# GPA-2 Final Performance Report

## Table of Contents:

# Introduction

In this final report for GPA2's implementations and optimizations for the GPT-2 project, we'll be going over some optimizations that we found useful or that we believe would've been very good to have implemented, and we'll talk about:

- The motivation behind the optimization,
- The implementation details regarding it,
- Some notable metrics,
- What went wrong or what challenges we faced, and
- Any potential improvements that we feel are useful or should be done to it.

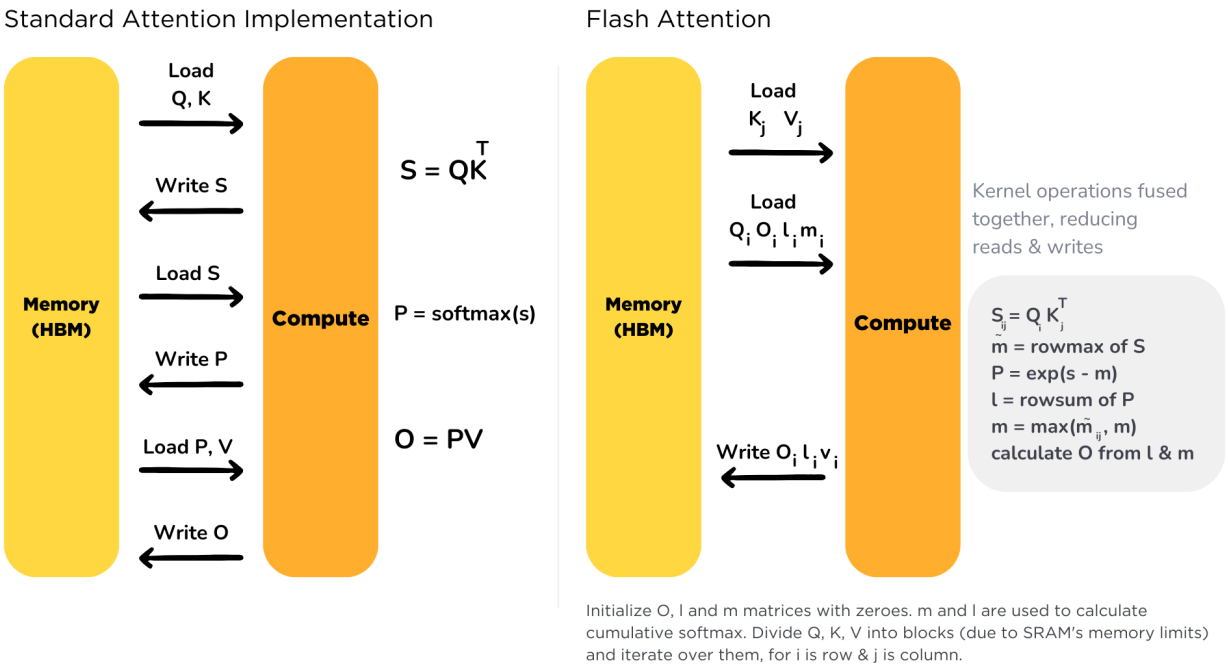
Most of these optimizations come from our Milestone 3 optimizations, and this is because we felt like they were very useful, and improved our code and performance in a noticeable manner that it should be mentioned. For any improvements that haven't directly impacted performance, we mention them anyways due to the technique and implementation needed to add it, and the insight that we learned from attempting the optimizations and what we could improve so that the optimization would have a greater impact.

# Optimization 3 and 9: Flash Attention & Windowed Attention

## Motivation

Traditional attention mechanisms, as used in Transformer-based architectures, require the computation and storage of a full attention matrix with quadratic time and space complexity, i.e., $O(T^2)$ for sequence length T. This becomes a significant bottleneck when scaling models to long sequences or large batch sizes, especially on GPU-constrained systems where memory bandwidth and cache locality are crucial to performance.

Flash Attention was proposed to address these bottlenecks. Avoiding the explicit construction of the attention matrix and using tiling and online softmax normalization significantly reduces both memory usage and latency. The key idea is to compute attention in a single kernel while fusing multiple operations (e.g., dot product, softmax, and weighted sum) to maximize data reuse and minimize memory traffic.




## Implementation Details

The flash attention kernel is implemented by avoiding the allocation of a full attention matrix (T*T) by computing the attention scores on-the-fly using per-thread register storage. Tensor shapes follow the transformer's convention: input shape is (B, T, 3*C), which holds unpermuted input token encodings. qkvr shape is (B, T, 3*C), which is used as a scratch space to hold q k v values. Out has a shape (B, T, C).

