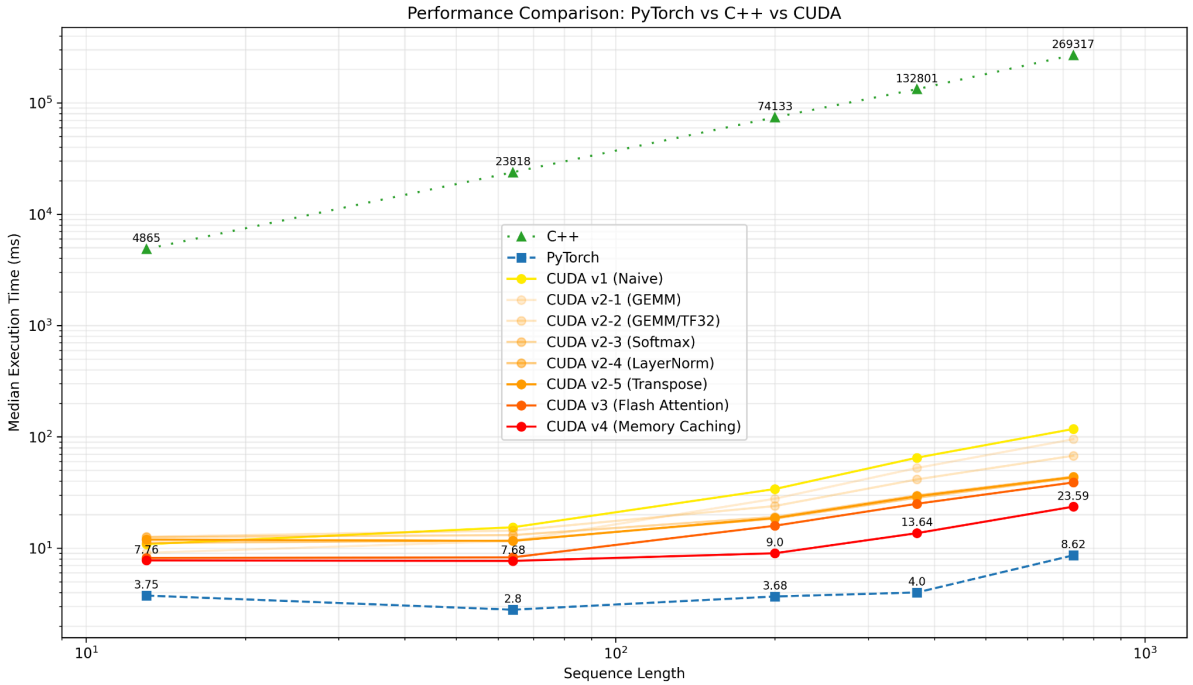
Computing II (Heterogeneous Parallel Computing)
Seoul National University

Project Report

Exploring Parallelization Techniques in Transformer Architecture

2024-27323
Hyoung Jo, Bhang



1. Introduction

Although the transformer architectures have become the foundation of modern large language models, their development and deployment are still constrained by significant computational demands. Fortunately, with the remarkable advancements in both hardware and software design of parallelism—particularly graphics processing units (GPUs)—the processing speed of these workloads has been greatly accelerated. However, the underlying parallelization techniques are often abstracted away by high-level libraries, leaving them obscure to many researchers and developers. Therefore, this project aims to look deeper into the high-performance computing principles that have made today's massive model architectures computationally feasible.

The objective is to implement a GPT-2 style (decoder-only) transformer from scratch[1]. Starting with a naive C++ CPU-based implementation of the model, the project bridges the performance gap by migrating the model's entire inference workload from the CPU to the GPU. During the optimization process, all acceleration is achieved through raw CUDA C++ kernels, strictly avoiding off-the-shelf NVIDIA CUDA libraries such as cuBLAS or cuDNN. This approach has been a valuable opportunity to understand and deconstruct the parallelization strategies in modern AI architectures, providing hands-on experience with the fundamental concepts of high-performance computing. Let's dive in!

