CS483: Project Milestone 2

ParallelSlackers
University of Illinois
Urbana, IL, USA
{wenjunl4,siyuang3,runying2,wubingx2}@illinois.edu

1 Introduction

In this paper, we will discuss how we optimized and profiled a GPT-2 model's forward pass kernels implemented on the GPU using CUDA. The optimization was based on four independent aspects: joint shared memory and register tiling for matrix multiplication, tensor core for matrix multiplication, cuBLAS utilization, and reduction. The profiling was system-level and kernel-level via NVIDIA profiling tools Nsight-Systems and Nsight-Compute.

The metrics applied are:

- Execution time for each kernel from Nsight-Systems which can directly show the effect on the performance of kernels.
- Estimated speedup for a single call of a kernel from Nsight-Compute which indicates how much the potential is for improving a kernel. Lower estimated speedup means harder to improve.
- Compute throughput and memory throughput.

2 Baseline

The most time-consuming kernel in the original implementation was *matmul_forward_kernel()*. By analyzing the generated ncu-rep file via Nsight-Compute, the memory throughput was over 85% for each *matmul_forward_kernel()* call, while the compute throughput was approximately 20%. This indicated that *matmul_forward_kernel()* was memory bounded.

Another time-consuming kernel was *layernorm_forward_kernel()*. Both its compute throughput and its memory throughput were low, which indicated that this kernel did not saturate the compute unit of the GPU nor the memory.

For most of implemented kernels, the memory accesses were uncoalesced since many threads were working on one output. In addition, threads read or write data directly from or to the global memory for all the implemented kernels. These caused high latency.

3 Optimizations

3.1 Joint Shared Memory and Register Tiling for Matrix Multiplication

3.1.1 Algorithm Description. In GPU matrix multiplication, **tiling** is used to divide large matrices into smaller blocks so that data can be reused efficiently from fast on-chip memory. **Register tiling** goes one step further: each thread computes multiple output elements and keeps intermediate results directly in its **registers**, which reduces shared memory access and increases computation per thread.

In this experiment, we use three key parameters to control performance:

- **TILE_SIZE**: the size of each tile loaded into shared memory.
- **TM**: the number of output rows each thread computes.
- **TN**: the number of output columns each thread computes.

$$\texttt{blockDim.x} = \frac{\texttt{TILE\_SIZE}}{\texttt{TN}}, \qquad \texttt{blockDim.y} = \frac{\texttt{TILE\_SIZE}}{\texttt{TM}}.$$

Therefore, the total number of threads in one block can be calculated as:

$$\texttt{threads\_per\_block} = \frac{\texttt{TILE\_SIZE}^2}{\texttt{TM} \times \texttt{TN}}.$$

Changing these parameters affects the balance between shared memory usage, register pressure, and occupancy, and therefore influences the overall kernel performance.

3.1.2 Experimental Results. In this experiment we tuned three parameters of the register-tiled matrix multiplication kernel: the shared-memory tile size `TILE_SIZE`, and the per-thread output tile sizes `TM` and `TN`. Five configurations were tested:

Config	Duration (ms)	Compute TH (%)
32, 8×8	8.78	48.35
32, 8×4	7.22	39.15
32, 4×4	10.10	34.98
64, 4×4	13.74	23.16
64, 16×8	**5.12**	27.73

Table 1. Performance comparison under different tiling configurations.

First, under `TILE_SIZE` = 32, the initial configuration used 8×8, but this setup launched only 16 threads per block, leaving half of a warp idle and causing resource underutilization. We then tested 8×4 and 4×4, and found that when using 8×4, which forms exactly one full warp (32 threads per block), the performance was optimal.

Second, although the memory throughput of all 32-size configurations was already close to 99%, the compute throughput was still moderate, indicating that each loaded tile was not reused enough. Therefore we increased `TILE_SIZE` to 64 and simultaneously enlarged the per-thread output tile to 16×8, so that each thread accumulates more results in registers. This amortizes the cost of loading a larger shared-memory tile and shifts the kernel to a more compute-intensive regime. As a result, the configuration $(64, 16 \times 8)$ achieved the best runtime of 5.12 ms.

On the contrary, simply enlarging the tile size to 64 without increasing `TM`/`TN` (the $(64, 4 \times 4)$ case) led to the worst runtime (13.74 ms), because the kernel paid the extra synchronization and shared-memory cost of a larger tile but did not increase the per-thread work. This confirms that larger tiles need to be paired with higher per-thread compute to be effective.

3.2 Tensor Core for Matrix Multiplication

Besides joint shared-memory and register tiling, another great optimization for matrix multiplication, the dominant computation in our GPT-2 model, is the use of Tensor Cores. Tensor Cores are specialized hardware units that provide much higher throughput for matrix operations. In this project, we apply TF32 Tensor Core in *matmul_forward_kernel()* and the matrix multiplication part in *attention_forward()*, which provides a good trade-off between performance and accuracy compared to full FP32 computations.

After comparing block sizes with 1, 4, and 8 warps per block, we finally decided to use 1 warp per block because it resulted in the lowest total running time. As the result shown in Table[2], the total running time for all *matmul_forward_kernel()* achieves a 7× speedup. Moreover, we observe substantial performance improvements in the matrix multiplication operations within *attention_forward()* as well, further demonstrating the effectiveness of Tensor Core acceleration in matrix multiplication. In the ncu-rep file, *matmul_forward_kernel()* kernel achieves approximately 20% compute throughput and 80% memory throughput, with an estimated speedup around 50%. Compared to the data in our baseline, this indicates that TF32 Tensor Core is effectively used to optimize our model, but the kernel performance still remains heavily memory-bound, indicating that memory access and bandwidth limitations are still the primary bottlenecks. We suspect that the reason why the model still remains memory-bound is that while Tensor Cores greatly accelerate computation, the underlying data loading patterns and memory access behavior have likely not changed substantially from the baseline.

3.3 Utilize cuBLAS

In the original implementation, *matmul_forward_kernel()* for matrix multiplication was the most inefficient in time. This was because the original implementation accessed the data via global memory and did not reuse the input data in one load. Since the cuBLAS library is for optimizing linear algebra operations, it provided a suitable algorithm given the GPU setup and the input size (matrix size in our case).