Data is first directed to a permutation kernel and permuted into q k v tensors, similar to the naive attention kernel.  In the kernel itself, a local array weight[MAX_T] is used to store attention scores between the current query token and all previous key tokens, where MAX_T is set to 1024 to ensure maximum compatibility. The weight arrays are then scaled and normalized using an on-the-fly softmax algorithm, by first finding the max value, subtracting the value, exponentiating the array, and then dividing by the sum of the exponential values. After this, we calculate the output by taking the weighted sum of the weight array and the token value tensor.

Note that in the main flash attention kernel, we pass the inp tensor as a buffer to store its intermediate output. This minimizes additional memory allocations.

After getting the intermediate output, we use the same unpermute kernel from the original implementation to get the result into the right shape, making it compatible with the subsequent layers.

## Notable Metrics

Kernel Runtime: 576.13μs (330μs)

Compute Throughput: 3.05% (0.31%) (softmax)

Memory Throughput: 9.49% (4.81%) (softmax)

L1 Hit rate: 95.61%

L2 Hit Rate: 90.90%

Total Instruction Count: 3,926,160 (much better than naive's 5,976,384)

Theoretical Occupancy: 100%

Achieved Occupancy: 14.49%

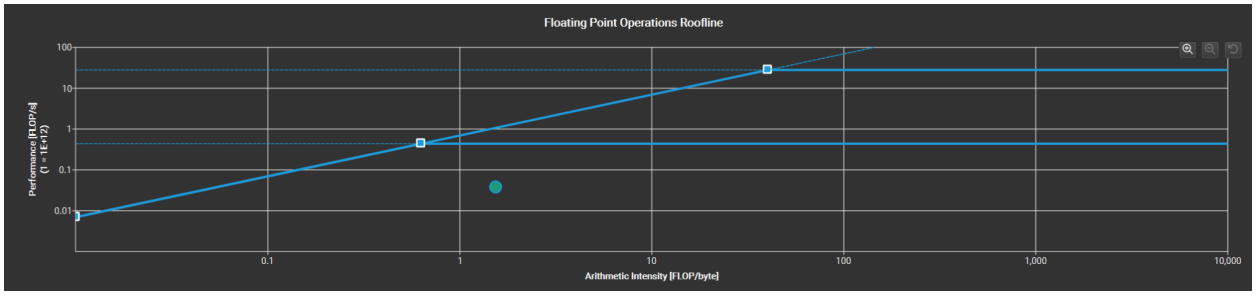
Grid Size: 12 (very small, but at least better than the original softmax's 3)

Branch Efficiency: 55.30%

Overall Verdict: Kernel is as fast as (if not a little bit better) than the original softmax kernel, but significantly slower than the computing attention score and weighted sum kernels (the matrix multiplication part)

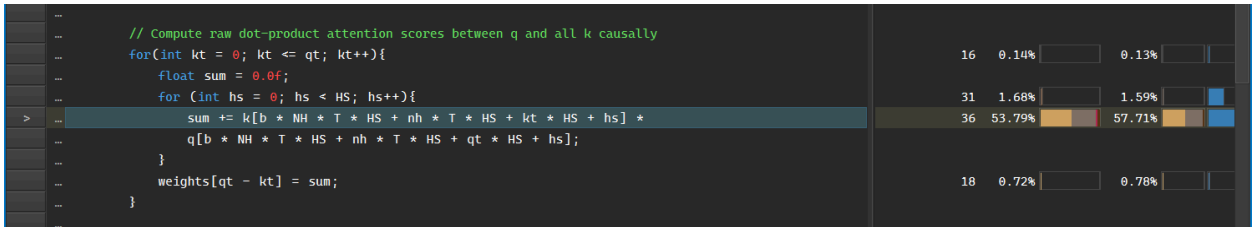
## What Went Wrong?

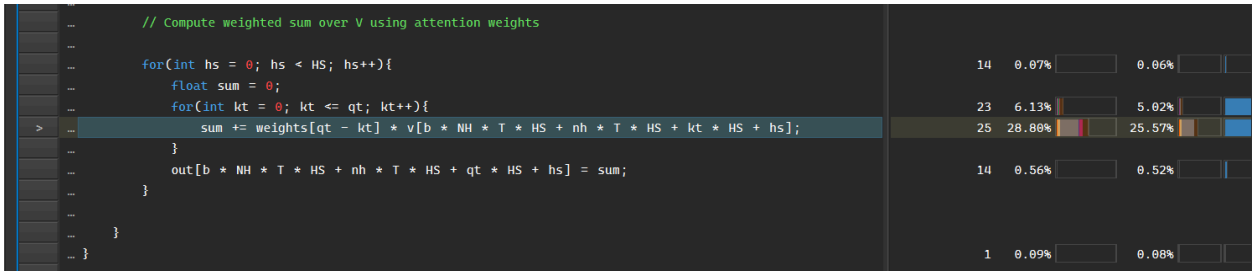
From the metrics above, we can see that the performance of this kernel is not that ideal. It is even more obvious when taking this roofline analysis into consideration.