2. Related Works

To build an efficient inference engine, existing literature addresses performance bottlenecks through foundational kernel optimization, memory-efficient attention mechanisms, and automated compilation. To begin with, the parallelization of the layer normalization and the softmax relies on optimized reduction patterns such as sequential addressing, algorithm cascading, and loop unrolling, introduced in CUDA webinar[2]. Building on this, Milakov and Gimelshein (2018) published an online normalization calculation for softmax, which computes normalization factors in a single pass without storing intermediate statistics, significantly reducing global memory traffic.[3]

Moreover, the self-attention mechanism has been the primary bottleneck for long sequences due to quadratic complexity. Dao et al. (2022) solved this with FlashAttention[4], which runs computation via tiling to keep data in the fast shared memory (SRAM), minimizing the high-latency global memory access. FlashAttention-2 (Dao, 2023) further refined this approach by reordering outer and inner loops to improve parallelism and occupancy[5]. Finally, PyTorch 2 (Ansel et al., 2024) utilizes dynamic graph compilation to fuse operations and optimize memory layouts[6]. This project evaluates performance against this highly efficient automated baseline.

3. Project Design

This project is structured around three distinct implementations to establish performance bounds and demonstrate the efficacy of low-level optimizations.

(a) PyTorch (Upper Bound)

The PyTorch implementation serves as both the performance upper bound and the validation standard. To ensure a competitive baseline, the model runs with TensorFloat-32 (TF32) precision enabled and utilizes torch.compile to leverage kernel fusion and optimization. Additionally, this implementation is used to export model weights and generate validation datasets (inputs and expected outputs) to verify the correctness of the C++ and CUDA versions. All performance metrics are captured using torch.utils.benchmark with blocked_autorange to guarantee statistical significance.

---

[1] Radford, A, et al. (2019), Language Models are Unsupervised Multitask Learners, OpenAI.
[2] Harris, M. (2007). Optimizing Parallel Reduction in CUDA. NVIDIA Webinar. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[3] Milakov, M., & Gimelshein, N. (2018). Online normalizer calculation for softmax. arXiv preprint arXiv:1805.02867.
[4] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv preprint arXiv:2205.14135.
[5] Dao, T. (2023). FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv preprint arXiv:2307.08691.
[6] Ansel, J., et al. (2024). PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. GitHub. https://github.com/pytorch/pytorch

(b) C++ (Lower Bound, Baseline)

The C++ implementation provides the lower bound baseline. It is a naive, single-threaded CPU execution of the GPT-2 architecture, strictly adhering to the official OpenAI implementation details (such as the tanh approximation for the GELU activation function). This codebase serves as the architecture template, which is subsequently ported as-is to the CUDA implementation.

(c) CUDA

The core of the project involves four distinct versions of the CUDA implementation, evolving from a direct port to a highly optimized inference engine. Due to the space limit, we only introduce high-level explanations in the report. The code itself includes highly detailed comments.

Version 1 (Naive)

This version represents a direct translation of the C++ logic to CUDA. The transformer architecture is decomposed into fundamental kernels: embed, transpose, matmul, add, add_bias, gelu, layer_norm, and softmax. Each kernel is implemented naively, relying on standard global memory accesses without architectural optimization.

Version 2 (Kernel Optimizations)

This version introduces specific optimizations across five incremental stages (v2-1 to v2-5).

- Matrix Multiplication (GEMM): The naive matrix multiplication is first upgraded to use shared memory tiling (v2-1). It is then further accelerated using Tensor Cores with TF32 precision (v2-2). To match the Tensor Core hardware instruction shape (16 x 16 x 8), shared memory tiles are adjusted to 32 x 16 and 16 x 32, with blocks launched across four warps.
- Reductions (Softmax & LayerNorm): Both operations (v2-3, v2-4) are optimized using the online softmax algorithm combined with warp-level parallel reduction. To minimize synchronization overhead (__syncthreads), each warp first reduces to a single value stored in shared memory, which is then aggregated by the thread 0 of the block.
- Coalesced Memory Access: The transpose kernel (v2-5) is optimized to ensure coalesced global memory access via shared memory tiling.

Version 3 (Flash Attention)

Based on the FlashAttention-2 algorithm, this version addresses the memory bandwidth bottleneck of the attention mechanism. By loading tiles of query, key, and value matrices into on-chip shared memory (SRAM), the kernel fuses the attention score calculation with the softmax normalization, drastically reducing redundant global memory reads and writes.

The kernel is architected such that each thread block processes a specific query block and an attention head. Within the block, warps are assigned to process individual rows of the query, and it keeps the query in the registers while streaming keys and values through shared memory.