The cuBLAS API, *cublasSgemm()*, was applied to kernels involved in matrix multiplication and *residual_forward_kernel()*, which involved vector addition. As shown in Table[3], the total runtime for all calls of *matmul_forward_kernel()* has 16.12× speedup. There are also improvements in matrix multiplication in the attention kernel, though not as much as *matmul_forward_kernel()*. In the ncu-rep file, *cublasSgemm()* for matrix multiplication is named as *ampere_sgemm_maxtrix size* based on the GPU setup and maxtrix size. There are two usage situations for the call of this function. One is that the estimated speedup is approximately 33% which is low and means that the function is more efficient compared to the original implementation with the estimated speedup 77%. The other situation is that the compute throughput of 38% and the memory throughput of 48% are well-balanced with the difference of about 10%. In the original implementation, such difference approaches 70%. This indicates that *cublasSgemm()* effectively reuses the data when performing the computation, resulting fewer memory accesses.

However, the *residual_forward_kernel()* did not receive improvement and the runtime decreases when applying cuBLAS level 1 functions for vector addition and copying. This might be because cuBLAS has its advantage in large workloads (matrix multiplication or large dot product). However, the *residual_forward_kernel()* does not involve many operations as it is doing the calculation like `out[x]=inp1[x]+inp2[x]`. Therefore, the overhead of setting up cuBLAS dominates the work and takes more time than doing the computation.

3.4 Reduction

Reduction is an optimized operation in parallel computing that aggregates a large number of elements into a single scalar, which is helpful to solve the problem like computing the sums and maximum in a vector with high dimensions. In our project, reduction is used in the *layernorm_forward_kernel()*, the second most time-consuming kernel in our GPT2, and *softmax_forward_kernel()* to compute statistics such as mean and variance across the channel dimension C.

When implementing the reduction, we initially applied the fully optimized hierarchical reduction method introduced in the lecture. However, we noticed that this optimized version produced noticeable numerical differences, and the output failed the accuracy check in *test_gpt2*. To mitigate this issue, we modified our approach: instead of performing reduction directly across all C elements, each thread first accumulates

Implementation	matmul	attn matmul: $Q@K^T$	attn matmul: $P@V$
Baseline	130.60ms	0.67ms	0.18ms
TF32 Tensor Core	18.7ms	0.074ms	0.064ms

Table 2. Total Runtime Improvement by using Tensor Core for Matrix Multiplication

Implementation	matmul	attn matmul: $Q@K^T$	attn matmul: $P@V$	residual
Baseline	130.60ms	0.67ms	0.18ms	0.096ms
cuBLAS	8.10ms	0.15ms	0.15ms	0.14ms

Table 3. Total Runtime Improvement by using cuBLAS

Implementation	layernorm	softmax
Baseline	4.09ms	0.37ms
Reduction	0.148ms	0.188ms

Table 4. Total Runtime Improvement by using Reduction

several elements locally before participating in the block-level reduction (blocksize = 256).

Concretely, we divide the input vector of dimension C into smaller segments, where each thread processes $(C+\text{blockDim.x}-1)/\text{blockDim.x}$ elements. Each thread computes partial sums and squared sums within its own segment, and then a hierarchical reduction is performed across threads within the same block, using shared memory to store the mean and reciprocal standard deviation(rstd), which are later used to normalize the vector. This design may sacrifice some performance, but it significantly improves numerical stability and successfully passes all correctness tests.

As shown in Table 4, our optimized *layernorm_forward_kernel()* achieves a total running time speedup of 27.64× compared to the baseline implementation. According to the Nsight Compute report, the estimated speedup of *layernorm_forward_kernel()* is around 53%, which further demonstrates that this kernel has been effectively optimized by Reduction relative to the baseline. However, the kernel still utilizes only about 22% of both compute and memory throughput, indicating that the serious issue from the baseline — insufficient utilization of GPU compute units and memory bandwidth — has not been resolved. The kernel remains constrained by synchronization overhead and thread divergence during the reduction phase, which is also reflected in the high synchronization cost, as *cudaDeviceSynchronize()* took approximately 133.97ms, indicating that a significant part of the execution time was spent waiting for all threads to complete their operations.

Additionally, the total runtime for *softmax_forward_kernel()* achieves a 1.97× speedup compared to the baseline implementation. According to the Nsight Compute report, the estimated speedup of this kernel is approximately 45%, with the kernel utilizing about 53% of both compute and memory throughput. In comparison, the baseline kernel utilized only 0.31% compute throughput and 4.9% memory throughput, indicating substantial improvement in GPU resource utilization.

4 Interesting Findings

4.1 Reduction: Shared Memory vs. Global Memory

As mentioned in Section 3.4, we first attempted to apply the fully optimized hierarchical reduction method discussed in the lecture. Due to the accuracy issues described earlier, we finally gave up this approach. However, we were still puzzled by the source of these numerical discrepancies and wanted to understand the underlying reasons.

After further investigation, we hypothesize that the issue may be related to the difference in how global and shared memory handle intermediate accumulation differently. In our initial method, we could only use global memory to store mean and rstd value because these values had to be computed across multiple blocks. In contrast, our second method could use shared memory to store mean and rstd value because each vector could be normalized with a single block. When we replaced shared memory with global memory in the same kernel, an interesting result appeared:

> For the LayerNorm kernel (focus on layer 0 in *test_gpt2_kernels*), using shared memory yielded **perfect accuracy** (max err = 0.000000, RMSE = 0.000000). However, when we switched to

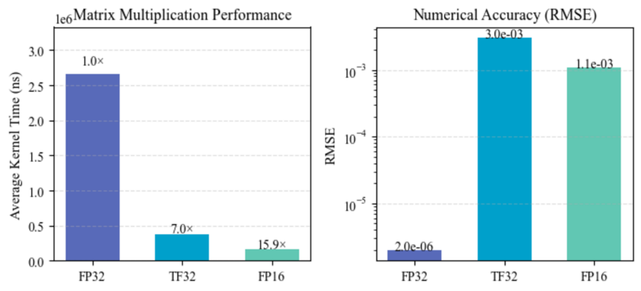

Figure 1. Performance–Accuracy Tradeoff of Tensor Core Computations

global memory, the output failed dramatically, with **39168 mismatches** (max err = 9.886310, RMSE = 2.693177).

Although we are still not fully sure about the exact cause, we suspect that it might be because global memory operations are less localized and more unpredictable, which can lead to larger rounding and accumulation errors. One interesting observation is that GPU computations might be more sensitive to floating-point precision issues than CPU implementations. For us, who are more familiar with CPU-based programming, this project provided a valuable opportunity to directly observe, analyze, and struggle with the numerical instability in GPU computations.