From this graph we can see that the achieved value is far away from the optimal performance point. Being on the left part of the graph means that the kernel is memory bounded, and the large vertical distance between achieved value and the roofline means there are still large room for performance improvements.

The NO.1 major issue encountered in this flash attention kernel is nothing more than warp stalling.




The primary bottleneck observed in the Flash Attention kernel originates from **warp stalling**, particularly in the nested loop structure within the attention computation. As shown in the Nsight metrics:

- The **dot-product attention score loop** (first screenshot) accounts for **over 53% of total warp stall cycles**, with the inner product calculation being a dominant contributor (line with 53.79% + 57.71% stalls).

- Similarly, the **weighted sum over V** (second screenshot) experiences **28.8% warp stalls**, again concentrated in the innermost accumulation line.

## Why This Happens:

Each CUDA thread is handling an entire (qt) token in the sequence, iterating across all previous tokens (kt) and all head dimensions (hs). This results in:

- **Deep loop nesting** and **non-coalesced memory accesses**, especially when indexing V and Q/K with computed offsets.

- **Thread divergence**, as different threads may process different token lengths (`qt` varies per thread), causing warp inefficiency.

Comparison with Baseline Implementation

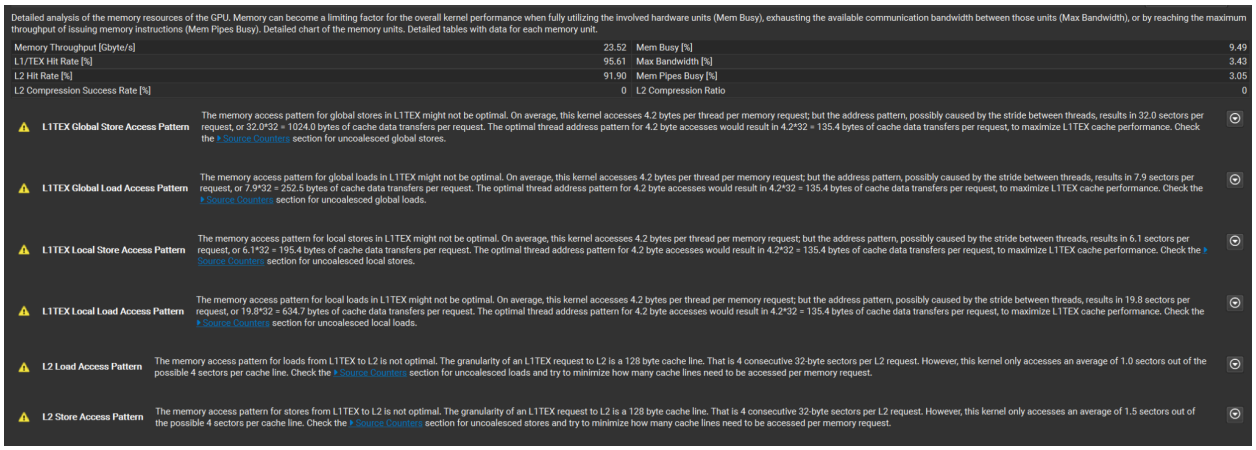
In the baseline (e.g., a matrix-multiplication-based softmax attention), each thread typically handles **one HS element** at a time. This fine-grained parallelism:

- Minimizes loop nesting within the thread,

- Ensures better memory coalescing,

- And significantly reduces warp stalls by **spreading work across threads rather than stacking it within a single thread**.

Besides the warp stalling issue, the second culprit is uncoalesced memory access.




Although we combined three kernels into one, the memory access pattern that were present in those kernels still exists. Primarily, this comes from the load and store from the weight array. Initially, we thought that the weight array is stored in registers. But after examining the report, we found that it is actually stored in local memory. This causes the access to become slower. And also add tons of memory overhead, because each thread is allocated 1024 float memory space, which adds up quickly.

The other contributors include reading of q, k, v tensors, and writing of out tensor. For each aforementioned memory operations, they follows a “consecutive thread load/store consecutive location” pattern. This caused uncoalesced memory operation, and thus longer time spinning, waiting for data to become available.