Algorithm 1 FLASHATTENTION-2 forward pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, block sizes B_c, B_r.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into T_r blocks $L_i, \ldots, L_{T_r}$ of size B_r each.
for $1 \le i \le T_r$ **do**
  Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.
  On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}, \ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}, m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.
  for $1 \le j \le T_c$ **do**
    Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \mathrm{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \mathrm{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.
    On chip, compute $\mathbf{O}_i^{(j)} = \mathrm{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})^{-1} \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.
  end for
  On chip, compute $\mathbf{O}_i = \mathrm{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$.
  On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.
  Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.
  Write L_i to HBM as the i-th block of L.
end for
Return the output $\mathbf{O}$ and the logsumexp L.

Version 4 (Memory Caching)

To mitigate the latency overhead of dynamic memory management, this version implements a static memory pool strategy similar to PyTorch's caching allocator. Instead of repeatedly calling cudaMalloc and cudaFree for temporary matrices during every layer of inference, the system allocates a single set of buffers at the start of the forward pass and reuses them across all transformer layers.

4. Experiments & Results

(a) Dataset

The dataset consists of texts with average sequence length of 296 about heterogeneous parallel computing. The shortest and longest sequences have lengths of 13 and 732, respectively. More details can be found in the appendix and the repository.

(b) System Configurations

The specifications for the device that the kernel was executed on are given below:

NVIDIA RTX A6000 (GPU)

The CUDA / Driver Version is 12.2 / 535.247.01. The memory is 48GB in its size, with a bandwidth of 768GB/s, type of GDDR6, and bus of 384 bit. The number of streaming processors (SMs) are 84 with 10,752 CUDA Cores and 336 Tensor Cores. The theoretical peak performance is 38.71 TFLOPS for FP32.

(c) Evaluation Settings

Both C++ and CUDA code are compiled with an optimization flag of -O3 for maximal performance. The architecture flag -arch=sm_86 (A6000) is added. All experiments are conducted in a single GPU and run with Slurm. The scripts are configured as main.sh and timer.sh, where the former validates the results and the latter runs benchmarking.

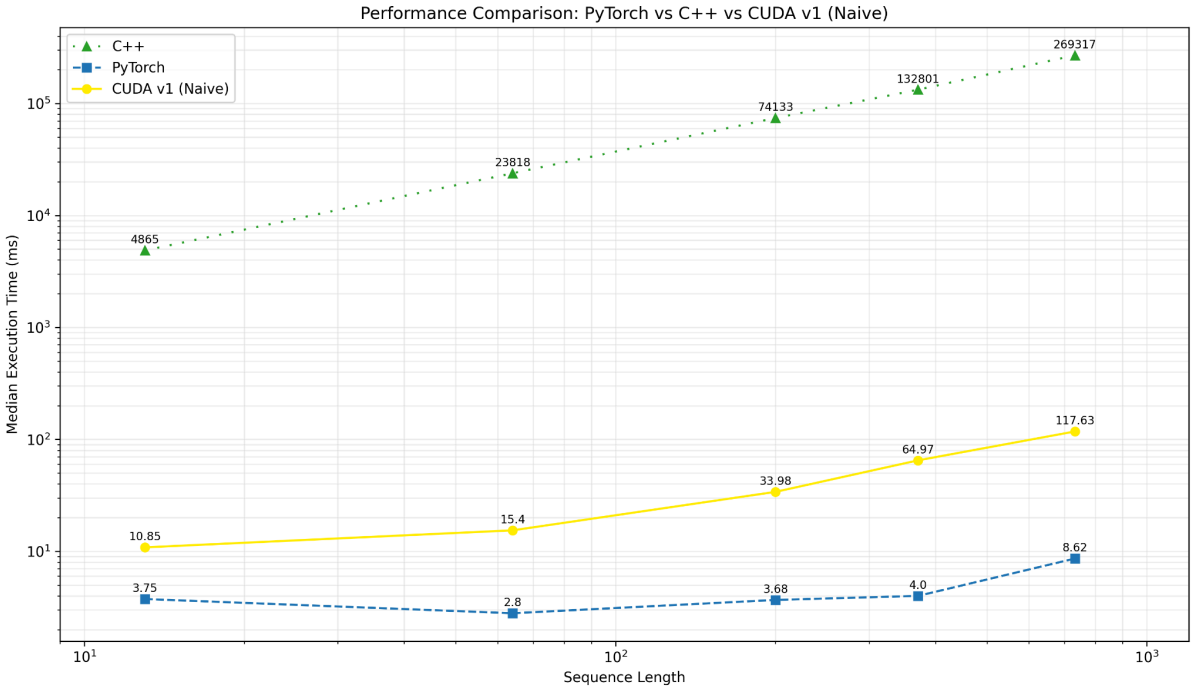
For the benchmarks, warm ups of 5 and iterations of 20 are repeated to obtain stable results. This design follows the PyTorch benchmark, and all results below are plotted with the median execution times. Furthermore, to warm up the GPU before the execution of timer.sh, three runs of main.sh are issued manually with sbatch. i.e., `sbatch ./cuda/main.sh` x 3, then `sbatch ./cuda/timer.sh`.