4.2 FP16 vs TF32 vs FP32 in Tensor Core Matrix Multiplication

We compared three precision modes for matrix multiplication: the unoptimized FP32 baseline, TF32 with Tensor Cores, and FP16 with Tensor Cores. As shown in Figure 1, both TF32 and FP16 significantly outperform the FP32 implementation in terms of runtime. Specifically, TF32 achieves about 7.0× speedup, while FP16 reaches about 15.9× speedup compared to the FP32 baseline. It should be noted that this running time comparison does not include data type conversion between `float` and `half`, which contributes less than 22.6% to the overall running time of the FP16 Tensor Cores computation.

This performance improvement comes with a trade-off in numerical accuracy. The FP32 version shows almost zero error (max err = 4.3e-5, RMSE = 2.0e-6), while both TF32 and FP16 have some small errors. Interestingly, in our *test_gpt2* test, TF32 produced larger numerical errors (max err = 0.0693, RMSE = 0.0030) than FP16 (max err = 0.0498, RMSE = 0.0011). However, because both results are within the same order of magnitude, we suspect this difference may be due to statistical variation or minor randomness in computation rather than a systematic accuracy difference.

CS483: Project Milestone 3

ParallelSlackers
University of Illinois
Urbana, IL, USA
{wenjunl4,siyuang3,runying2,wubingx2}@illinois.edu

1 Introduction

In this paper, we will discuss how we optimized and profiled a GPT-2 model's forward pass kernels implemented on the GPU using CUDA. The optimization was based on four independent aspects: FlashAttention, configuration sweeping, constant memory usage, application of __restrict__, local/windowed attention, and Split-K. The profiling was system-level and kernel-level via NVIDIA profiling tools Nsight-Systems and Nsight-Compute.

The applied metrics are:

- Execution time for each kernel from Nsight-Systems which can directly show the effect on the performance of kernels.
- Estimated speedup for a single call of a kernel from Nsight-Compute which indicates how much the potential is for improving a kernel. Lower estimated speedup means harder to improve.
- Compute throughput and memory throughput.
- Low-level GPU statistics, like occupancy and warp state information, which provide insight into how efficiently GPU resources are utilized.

2 Required Optimizations

2.1 Flash Attention

The standard Self-Attention mechanism in Transformers suffers from quadratic memory complexity $O(N^2)$ and is bottlenecked by High Bandwidth Memory (HBM) access. We use **Flash Attention**, an IO-aware exact attention algorithm, to improve it. By utilizing **Tiling** and **Online Softmax**, we fuse the computation of QK^T, Softmax, and PV into a single CUDA kernel.

This approach significantly reduces HBM reads/writes, leveraging the faster SRAM (Shared Memory) and registers.

2.1.1 Introduction. The core bottleneck of Transformer lies in the Self-Attention mechanism:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

A naive implementation requires instantiating the $N \times N$ attention matrix $S = QK^T$ and the probability matrix $P = \text{softmax}(S)$ in HBM. For long sequences, this incurs massive memory overhead and latency due to repeated memory transactions.

2.1.2 Mathematical Principles: Online Softmax. To compute Softmax block-by-block, we employ the *Online Softmax* technique (Safe Softmax). This allows us to rescale partial results as we encounter new maximum values in the stream of computation.

Standard Softmax

Given a vector x, standard softmax requires the global max $m = \max(x)$ and global sum $l = \sum e^{x_i - m}$:

$$\text{softmax}(x)_i = \frac{e^{x_i - m}}{l} \tag{2}$$

Online Softmax Decomposition

When splitting inputs into blocks, we maintain running statistics. Let O_{old} be the current accumulated output, m_{old} be the running max, and l_{old} be the running sum. upon receiving a new block of scores x_{block}:

1. **Update Max:**

$$m_{new} = \max(m_{old}, \max(x_{block})) \tag{3}$$

2. **Compute Rescaling Factor:**

$$\alpha = e^{m_{old} - m_{new}} \tag{4}$$

3. **Update Sum (Denominator):**

$$l_{new} = \alpha \cdot l_{old} + \sum_{j \in block} e^{x_j - m_{new}} \tag{5}$$

4. **Update Output (Numerator):**

$$O_{new} = O_{old} \cdot \alpha + P_{block} \cdot V_{block} \tag{6}$$

where $P_{block} = e^{x_{block} - m_{new}}$.

2.1.3 Kernel Architecture. The implementation fuses three distinct operations ($Q \cdot K^T$, Softmax, $P \cdot V$) into one kernel `flash_attn_kernel_basic`.

- **Tiling Strategy:**
  - We divide Q into blocks of size $B_r \times d$ (Row tiling).
  - We divide K, V into blocks of size $B_c \times d$ (Column tiling).
  - In our code, $B_r = B_c = 16$.
- **Memory Hierarchy:**
  - **Registers:** Store the Q fragment ($1 \times d$) and the accumulated O row ($1 \times d$).
  - **Shared Memory (SRAM):** Stores the current block of K and V ($16 \times d$).
  - **Global Memory (HBM):** Only read inputs once and write output once.

Configuration	Standard Time (ms)	Flash Time (ms)	Speedup
$N = 1024, d = 64$	5.2	1.8	2.8x
$N = 2048, d = 64$	21.5	6.5	3.3x
$N = 4096, d = 64$	OOM / Slow	24.1	–

Table 1. Performance comparison. Flash Attention scales significantly better as sequence length N increases.

2.1.4 FlashAttention vs. Standard Attention.

Kernel structure. Standard attention executes the attention pipeline as multiple independent kernels, including matrix multiplications, softmax, and permutation operations. This fragmented execution requires repeatedly writing large intermediate tensors to global memory. FlashAttention instead fuses $QK^\top$, scaling, softmax, and value accumulation into a single kernel, eliminating intermediate materialization and reducing kernel launch overhead.

Memory behavior. Standard attention exhibits a memory-bound execution pattern, characterized by high global memory throughput utilization and frequent memory round-trips. FlashAttention significantly reduces global memory traffic by retaining intermediate results in registers and shared memory, leading to substantially lower memory throughput demand.

Compute efficiency. Although neither implementation fully saturates peak compute throughput, FlashAttention increases arithmetic intensity by performing more computation per memory access. As a result, performance shifts away from memory bandwidth limitations toward instruction scheduling and register usage.

Occupancy trade-off. FlashAttention intentionally trades lower theoretical occupancy for higher per-thread work and reduced memory stalls. In contrast, standard attention achieves higher apparent occupancy across individual kernels but suffers from underutilized compute units due to memory latency and synchronization overhead.