## Potential Improvements

- Move T iteration to blocks
    - Currently, each thread handles a full token, iterating over all kt <= qt, resulting in nested loops, uneven workload, and warp divergence.
    - We need to restructure the kernel so that each thread works on a specific (b, nh, qt, hs)
    - By reconstructing the kernel, we can achieve better wrapping uniformity and parallelism, significantly lower the time taken to grind the nested loops, eliminating the warp stalling problem that we encountered in matmul and weighted sum.
    - Need to take care about the softmax portion of the kernel: can probably use one thread per T to calculate softmax

- Use shared memory for k, q, v, and weight

    - Since k, q, v are accessed multiple times within loops, it might be a good idea to preload them into shared memory, reducing needs to access global memory
    - Weight array is stored in global memory. Similarly, this per-thread array is also read/written multiple times during the execution of the kernel, so storing it in shared memory can be beneficial. This is helpful if we're going to encompass T over block.

- Loop unrolling
    - The inner-most loop over hs can be unrolled (either manually or using #pragma unroll) to reduce loop overhead and enable more aggressive compiler optimization.
    - This can benefit higher instruction throughput

# Optimization 4: Reduction

## Motivation

In a short and simplified way, reduction consists in making a commutative and associative series of operations more efficient. During our GPT-2 forward pass, there are many points where we are able to utilize and take advantage of the efficiency that Reduction brings in order to have a faster forward pass runtime.

## Implementation details

We had some issues during our implementation and were not able to implement it fully, however we do have notes during the attempts in implementation.

For the softmax kernel, there are two points where we can run reduction and one point where we can run scan, however since there are three unique points that would require this and they rely on each other's data, we'd have to declare three different kernels - possibly two if we can combine the two reduction runs in a single kernel, or create a kernel that is general enough to do both - and suffer the launch overhead for these three kernels.

Residual kernel does not have an associative operation occurring in a loop, so reduction can't be used here.

Matmul's original implementation, and possibly some of the optimized versions from Milestone 2, are under consideration for having reduction applied here. The operations

consist of a sum of products, both of which are associative and commutative as long as you're summing the correct pairs of products.

The layernorm kernel is also under consideration, however it has similar challenges to the softmax kernel as we'd need to test if the benefits from the reduction kernel aren't being cancelled out by the launch overhead of the different kernels we'd need to implement for the reduction optimization.

The GELU kernel also doesn't have an applicable commutative and associative series of operations. Not applicable for reduction.

The Encoder kernel also doesn't have an applicable commutative and associative series of operations. Not applicable for reduction.

Attention kernel does have two points where it may be possible to do reduction. The two possible points are in the "compute_attention_scores_kernel" and the "attention_weighted_sum_kernel" as they both do an associative and commutative series of operations.

## Notable Metrics

Not applicable as the implementation wasn't complete by the time of writing.

## What Went Wrong?

We simply didn't have enough time to complete the implementation, however we're positive that it would've led to an improvement in our GPT-2 implementation.

## Potential Improvements

Finishing the implementation, and adding the optimization to all the different points mentioned above.

# Optimization 5: Configuration Sweep/Optimization

## Motivation

When running our optimized code, we need to continuously tweak and look at our block size to ensure that our program runs in an optimized way that also uses as much of the streaming multiprocessors (SMs) as possible. However, compiling, sbatch-ing, then finally having the code run is a huge time-costly series of tasks. Under such a structure, it might take hours to look at the results that come from running GPT-2 under different block sizes.

So how do we solve this? One way is to create a script that automatically compiles our program under a specific series of block sizes, rename the compiled program executable so that when we change our block size for a future run, we won't be re-running the same executable, and finally run sbatch with our modified executable.

## Implementation details

This optimization is contained in three new files.

First, we create a new file, we call ours "config.h" and put it in the same directory as our kernels. This file has the declaration for block size, and every kernel will import it and use its value for block size. This saves us from having to open each kernel and setting the BLOCK_SIZE variable to whatever we want to test.

Second, we created a new python file called "sweeper.py". This file declares the block sizes that we want to test as a python list.

It then iterates through each block size in the list and for each item in the list:

Opens our "config.h" file and updates the BLOCK_SIZE variable.

Uses the OS library to run "make all".

Appends the size as a string to the initial executable name, so if our initial executable is called "test_gpt2", our new name is: "test_gpt2_{BLOCK_SIZE}".

Finally, our script calls our third file into play and then iterates to the next size to do everything all over again.

Our last file is a modified version of the original slurm file that we were originally provided. When called, it also takes in the name of the executable as input. This way, we can srun our new and renamed executable under the same conditions as our original slurm file, and be able to reuse the file to srun a different executable.

## Notable Metrics

The most notable part of this script was that we were able to run and queue 5 different jobs within seconds of executing the sweeper script. However, we weren't able to compare SM utilization due to the workload near the end of the semester + we still had some issues within the script that'll be mentioned in the next section. However, even with such issues, it's incredibly useful to have such a script since, before the sweeper optimization, we had to spend minutes doing the entire process and making sure everything was consistent. Adding together the potential improvement below would also make it much more useful as that's an optimization that's difficult to do in-person.