(d) Experimental Results (Performance Profiling)

The final version concluded with 99% performance gain over the C++ baseline, 2.7 times slower than the PyTorch upper bound. The sections below elaborate the accelerations for each version.

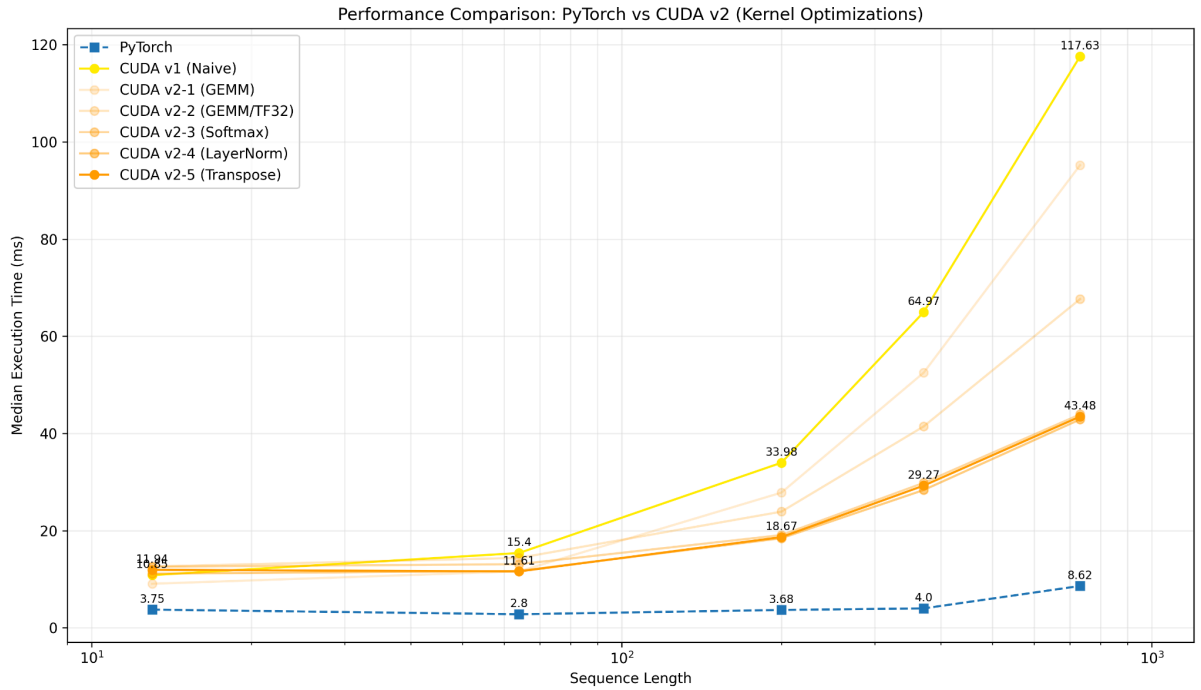
Version 1 (Naive)

The figure compares the performance of PyTorch (Upper Bound), C++ (Lower Bound, Baseline), and the CUDA v1 (Naive). It could be observed that a naive port from C++ to CUDA still yields extremely high parallelism, with speed up ranging from 450 to 2300 times for different sequence lengths.



Version 2 (Kernel Optimizations)

The plot shows each mini-versions of kernel optimizations from v2-1 to v2-5. In overall, the longest sequence exhibited 2.7 times faster execution time. For a more detailed view on the boost of each kernel, the measurements conducted with Nsight systems are also shared below.