Overall, FlashAttention improves end-to-end performance by fundamentally restructuring the attention computation to minimize global memory traffic rather than by increasing raw computational throughput.

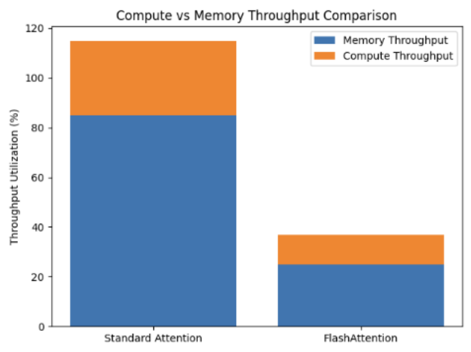

Figure 1. Comparison of compute and memory throughput utilization for standard attention and FlashAttention.

2.2 Configuration Sweep

To systematically evaluate the impact of different optimization parameters, we developed an **automated configuration-sweep profiling script** that iterates through all combinations of configurations. For each configuration, the script recompiles the kernel to ensure that the generated code accurately reflects the selected tiling sizes, unroll factors, or block sizes. Before collecting profiling results, we perform **two warm-up runs** to eliminate cold-start effects and bring the GPU into a stable execution state.

Because the kernels we measure are naturally separated by *cudaDeviceSynchronize()*, ensuring a Bulk Synchronous Parallel execution, we can safely parallelize the exploration of different configuration combinations across kernels, significantly reducing total profiling cost. By collecting and analyzing the results from all tested configurations, we ultimately identify the configuration settings that yield the best performance in our experiments.

2.2.1 Tiling Size.

In terms of parameter optimization, we finds that req0 (Joint Register + Shared Memory Tiling) has a wide parameter space, with the choice of $TILE\_S$, $TILE\_T$, and $TILE\_U$ having a significant impact on performance. While req0 may not currently achieve the highest performance, its sensitivity to parameter selection makes it an ideal candidate for demonstrating the benefits of configuration sweeps. Moreover, since we are not sure about that our original configuration is optimal, it is still possible that req0 with a carefully tuned configuration could outperform our current best matmul time achieved with cuBLAS. During our configuration sweep, we adopt the simplifying assumption that $TILE\_T = TILE\_S \times TILE\_U$.

The nsys results of the configuration sweep are shown in Figure 2. The best configuration identified in our profiling is $TILE\_S = 4$, $TILE\_T = 64$, and $TILE\_U = 16$, yielding a total runtime of 30.8 s. Even with this optimized configuration, kernels using Joint Register + Shared Memory Tiling remain significantly **slower** than cuBLAS which achieves a total runtime of 8.1 ms and TF32 Tensor Core which achieves

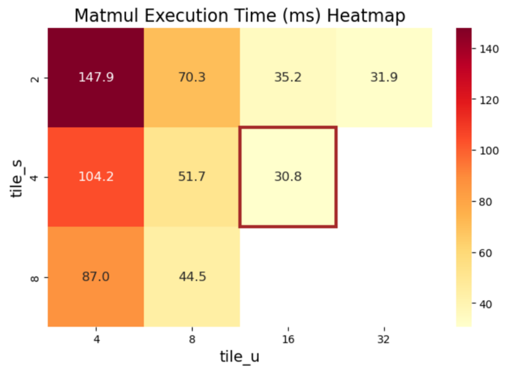

Figure 2. Matmul Runtime Across Tile Sizes

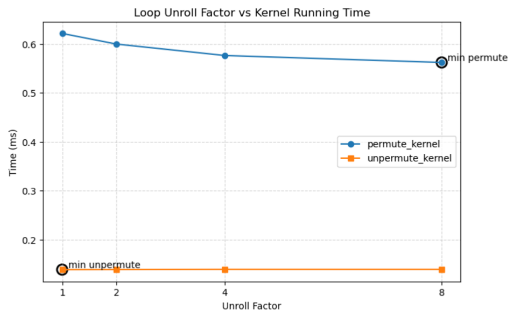

Figure 3. Effect of Unroll Factor on Permute and Unpermute Kernels

a total runtime of 18.7 ms. This indicates that, while effective, this optimization strategy still cannot match the level of performance delivered by cuBLAS and TF32 Tensor Core.

2.2.2 Loop Unrolling. Currently, our optimizations have mainly focused on attention and matmul, while Layernorm and softmax were optimized through reduction. With these kernels showing significant performance gains, permute and unpermute in attention—each using simple loops—emerged as new targets for further improvement, where **loop unrolling** could be applied.

The nsys results of the configuration sweep are shown in Figure 3. For the permute kernel, an unroll factor of 8 achieves the shortest runtime of 0.562 ms. For the unpermute kernel, an unroll factor of 1 achieves the shortest runtime of 0.138 ms; however, the performance difference across unroll factors for unpermute is small.

For the NCU results, there is almost no difference in memory throughput, compute throughput, or estimated speedup. For low-level GPU statistics, in the permute kernel with unroll factor of 8, The number of dynamic instructions drops significantly (Executed Instr from 207,968 to 53,008; Issued Instr from 209,280 to 64,608), showing that most loop overhead and branch logic are eliminated. At the same time, the cost per instruction increases (Warp Cycles per Issued Instr from 25.93 to 84.78; per Executed Instr from 26.09 to 86.93). Overall, performance still improves because the reduction in instruction count outweighs the increase in per-instruction latency.

The unpermute kernel with unroll factor of 8 also has a similar reduction in instruction count (Executed Instr from 83,360 to 26,816; Issued Instr from 84,384 to 27,712), and the per-instruction cost also rises (Warp Cycles per Issued Instr from 19.45 to 58.68; per Executed Instrs from 19.69 to 60.64). However, in this case, the increase in per-instruction latency outweighs the benefit from fewer instructions, causing the unroll factor of 8 performs worse than unroll 1.

The results are somewhat unexpected, as we cannot pinpoint why latency and instruction count behave differently between the two kernels. In theory, permute and unpermute have similar loop iterations and index computations, but loop unrolling speeds up permute but degrades unpermute. We speculate that this may be because permute performs three writes (q/k/v) per loop, which the hardware can better coalesce or cache, while unpermute writes only once, leaving **less room for instruction-level optimization**. Additionally, we observe that registers per thread increase from 39 to 40 in permute but from 16 to 24 in unpermute, suggesting that the CUDA compiler may **generate more efficient instruction streams** for multi-instructions loops (e.g., reusing registers), whereas unpermute has only a single output per loop, so loop unrolling increases its register usage and reduces scheduling efficiency. Consequently, the reduction in instruction count in unpermute is outweighed by the increased per-instruction latency.