## What Went Wrong?

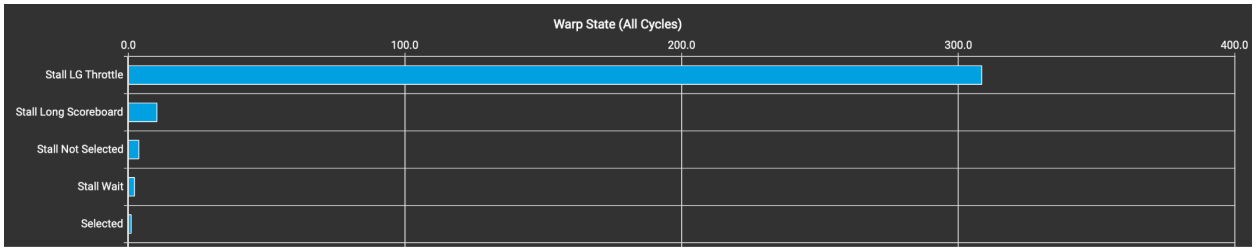
When running the script, our executables were able to be queues and did run within the delta clusters, however we had an issue with a cublas library that made it so that our runs ended in error. Our next steps would have been to fix this issue and compare the SM utilization and runtime between different block size declarations.

## Potential Improvements

Instead of having all our kernels run the same block size globally, we can introduce a second loop in our program so that we change and test all possible combinations of block sizes in our program. For example, kernel A would run block size = 256, but kernel B would run block size = 512 in the same execution of GPT-2.

# Optimization 6: Constant Memory

## Motivation




According to our profiling result for the baseline, we found that for the matmul kernel that took longest time, almost 94.6% of total warp cycles 326.1 were spent on LG Throttle, meaning that the global loading instructions were stalled for busy global loading units. For kernels like matmul and layernorm, the same weight and bias were accessed repeatedly by different sequences, leaving space for constant memory optimization.

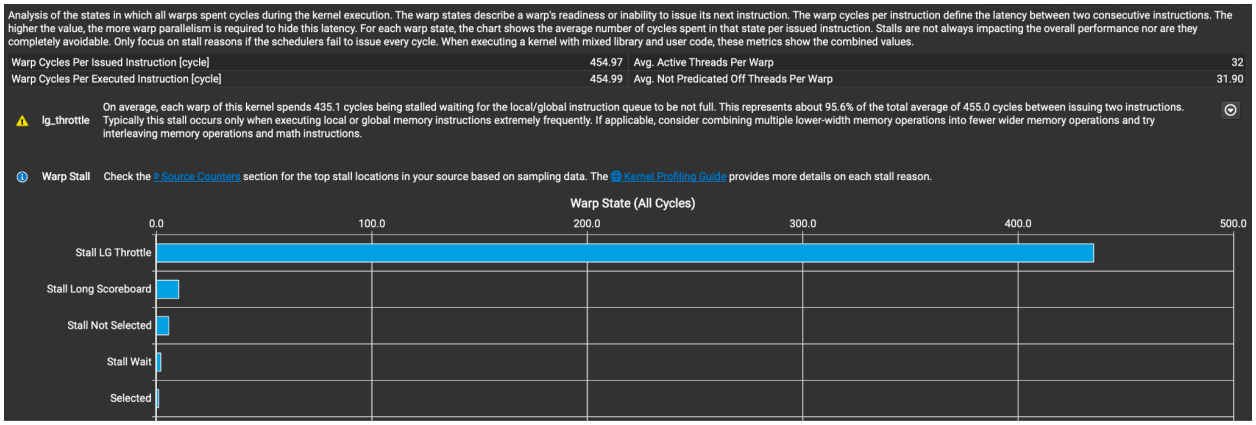
## Implementation details

For layernorm kernel, the largest size for weight and bias are fixed to be 3072. However, for the matmul kernel, the size for weight can be a maximum of 738x3072, which cannot fit in the constant memory. So we only implemented constant memory for bias in the matmul kernel.

## Notable Metrics

The L1 Cache Throughput is increased from 89.19% to 92.61%, the L1 Cache Hit Rate increases from 60.64% to 72.93%. All of these metrics have shown that the implementation of constant memory successfully increases the data reuse from L1 Cache, reducing the runtime of the largest matmul kernel by 4ms (104.77ms to 100.79ms).

## What Went Wrong?




However, from the profiling result we can see that the warp stalling rather gets worse. The warp stall cycle has increased to 455 cycles. This might be because our original baseline implementation is already heavily memory-bounded, while we were using cudamemcpy to load in the constant memory, it can rather increase the workload of memory loading.