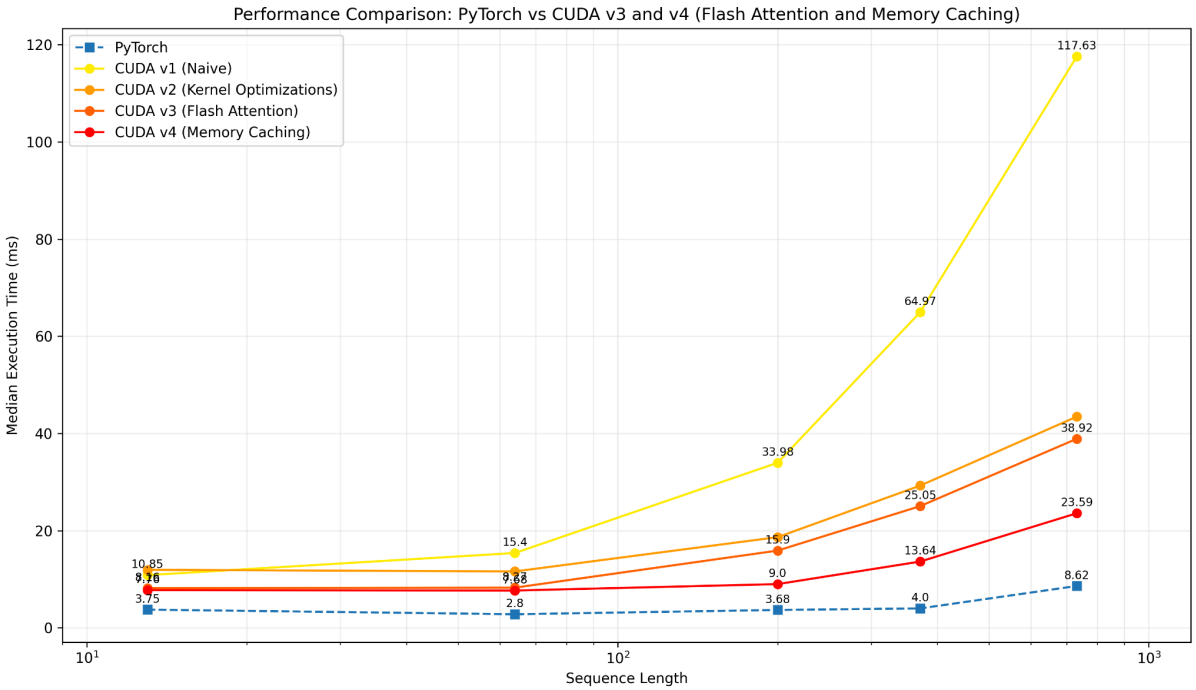


	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
v1	11,320,945,018	126,375	89,582.2	14,912.0	1,823	1,656,716	226,602.3	operations::naive_matrix_multiplication_kernel(float *,
v2-1	7,491,476,330	126,375	59,279.7	8,000.0	1,407	1,131,273	152,563.3	operations::matrix_multiplication_kernel(float *, const
v2-2	4,562,147,249	126,375	36,100.1	8,224.0	2,367	358,018	70,079.1	operations::tensor_core_matrix_multiplication_kernel(flo
v2-2	3,301,120,335	54,000	61,131.9	60,544.0	3,200	173,601	37,460.9	operations::naive_softmax_kernel(float *, int, int)
v2-3	156,821,123	54,000	2,904.1	2,816.0	1,983	7,584	924.1	operations::softmax_kernel(float *, int, int)
v2-3	568,581,653	9,375	60,648.7	60,190.0	55,038	76,926	3,207.7	operations::naive_layer_norm_kernel(float *, const float
v2-4	40,481,832	9,375	4,318.1	4,256.0	3,231	7,520	784.8	operations::layer_norm_kernel(float *, const float *, co
v2-4	119,153,148	54,001	2,206.5	2,240.0	1,183	1,294,807	5,580.5	operations::naive_transpose_kernel(float *, const float
v2-5	72,699,007	54,001	1,346.3	1,344.0	1,247	575,773	2,472.2	operations::transpose_kernel(float *, const float *, int

v2-1 (GEMM) showed a performance increase of approximately 44% and v2-2 (GEMM/TF32) further improved it by another 39%. v2-3 (Softmax) and v2-4 (LayerNorm) kernels were also dramatically accelerated by 95% and 93%. Lastly, v2-5 transpose kernel gained 39% in performance.