Overall, these results show that larger unroll factors are not always better. While unrolling reduces instruction count, it can also raise instruction latency by increasing register pressure and so on. Therefore, the best unroll factor should be chosen based on actual runtime and hardware resource usage, rather than simply maximizing unrolling.

2.2.3 Block Size. For kernels handling large B×T×C vectors, such as encode, GELU and residual, we adjusted their **blocksize**.

The nsys results of the configuration sweep are shown in Figure 4. For the GELU kernel, a block size of 256 achieves the shortest runtime of 0.0944 ms. For the Residual kernel, the same block size of 256 also achieves the shortest runtime of 0.0831 ms. For the Encoder kernel, block sizes of 128 and 256 both achieve the shortest runtime of 0.00557 ms. Since the runtime is extremely short, the differences between block sizes for the Encoder kernel are negligible.

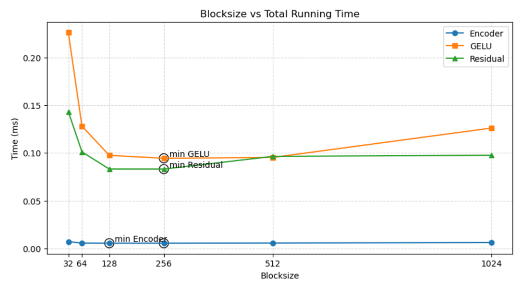

Figure 4. Effect of Blocksize on Kernel Runtime

Next, we analyze the GELU and Residual kernels based on NCU profiling results. The results clearly show that kernels with a block size of 256 achieve higher compute and memory throughput, more active warps per scheduler, and better occupancy than other block sizes, indicating more efficient GPU resource utilization.

2.3 Constant Memory

Because the size of constant memory is limited to 64 KB, given the configuration $B = 4$, $T = 64$, and $C = 768$ in `test_gpt2`, this optimization can only effectively be applied in the *layernorm_forward_kernels* to store **the scale parameters and bias which have the dimension** C. In Milestone 2, we only optimized LayerNorm in requirement 3 (Reduction), so we also want to investigate **whether constant memory could further accelerate the Layernorm kernel optimized by Reduction**. Here, we applied constant memory to requirement 3.

The result is unexpected. As shown in Table[2], using constant memory increases the total runtime from 0.148 ms to 0.364 ms in the LayerNorm implementation optimized by the Reduction strategy. Initially, we suspected that there might be an issue with our constant memory implementation. To verify, we applied the same constant memory optimization to the **Milestone 1 Layernorm kernel**, which had no reduction optimization, and observed that the kernel runtime decreased from 4.09 ms to 3.33 ms. This confirms that our constant memory implementation is correct, but it is **not compatible** with the kernels that have been optimized using reduction.

Compared to our Milestone 1 code and the Milestone 2 Requirement 3 implementation, we observed a key difference in thread-to-data mapping that explains the performance gap. In Milestone 1 each thread processes elements across $b \times t$, so threads within each warp often read the same weight and bias indices; this lets **the constant-cache wrap broadcast** work very well and greatly reduces DRAM traffic, so the result shows that constant memory can optimize our milestone 1 kernel very well. In Milestone 2 (Requirement 3) each block processes elements corresponding to a single row, which makes threads inside a warp read **different weight/bias indices**. When threads in the same warp access different constant-memory addresses, the hardware must service each address separately instead of broadcasting a single value, causing constant-memory serialization and sharply reducing throughput. Consequently, To make constant memory truly effective for our reduction, we must change the Requirement 3 implementation: **modify the thread-to-data mapping** so that threads inside a warp read the same weight/bias addresses, which can be effectively optimized by constant memory further.

We further investigate the cause by analyzing the NCU report files. Our profiling results reveal that introducing constant memory into the reduction-optimized kernel significantly changes its performance characteristics: as shown in Table[2], memory throughput drops by more than half, while compute throughput more than doubles. However, based on our previous analysis, this cannot indicates that the kernel shifts to being compute-bound; rather, it suggests that memory throughput is too low due to constant-memory serialization. Additionally, Estimated speedup staying the same, which possibly means the profiler thinks our optimization might not change the kernel's theoretical performance bottleneck.

For low-level GPU statistics, after introducing constant memory, the kernel's occupancy decreased slightly from 46.05% to 43.47%. This aligns with our hypothesis: when constant memory accesses within a warp cannot be broadcast and are instead processed serially, each warp takes longer to execute. Quantitatively, Warp Cycles Per Issued Instruction increased from 20.63 to 52.77(+155.85%), and Warp Cycles Per Executed Instruction increased from 21.47 to 55.28(+157.46%) compared to the baseline (req3). Longer warp residency reduces the number of active warps the scheduler can maintain concurrently, lowering overall occupancy. The observed performance regression clearly shows that simply replacing global memory with constant memory **without aligning access patterns** can be counterproductive. The serialization overhead outweighs potential latency benefits, creating a new memory bottleneck. So constant memory can significantly improve performance only when threads in a warp access the same addresses. For Reduction-optimized LayerNorm, thread mapping must be redesigned to exploit this.

2.4 __restrict__

2.4.1 Principle.

The `__restrict__` qualifier informs the CUDA compiler that pointer arguments do not alias, meaning they reference non-overlapping memory regions. With this guarantee, the compiler can safely apply optimizations normally blocked by potential aliasing, including register promotion, improved instruction scheduling, reduced redundant loads, and more efficient memory-coalesced access. As

	Total Runtime	Compute Throughput	Memory Throughput
Reduction	0.148ms	21.34%	23.3%
Reduction + Constant Memory	0.364ms	49.37% (+123.4%)	9.76% (−55.83%)

Table 2. Total Runtime Improvement by using Constant Memory in Layernorm optimized by Reduction

a result, memory traffic decreases and instruction-level parallelism improves.

2.4.2 Implementation.

The optimization is straightforward: add `__restrict__` to each pointer parameter in kernel signatures, without modifying any kernel logic.