## Potential Improvements

- While the weight size in the matmul kernel (738x3072) exceeds constant memory capacity, you can still leverage constant memory for smaller segments of the weight matrix that are frequently accessed.
- Use streams and asynchronous cudaMemcpyAsync to load constant memory for the next operation while the current kernel is still running.

# Optimization 7: __restrict__

## Motivation

By using the __restrict__ keyword, we inform the compiler that the pointers marked with __restrict__ do not alias (i.e., they point to distinct memory regions) under the scope of the function. This allows the compiler to optimize memory access patterns more aggressively. For exampling, knowing that the pointers are not aliased, the compiler can optimize by prefetching memory of the pointers and loading the pointed memory into registers to avoid reloading from global memory, which can lead to higher performance.

## Implementation details

Since almost every kernel has unaliased pointers, we implemented __restrict__ to almost every pointer in every kernel. For kernels in attention layer, there are pointers like q, k, v which are all essentially pointing to memory under qkvr:

```
float *q, *k, *v;
q = qkvr + 0 * B * T * C;
k = qkvr + 1 * B * T * C;
v = qkvr + 2 * B * T * C;
```

These seems unaliased while since the size of each query, key, and value are strictly specified, we can guarantee that they won't access overlapping memory.

## Notable Metrics

However, based on the profiling result there were no notable differences before and after implementing __restrict__.

## What Went Wrong?

Still, our main bottleneck of the baseline kernel is global memory accessing, which won't be significantly improved by reducing aliasing checks.What's more, when the global memory accessing pattern is bad enough, prefetching and reordering instructions won't help much with the situation.

# Optimization 8: Loop Unrolling

## Motivation

Loop unrolling is an optimization technique that reduces these overheads by manually expanding the loop body, allowing the compiler to:

- Eliminate Branching: By unrolling the loop, the condition check is reduced.
- Optimize Memory Access: Precomputed index values reduce the need for repeated address calculation.
- Leverage Instruction-Level Parallelism: The compiler can reorder and pipeline instructions inside the unrolled iterations.

## Implementation details

We implemented a loop unrolling of 4 iterations for loop contained kernels like attention kernels, layernorm, and matmul. The softmax kernel is not suitable for unrolling because of the Non-Uniform Loop Bound. Each thread may perform a different number of iterations (masked softmax), leading to severe warp divergence. Unrolling may worsen this divergence, as some threads may exit the loop early while others continue.

## Notable Metrics

Although we observed a reduced number of warp stall cycles, from 454 to 435, and a 1ms reduction of runtime, the loop unrolling of 4 iterations didn' show a significant improvement.

## What Went Wrong?

Again the loop unrolling was not able to improve the memory access pattern. Previously we thought that by implementing loop unrolling of 4, in one iteration each thread would load in four float elements, which could fit in 1 burst, thus improving the memory coalescing. However, after consulting online resources we found that this is not the case. First, the memory burst in CUDA is warp wise meaning that a single memory transaction of 128 bytes is triggered for one warp. However, when each thread is accessing 4 consecutive elements in each iteration this rather breaks the coalescing.

## Potential Improvements

We might want to either increase the iterations being unrolled or improve the essential memory access pattern.

# Optimization 10: Further Matmul Optimization (without cuBLAS)

## Motivation

In CUDA, memory transactions are most efficient when data is accessed in a coalesced, vectorized manner. By using vectorized memory loads (e.g., float4, float2), we can achieve:

- Higher Memory Throughput: A single vectorized load instruction can replace multiple scalar load instructions.
- Fewer Memory Transactions: The GPU can coalesce these vectorized loads into a single transaction.
- Reduced Instruction Overhead: Fewer instructions are issued for the same data volume.

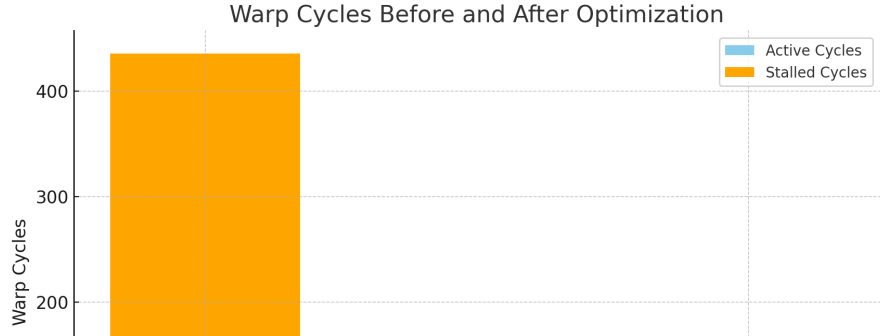
In our GPT-2 forward pass kernels, we identified that the most time-consuming kernels, such as the matmul and layernorm, involve significant memory access. Our profiling revealed that a 95.6% of the warp stall is spent on memory read and write operations. By leveraging vectorized memory access, we aimed to significantly improve the memory efficiency of these kernels.