Version 3 (Flash Attention), Version 4 (Memory Caching)

The plot shows the profiled time for Flash Attention and Memory Caching optimizations. The Flash Attention turned up to be approximately 32% ~ 10.5% faster, and Memory Caching yielded 5% ~ 39% speedup.



5. References

The architectural design and optimization strategies were primarily derived from course lecture materials. For the Python and C++ implementations, the PyTorch documentation[7] and its associated GitHub repository[8] served as essential references. For the low-level CUDA implementation, the following additional resources beyond the lecture slides were used as references: (1) the attention kernel design was adapted from the methodologies presented in the official implementation of FlashAttention[9]; and (2) the development was further guided by official NVIDIA documentation, specifically CUDA C++ Best Practices Guide[10] and technical blogs (Programming Tensor Cores in CUDA 9[11], Using Tensor Cores in CUDA Fortran[12]). The hardware constraints and specifications were noted from the NVIDIA RTX A6000 datasheet[13]. Finally, Gemini was used to assist in designing the project structure and refining the draft of this report.

6. Appendix

The source code of the project is hosted as a public repository on GitHub[14]. The code is organized to effectively compare the performance of Transformer architectures across Python, C++, and CUDA implementations. Each language version consists of a self-contained directory containing the full

[7] PyTorch. PyTorch documentation. https://docs.pytorch.org/docs/stable/index.html
[8] PyTorch. (2023). PyTorch. GitHub. https://github.com/pytorch/pytorch
[9] Tri Dao. (2023). Dao-AILab/FlashAttention. GitHub. https://github.com/Dao-AILab/flash-attention
[10] NVIDIA Corporation. (2025). CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide
[11] Appleyard, J., & Yokota, S. (2017). Programming Tensor Cores in CUDA 9. NVIDIA Technical Blog. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9
[12] Ruetsch, G. (2017). Using Tensor Cores in CUDA Fortran. NVIDIA Technical Blog. https://developer.nvidia.com/blog/using-tensor-cores-in-cuda-fortran
[13] NVIDIA Corporation. NVIDIA RTX A6000 datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web%20(1).pdf
[14]https://github.com/hyoungjo/transformer-cuda

source code, headers, and dedicated Makefiles and scripts for compilation and execution. Additionally, the **source files include all naive and intermediate versions** implemented as separate functions/kernels in the code (the launch commands are commented out).

The repository holds a shared data/ folder to store model weights (and structure), input data (input_ids) and expected outputs (probs) for validation, and actual texts used in the experiment. It also contains a logs/ directory that archives the output from main.sh and timer.sh execution scripts via sbatch. Detailed instructions for setting up the Miniconda environment and execution commands for benchmarks are provided in the README.md.

The structure of the entire project is shown below for reference.

```
transformer-cuda
├── cpp
│   ├── include
│   │   ├── gpt2.hpp
│   │   ├── operations.hpp
│   │   ├── tensor.hpp
│   │   └── utils.hpp
│   ├── main.sh
│   ├── Makefile
│   ├── source
│   │   ├── gpt2.cpp
│   │   ├── main.cpp
│   │   ├── operations.cpp
│   │   ├── tensor.cpp
│   │   ├── timer.cpp
│   │   └── utils.cpp
│   └── timer.sh
├── cuda
│   ├── include
│   │   ├── gpt2.hpp
│   │   ├── operations.hpp
│   │   ├── tensor.hpp
│   │   └── utils.hpp
│   ├── main.sh
│   ├── Makefile
│   ├── source
│   │   ├── gpt2.cu
│   │   ├── main.cu
│   │   ├── operations.cu
│   │   ├── tensor.cu
│   │   ├── timer.cu
│   │   └── utils.cu
│   └── timer.sh
├── data
│   ├── 0
│   │   ├── input_ids.bin
│   │   └── probs.bin
│   │   ...
```

```
│   ├── 14
│   │   ├── input_ids.bin
│   │   └── probs.bin
│   ├── modules.txt
│   ├── texts.txt
│   └── weights.bin
├── logs
├── Makefile
├── python
│   ├── helper.py
│   ├── main.py
│   ├── main.sh
│   ├── timer.py
│   └── timer.sh
└── README.md
```