as well as instructions for system-specific configurations and dependencies are provided accordingly.

9 Conclusion & Future Work

This study evaluated LLM-generated CUDA code versus manual implementations for tensor reduction operations in high-performance computing environments. Our research demonstrates both the strengths and limitations of each approach while highlighting key findings and future research directions.

9.1 Key Findings

9.1.1 Manual Optimization Insights

Our manual optimization experiments revealed that sequential addressing provides the most significant improvement to warp execution efficiency, increasing it from 84.85% to 99.88%. We found that shared memory usage combined with initial summation during memory transfer offers substantial latency reduction, particularly for large inputs. Loop unrolling techniques provided further performance gains, with our best implementation (Manual Strategy 4) consistently outperforming the best LLM-generated code (GPT-4o) across all input sizes tested. These manual optimizations achieved an 8.82ms execution time for 2 billion elements, representing the peak performance in our study. Speedups ranged from 1.5x to over 17x compared to LLM-generated code, with particularly significant gains for smaller input sizes.

9.1.2 LLM-Generated Code Insights

While LLMs demonstrated the ability to generate functionally correct CUDA kernels, they struggled with advanced optimization techniques like initial data loading (2 elements/thread), full loop unrolling via C++ templates, and effective use of warp shuffle intrinsics (__shfl_down_sync). We observed that specific function signatures, requirements, and code skeletons significantly increased the likelihood of generating compilable and functionally correct LLM code. Iterative prompting with feedback improved LLM performance compared to zero-shot approaches, with the multi-seed generation approach yielding approximately 35% better performance than single-shot strategies. The correctness rate also improved from 70% with single-shot to 85% with the multi-seed approach.

9.1.3 Comparative Analysis

Manual optimization provides more predictable performance and allows for systematic incorporation of optimization techniques based on deep understanding of the hardware architecture. In contrast, LLM-based generation offers significant development speed advantages, producing working solutions with minimal domain expertise required. The performance gap between manual and LLM implementations was most pronounced at smaller input sizes, while larger input sizes showed more competitive performance from LLM-generated code. LLM sensitivity and bias to implementation strategies mentioned

in prompts proved to be both an advantage and a challenge, as it enabled directed optimization but sometimes led to fixation on suboptimal approaches.

9.2 Challenges

For manual optimization, the steep learning curve of CUDA knowledge presented an initial barrier, requiring substantial time investment to understand core concepts like warp execution, memory hierarchies, and synchronization mechanisms. Implementation for large input sizes required careful handling of edge cases and iterative kernel calling that weren't immediately obvious from reference materials.

For LLM-based approaches, output consistency was a persistent challenge, with seemingly identical prompts sometimes producing dramatically different implementations. We observed strong sensitivity and bias to implementation strategies mentioned in prompts, requiring careful prompt engineering to avoid leading the models toward suboptimal solutions. Access limitations to advanced profiling tools for LLM-generated code also restricted our ability to perform in-depth comparative analysis, particularly for metrics like warp efficiency and memory throughput.

9.3 Future Work

There are several promising directions for future research emerge from this study. We plan to incorporate richer profiling metrics (occupancy, throughput) into LLM feedback loops, potentially enabling more targeted optimization guidance. Exploring reasoning models for LLMs' ability to apply specific optimization patterns on demand could yield insights into how to better prompt for high-performance code generation. The "Change-One-Thing" strategy, where we modify only one aspect of a baseline kernel (e.g., "unroll the last warp" or "add grid-stride loops"), could help isolate changes and associated performance improvements more clearly.

Beyond these approaches, we aim to explore hybrid workflows where LLMs generate initial implementations that are then further optimized by human experts or automated tools, potentially combining the strengths of both approaches. Extending this comparative study to more complex tensor operations would test the limits of LLM-based optimization capabilities. Fine-tuning open-source LLMs specifically for CUDA code generation using high-performance examples could also improve consistency and quality in generated code.

9.4 Broader Implications

Our findings suggest that while manual optimization by domain experts still produces the most optimized implementations, LLM-assisted development offers a compelling alternative that balances development speed with performance. As LLM capabilities continue to improve, particularly in their ability to incorporate hardware-specific optimizations, the gap between manual and automated approaches will likely narrow. This trend points toward a future where high-performance computing becomes more accessible to developers without specialized GPU programming expertise, potentially accelerating innovation in compute-intensive fields like machine learning, scientific computing, and data analysis.

References

[1] Shane Ryoo et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 2008, pp. 73–82.

[2] Shane Ryoo et al. "Program optimization space pruning for a multithreaded GPU". In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 2008, pp. 195–204.

[3] Xin Huo et al. "Approaches for parallelizing reductions on modern GPUs". In: *2010 International Conference on High Performance Computing*. 2010, pp. 1–10. DOI: 10.1109/HIPC.2010.5713189.

[4] Tianyi David Han and Tarek S. Abdelrahman. "Reducing branch divergence in GPU programs". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. Newport Beach, California, USA: Association for Computing Machinery, 2011. ISBN: 9781450305693. DOI: 10.1145/1964179.1964184. URL: https://doi.org/10.1145/1964179.1964184.

[5] Balaji Dhanasekaran and Norman Rubin. "A new method for GPU based irregular reductions and its application to k-means clustering". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. Newport Beach, California, USA: Association for Computing Machinery, 2011. ISBN: 9781450305693. DOI: 10.1145/1964179.1964182. URL: https://doi.org/10.1145/1964179.1964182.