## Implementation details

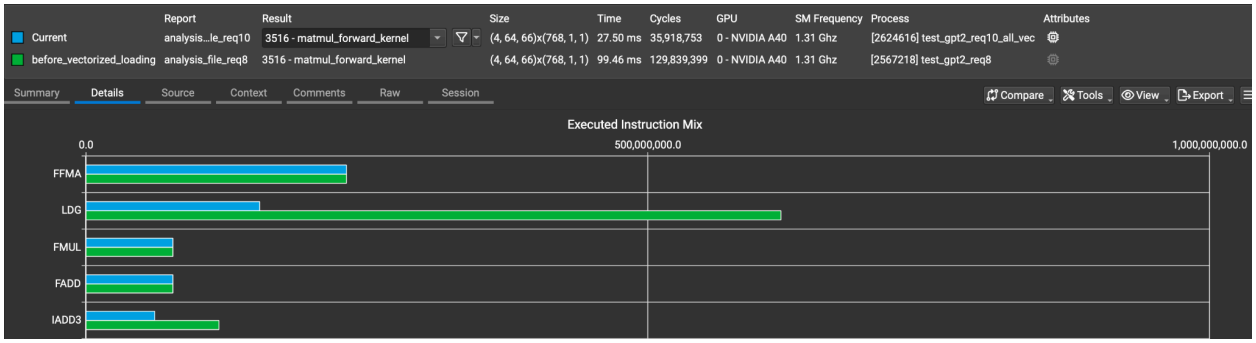
We utilized vectorized load of float4, which can fully utilize the 128-byte memory transaction size. We implemented it for attention kernels (except for attention_weighted_sum_kernel since the v access pattern is not consecutive, so vectorize loading might not bring in improvement), for encoder kernel, gelu, layernorm, matmul, and residual kernel. Still, not for softmax for its dynamic loop bound and irregular access pattern.

## Notable Metrics

We significantly reduced the warp cycles from 435.4 cycles to 107.78 cycles, with 68.4% of them being stalled (previously 95.6%).



What is also worth noticing is that We also reduced the LDG (loading global memory) instructions from 61837984 times to 154533888 times, which is **exactly a factor of 4.**




What's more, the vectorized loading also significantly improve the memory access pattern: it reduces the 85% of uncoalesced global access to 48%.

By significantly reducing the global memory access instruction and warp stall, we reduced the runtime from 99.46ms to 27.5ms.

## Potential Improvements

Vectorized loading can be combined with shared memory tiling and register tiling to further reduce global memory stalling.

# Proposed Optimization 1: Multi-GPU

## Motivation

As the size of the input grows, the computational and memory demands of each kernel scale proportionally, often exceeding the capacity of a single GPU. To mitigate these issues, we adopt a multi-GPU strategy, partitioning the workload across multiple devices to reduce memory pressure and improve computational parallelism. By slicing the batch dimension and distributing the computation across available GPUs, we unlock significantly higher throughput and enable large-batch inference or training on models that would otherwise exceed single-device limits. This approach also improves utilization of modern multi-GPU nodes by allowing concurrent execution of the same kernel on independent data slices.

## Implementation details

The implementation of each kernel in our GPT2 project remain roughly the same, so here we will be using the attention kernel as an example.

Note that in original implementations, we use attention_forward() as the wrapper host function to launch kernel; in our optimization, we rename the function into launch_attention_forward() and wrap that inside another host function, which handles memory allocation and kernel launch for multi GPUs.

Inside the host function, we first get the available GPU device count, then we proceed to calculate how much of B each GPU is responsible for.

The batch dimension B is divided into B_local = ceil(B / num_gpus), with each GPU assigned a disjoint slice of the input. Each device receives a portion of the inp tensor (shape (B_local, T, 3*C)), as well as corresponding scratch memory (qkvr) and output space (out). We then proceed to a for loop with num_gpus iterations. After setting the device, all data is copied using cudaMemcpy from GPU 0 into per-GPU memory pools, and each GPU launches its own instance of launch_attention_forward()

Within each GPU, the flash attention kernel flow remains unchanged:

- Inputs are first permuted via permute_kernel to obtain Q, K, and V (each shaped (B_local, NH, T, HS)).

- These tensors are passed into flash_attention_forward_kernel, which performs the online softmax and weighted sum computation, storing the result back into the inp buffer.

- This buffer is finally passed through unpermute_kernel to return the data to the expected (B_local, T, C) output format.

The kernel remains the same in the sense of operation and structure, but now each GPU operates independently, enabling concurrent compute across devices.