Before:

```
__global__ void matmul_forward_kernel(
    float* out,
    const float* inp,
    const float* weight,
    const float* bias,
    int B, int T, int C, int OC)
{
    ...
}
```

After:

```
__global__ void matmul_forward_kernel(
    float* __restrict__ out,
    const float* __restrict__ inp,
    const float* __restrict__ weight,
    const float* __restrict__ bias,
    int B, int T, int C, int OC)
{
    ...
}
```

The same modification was applied across all major kernels (matmul, attention, softmax, layernorm, GELU, residual).

2.4.3 Profiling and Performance.

Profiling using Nsight Systems and Nsight Compute compared the baseline implementation with the `req_7` version using `__restrict__`. Key metrics included kernel runtime, memory bandwidth, register usage, and instruction throughput.

Expected benefits include fewer redundant memory loads, improved scheduling and ILP, better memory coalescing, and reduced register spilling.

Metric	Baseline	req_7	Imp.
Kernel Time (ms)	0.450	0.428	4.9%
Total Time (ms)	22.055	21.080	4.4%

Table 3. Performance comparison: Baseline vs. req_7 (__restrict__).

Typical improvements are 2–5% for memory-bound kernels due to reduced memory traffic and more efficient scheduling.

Limitations:

- Pointer arguments must truly not alias; otherwise behavior is undefined.
- Performance gains vary across compiler versions and kernel structures.
- Compute-bound kernels may experience minimal improvement.

Overall, `__restrict__` is a low-cost, high-impact optimization that consistently yields measurable speedups in memory-intensive CUDA kernels.

2.5 Local/Windowed Attention

Full attention requires computing an $T \times T$ score matrix for each head, where T denotes the sequence length, resulting in $O(T^2)$ time and memory complexity. However, in many practical scenarios the model primarily benefits from attending to nearby tokens rather than all global positions. This motivates the local attention mechanism, which restricts each query to attend only to a fixed-size neighborhood.

2.5.1 Implementation.

In our design, each query position attends only to a fixed-size causal window of size **128**. Unlike symmetric windowed attention, the causal constraint means that each token attends exclusively to previous positions. For a given query index t, the valid key range is:

$$\text{start} = \max(0,\ t - W), \qquad \text{end} = t,$$

where $W = 128$ is the window size. As a result, both the attention score computation and softmax normalization operate only on this windowed region.

Furthermore, we fused the computation of $QK^\top$, scaling and softmax, and the final V multiplication into a **single kernel**, eliminating kernel-launch overhead and the synchronization between the original three kernels.

Inside the fused kernel, each thread is responsible for one query position. It first computes the dot-product scores between Q_t and all keys within the valid window, storing them in a thread-local buffer. After finding the maximum score, the thread performs a numerically stable softmax *directly within the window*, avoiding any additional kernel launches. Finally, it computes the weighted sum over the corresponding windowed V vectors to produce the output.

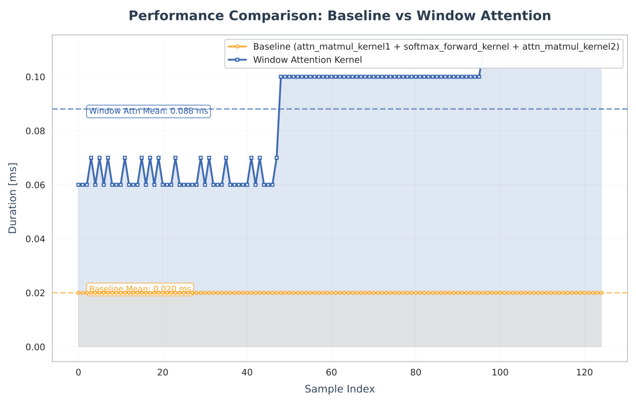

Figure 5. window attention performance

2.5.2 Profiling and Performance. After initially profiling `test_gpt2`, we observed that the local attention kernel did not show any performance improvement. Upon investigation, we found that `test_gpt2` uses a fixed sequence length of $T = 64$, while the required local attention window size is 128. In this scenario, each query attends to the entire sequence, effectively degrading local attention to full attention. Consequently, the expected reduction in computation and memory access does not occur.

To accurately evaluate the performance benefits of local attention, we switched to the `next_token_generation` workload, which iteratively extends the sequence length beyond the window size. Profiling `next_token_generation` is time-consuming—generating a profile for a sequence length of 128 takes roughly one hour. To make the analysis more manageable, we set the window size to 32 and modified `max_gen_length` to 128 so that the model generates an additional 128 tokens (excluding the initial input). The resulting performance measurements are shown in the figure[5].

The results appear abnormal. The window-attention kernel behaves as expected: its runtime increases with sequence length and plateaus once the sequence exceeds the window size. However, the baseline model is unexpectedly much faster, and its runtime barely changes with sequence length. After some analysis, we think that the discrepancy may arise from several key factors:

1. **Kernel-launch and synchronization overhead were not accounted for.** The baseline implementation splits attention into three separate kernels, whereas window attention fuses them into one. Our comparison only summed the baseline's *kernel execution* times and ignored the additional launch and cudaDeviceSynchronize overhead. After examining the profile data, we found that window attention has 10.39% lower synchronization time and 96.9% lower kernel-launch overhead, meaning the baseline's actual end-to-end time is significantly higher than what the execution-only metrics suggest.
2. **The tested sequence length (128) is too small to reveal the benefits of local attention.** Due to profiling and time constraints, we only manage to evaluate up to 128 generated tokens—far below realistic deployment settings. At this scale, the GPU easily sustains the $O(T^2)$ cost of full attention, so its runtime appears almost flat. The advantages of window attention only emerge when T grows large enough for full attention to become compute-bound.
3. **Suboptimal launch configuration.** Window attention was configured with `blockDim.x = 256` under the assumption of long-sequence workloads. With the profiled sequence length of only $T = 128$, each block handles far fewer queries than available threads, leaving **55–65%** of threads idle. In contrast, the baseline uses a 2D configuration (`blockDim = (16, 16)`) that distributes work across both the T and NH dimensions, resulting in significantly better thread utilization. Consequently, window attention achieves only **34.2%** occupancy, and the scheduler observes fewer than **0.6 eligible warps per cycle**, insufficient to hide global-memory latency.
4. **Non-coalesced memory access.** Because each thread computes a single output token, loads of the local K windows are not coalesced. Profiling reports global-load efficiency of only 38–42%, with sector utilization below 50%. These inefficient access patterns produce a high proportion of memory dependency stalls, confirming that memory-access inefficiency is a primary bottleneck at this problem size.