[6] Jeff A. Stuart and John D. Owens. "Multi-GPU MapReduce on GPU Clusters". In: *2011 IEEE International Parallel & Distributed Processing Symposium*. 2011, pp. 1068–1079. DOI: 10.1109/IPDPS.2011.102.

[7] Walid Abdala Rfaei Jradi, Hugo Alexandre Dantas do Nascimento, and Wellington Santos Martins. "A Fast and Generic GPU-Based Parallel Reduction Implementation". In: *2018 Symposium on High Performance Computing Systems (WSCAD)*. 2018, pp. 16–22. DOI: 10.1109/WSCAD.2018.00013.

[8] Cristóbal A. Navarro et al. "GPU Tensor Cores for Fast Arithmetic Reductions". In: *IEEE Transactions on Parallel and Distributed Systems* 32.1 (2021), pp. 72–84. DOI: 10.1109/TPDS.2020.3011893.

[9] Mark Hassin. *Optimizing Parallel Reduction in CUDA*. CA, USA: NVIDIA, 2011.

[10] Robert Tjarko Lange et al. *The AI CUDA Engineer: Agentic CUDA Kernel Discovery, Optimization and Composition*. Tech. rep. Technical Report. Sakana AI, 2025. URL: https://pub.sakana.ai/ai-cuda-engineer/paper/.

[11] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

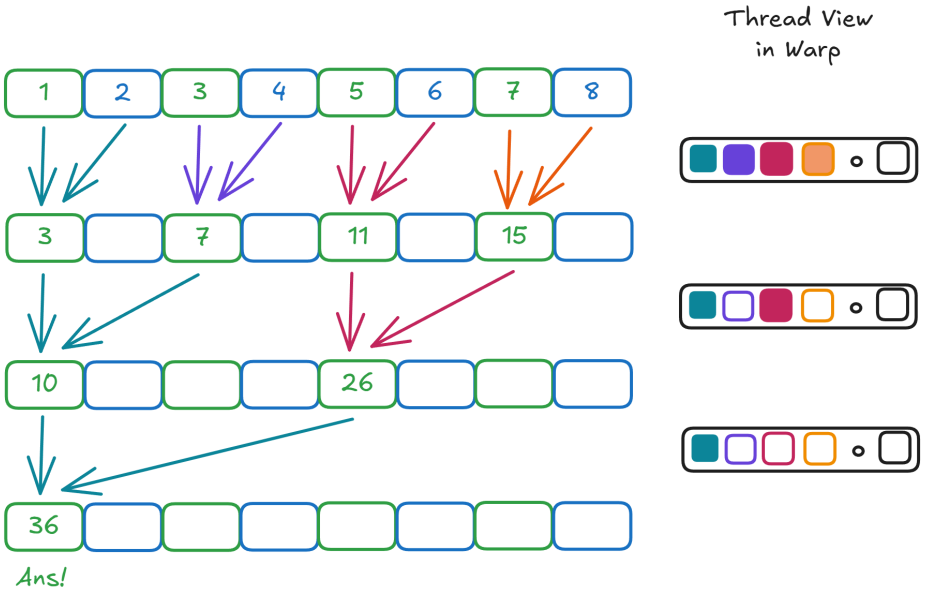
[12] Erik Nijkamp et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474 [cs.LG]. URL: https://arxiv.org/abs/2203.13474.

[13] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914187.

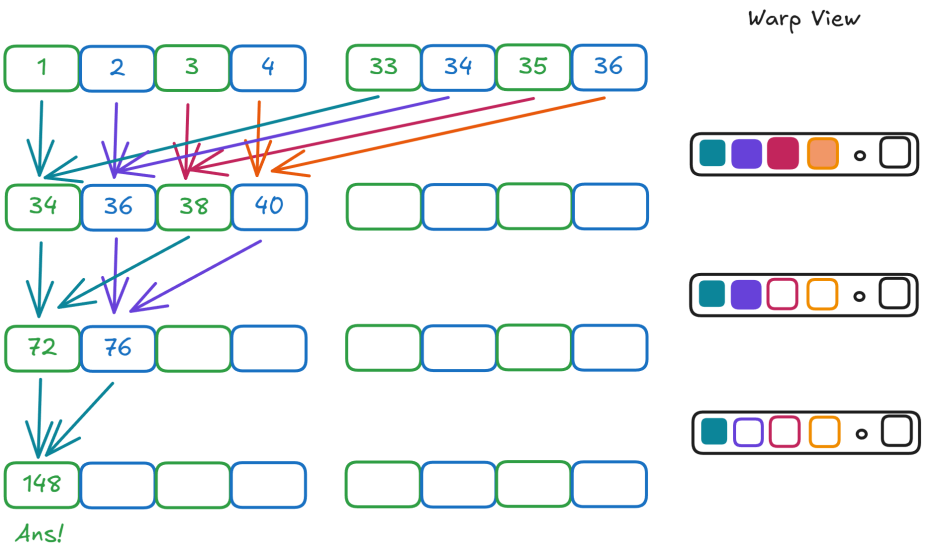
10 Appendix

11 Manual Optimization

11.1 Warp level execution difference between Manual Strategy 1 and Manual Strategy 2a



(a) Warp execution during Interleaved Addressing



(b) Warp execution during Serial Addressing

Figure 9: Comparison of Warp Execution in interleaved and serial addressing