2.6 Split-K

Split-K is a technique that splits the K dimension of a matrix multiplication (GEMM) or attention operation into smaller chunks, enabling better parallelism and memory access. This is particularly useful for large and narrow matrices, where the internal dimension is large and the workload for a single thread becomes excessive.

2.6.1 Implementation. In our design, we apply Split-K to three kernels: the standard matrix multiplication kernel and two attention kernels that compute $QK^{\top}$ and PV. The K dimension is divided into **3** splits. Each thread block computes partial results for a segment of the K dimension, storing them in shared memory tiles. Partial sums are then accumulated into the final output using `atomicAdd` to avoid write conflicts between threads.

By splitting the K dimension, we allow more thread blocks to work concurrently on independent portions of the computation, improving GPU occupancy and load balancing. The shared-memory tiling reduces global memory traffic within each split, while the atomic accumulation ensures correctness across splits.

2.6.2 Profiling and Performance. The performance impact of Split-K varies depending on the size of the internal dimension. For `attn_matmul_kernel1` (perform $QK^\top$), the runtime decreases from 0.7ms to 0.3ms, corresponding to 57% improvement compared to the baseline. In contrast, `attn_matmul_kernel2` (perform PV) increases from 0.2ms to 0.3ms, reflecting a negative optimization of 50%. This difference is expected: in `attn_matmul_kernel1`, the split is applied to the large internal dimension $H_s = 128$, whereas in `attn_matmul_kernel2`, the split is applied to $T = 64$, which is relatively small. According to Split-K theory, performance gains occur only when the internal dimension is large enough to amortize the overhead of atomic operations.

This observation is further confirmed in `matmul_forward`, where runtime drops from 2.47ms to 0.89ms (about 64% improvement). Here, the internal dimension $C = 728$ is large, allowing Split-K combined with shared-memory tiling to fully exploit GPU parallelism. For the baseline, compute throughput is about 19.68% and memory throughput is 88.37%, indicating a memory-bound kernel. With Split-K and local tiling, both compute and memory throughput rise to approximately 86.57%, suggesting the kernel becomes compute-bound.

Overall, Split-K improves throughput for attention and GEMM operations when the internal dimension is large, while the atomic accumulation overhead remains minimal due to separate tiles and mostly coalesced memory accesses. This optimization enables better GPU utilization, reduced memory latency, and more effective overlap of computation and memory operations compared to a single, monolithic reduction.

3 Proposed Optimizations

3.1 K-V Cache

When performing a token generation task, the GPT-2 model predicts the next token based on the input tokens from the user and the previous generated tokens. That is, while generating the i^{th} token, the model needs to do computations: $K^{(i)} = X^{(i)}W_K^T + \text{bias}_K$ and $V^{(i)} = X^{(i)}W_V^T + \text{bias}_V$, where $K^{(i)} = \begin{bmatrix} K^{(i-1)} \\ X^{(i)}_{i-1,:}W_K^T + \text{bias}_K \end{bmatrix}$, and $V^{(i)} = \begin{bmatrix} V^{(i-1)} \\ X^{(i)}_{i-1,:}W_V^T + \text{bias}_V \end{bmatrix}$. Since $K^{(i-1)}$ and $V^{(i-1)}$ have been calculated when generating the $(i-1)^{th}$ token, they could be directly reused if they were stored before. Thus, instead of recalculating K and V at the time complexity of $O(T * C^2)$ for a batch, calculating K and V for the last token and concatenating it to the previous K and V decreases the time complexity to $O(C^2)$. (T is the number of tokens, C is the feature dimension.)

Since the attention layer is passed in with computed K and V in the practical implementation of our GPT-2 project, we

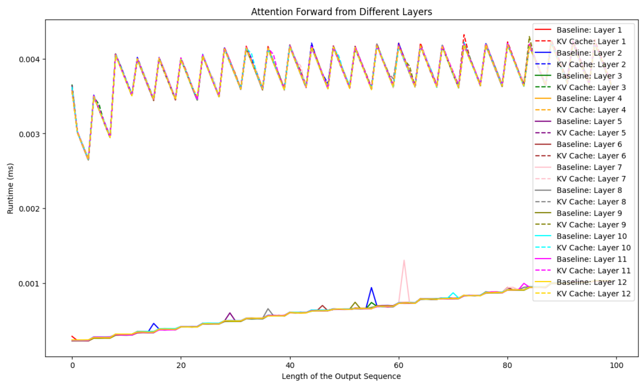

Figure 6. Attention Forward Comparison of Generating 100 Tokens

combined the attention layer with the matmul layer before it, so that we could store K and V once they were calculated. Since K and V were large and different in different layers, we stored K and V from all layers in global memory, i.e. K and V were stored in setup spaces of size (`L`, `B`, T_{cap}, `C`), where T_{cap} was the max number of token the cache could take or it would resize the cache space by doubling T_{cap}.

As a result, the K-V Cache attention kernel (fusion of the matmul layer and the attention layer) used $3.8e-3$ ms on average while the original implementation took $6.5e-4$ ms on average. This might because that the K-V Cache attention kernel required $O(L * T * C)$ space, where L is the number of layers, which spent more time in loading. As shown in figure[6], the runtime of the K-V Cache attention forward was a zigzag line, indicating that the cache hit and cache miss alternatively appeared when the same K-V Cache is visited. In our current implementation, we did not take a step further to mitigate this loading issue. However, the increasing rate of the runtime to sequence of the K-V Cache model ($7.19e-6$) was slower the rate of the baseline model($5.05e-6$), which indicated that as the length of output sequence increases, the baseline model would take more time and the K-V Cache model might have a chance to outperform it.

3.2 Kernel Fusion

There are some intermediate variables when performing the GPT-2 forward pass. For example, `attproj` and `fcproj` between the matmul layer and the residual layer, as well as `fch_gelu` between the matmul layer and the gelu layer. These intermediate variables use extra space and time for loading. Hence, combining layers can remove these intermediate variables and save space and time for the program.

In our implementation of combining kernels, we used CUTLASS, a CUDA template library, for its advantages in combining general expression matrix multiplication (GEMM) + epilogue (for post-processing) as one fused kernel. The original version of matmul kernel in the GPT-2 model is GEMM + bias, which can be replaced with `cutlass::gemm::device::Gemm` + `cutlass::epilogue::thread::LinearCombination`. The setup of the kernel:

```
using ElementA = cutlass::tfloat32_t;
using ElementB = cutlass::tfloat32_t;
using ElementOut = float;
using LayoutA  = cutlass::layout::RowMajor;
using LayoutB  = cutlass::layout::
    ColumnMajor;
using LayoutOut  = cutlass::layout::RowMajor
    ;
using ElementAccumulator = float;
                       // A*B
using ElementComputeEpilogue =
    ElementAccumulator;  // alpha
using MMAOp = cutlass::arch::OpClassTensorOp
    ;
using SmArch = cutlass::arch::Sm80;
using ThreadblockShape = cutlass::gemm::
    GemmShape<128, 128, 32>;
using WarpShape = cutlass::gemm::GemmShape
    <64, 64, 32>;
using MMAOpShape = cutlass::gemm::GemmShape
    <16, 8, 8>;
using SwizzleThreadBlock = cutlass::gemm::
    threadblock::
    GemmIdentityThreadblockSwizzle<>;

using EpilogueOp = cutlass::epilogue::thread
    ::LinearCombination<
    ElementOut,
    128 / cutlass::sizeof_bits<ElementOut>::
        value,
    ElementAccumulator,
    ElementComputeEpilogue,
    cutlass::epilogue::thread::ScaleType::
        NoBetaScaling
>;

using Gemm = cutlass::gemm::device::Gemm<
    ElementA, LayoutA,
    ElementB, LayoutB,
    ElementOut, LayoutOut,
    ElementAccumulator,
    MMAOp,
    SmArch,
    ThreadblockShape,
    WarpShape,
    MMAOpShape,
    EpilogueOp,
    SwizzleThreadBlock,
    3 // Num of stages
>;
```

Listing 1. CUTLASS kernel for matmul layer

For the matmul layer and the gelu layer, we just used another epilogue: `LinearCombinationGELU`. For the matmul layer and the residual layer, we used `GemmUniversalWithBroadcast` and `LinearCombinationBiasElementwise`.

Despite kernel fusion, CUTLASS also allows for customized configuration of the kernel. As shown above, we let the input for GEMM be TF32, which allows our program to use Tensor Core operations for fast calculation. However, converting from FP32 to TF32 results in a decrease in precision. By running the `test_gpt2` test, the maximum error increased to 0.080241 and the RMSE reached 0.002826, when running the code with the matmul kernel and the gelu kernel fused and operated in TF32. Therefore, to leverage the speedup provided by the Tensor Core operation and control the error under an acceptable threshold, we let the matmul+gelu fused kernel and matmul+residual fused kernel operate in TF32 and keep matmul kernel operating in FP32 (using SIMT operation). Thus, the error would not be too large to break the performance of GPT-2 model. At this point, the maximum error was 0.047246 and the RMSE was 0.002478.

Since CUTLASS has a strict address alignment, we had to allocate extra space to store the temporary values of input and output for CUTLASS operations. Hence, the number of calls of `cudaMalloc`, `cudaMemcpy`, and `cudaFree` increased. As shown in Table[4], the extra memory-allocation overhead was worthwhile, yielding a 2.2× performance improvement. (The speedup of matmul kernels is approximated as the average runtime of the kernels of the baseline divided by the average runtime of the ones of Kernel Fusion model since some matmul calls were splitted into fused kernels.)

4 Test

4.1 Unit Test

For fused kernels implemented with CUTLASS, we have designed unit tests for each of them to ensure that they can produce expected output. In the unit test of matmul+gelu fused kernel, for example, two matrices and a bias vector are randomly generated, and then we compute outputs with a naive CPU implementation and the CUTLASS kernels on GPU. Finally, we compare the results of these two calculation routines to verify the correctness of the fused kernels.

4.2 Integration Test

Since the original `test_gpt2` did not support testing K-V Cache and fused kernels, we applied other integration tests for these two optimizations. Since K-V Cache demonstrates its ability over multiple consecutive GPT-2 forward passes

Kernels	Baseline	Kernel Fusion	Speedup
cudaMalloc	0.8111ms	12.02ms	−14.8×
cudaMemcpy	60.33ms	49.94ms	1.2×
cudaFree	9.961ms	23.96ms	−2.4×
matmul	130.6ms	2.978ms	11.6×
gelu	0.09405ms	–	–
residual	0.09603ms	–	–
matmul+gelu	–	0.4818ms	–
matmul+residual	–	1.983ms	–
Total	201.9ms	91.36ms	2.2×

Table 4. Total Runtime Improvement of Mainly Involved Kernels in Kernel Fusion

and the original `test_gpt2` evaluates only on a single GPT-2 forward pass, we created a test based on the next token generation code. We reduced the temperature of the model to a small value (`TEMPERATURE` $= 1e-5$) to ensure fixed outputs of multiple calls on the same input tokens for the same model. Then we ran the tests using both the baseline model and the K-V Cache model, and compared the generated tokens. The consistency of the generated tokens from both models are viewed as the pass of the K-V Cache model.

For the model with fused kernels, we replaced the original kernels with the corresponding fused kernels and deleted the intermediate variables used by the removed kernels in `gpt2.cuh`. Therefore, we were able to run `test_gpt2` to check it the fused kernels work properly as part of the GPT-2 forward. The original checks in `test_gpt2` are sufficient for the correctness verification of the fused kernels.

4.3 Perplexity Scoring System

As mentioned above, the original `test_gpt2` did not support testing the K-V Cache and fused kernels, so we needed to design another test to demonstrate the similarity of token generation ability between the baseline model and the K-V Cache model. Hence, we used the perplexity scoring system $\mathrm{PPL}(X) = \exp\{-\frac{1}{t}\sum_i^t \log p_\theta(x_i|x_{<i})\}$, where X is the entire output sequence, t is the length of the output sequence, and $p_\theta(x_i|x_{<i})$ is the probability of generating a token x_i given all previous produced tokens $x_{<i}$. The input sequence is from WikiText-2, C4, and gsm8k datasets. We randomly chose 10 pieces of text of 60 words each. Each piece is with an input sequence of 10 words and the rest 50 words as the target output sequence (for measuring the ability of the model, unnecessary in verifying the correctness of models). We took the average of the perplexity scores of 10 for each dataset of the baseline model. Then we repeated the same test for the optimized model and compared the obtained values with those from the baseline model. Since `TEMPERATURE` $= 1e-5$ is a small value, the generation ability is stable and less random. Thus, if all averages of both models are the same, then both models have the same capacity to generate text. Therefore, the alternative kernels of the optimized model work exactly the same as the original kernels in the baseline model. This shows that the alternative kernels are correct. The perplexity test is in `test` directory with the instruction to run the test.