Weights and biases are also replicated across GPUs, and the final outputs are gathered from each GPU stream using another cudaMemcpy call to copy back to GPU0, which is done inside another for loop.

At the end, we free the memory allocated on each GPU.

By avoiding in-kernel peer-to-peer communication and synchronizing only at the end of each stream, we ensure minimal overhead while preserving correctness.

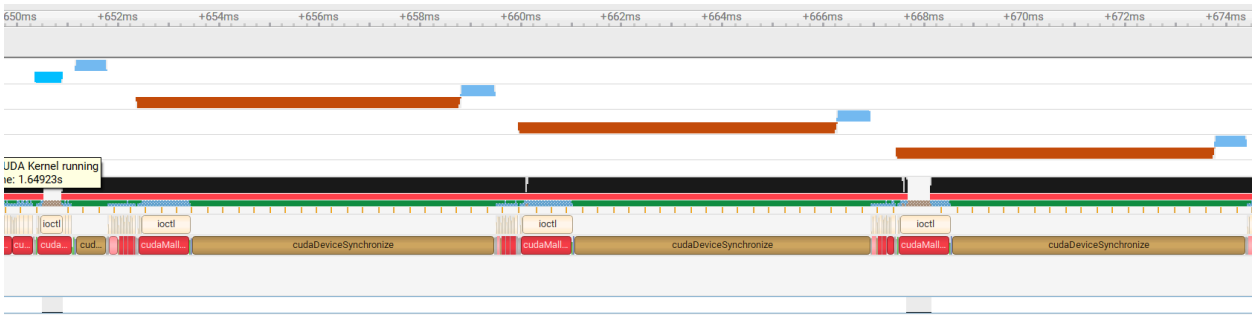
## Notable Metrics

All data are collected using 4 GPU settings

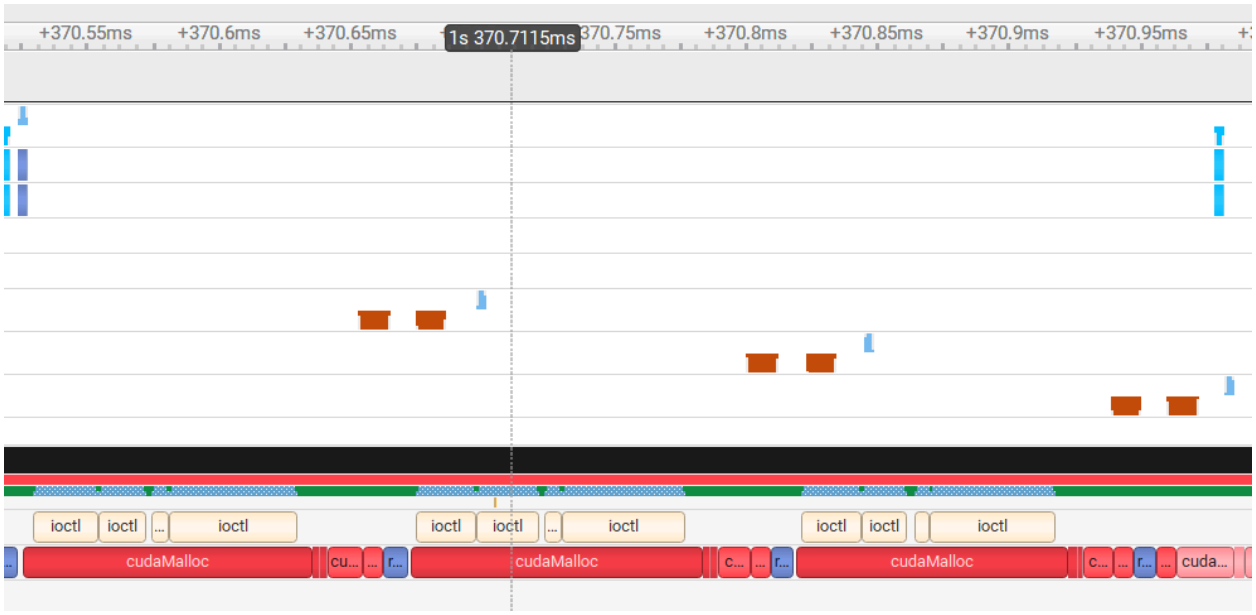
- Most multi GPU kernels each only take ⅓~¼ time of the original kernel runtime
- However, the overall runtime is often longer than the naive implementation due to significant overhead from cudaMalloc and especially cudaMemcpy operations, which dominate the performance gains from parallelization
- For example, some kernels' performance are shown below
    - Layernorm: 660μs (naive) 638μs (4GPU) (minor improvement)
    - Flash Attention: 576μs to 1.267ms (some performance drop, but still "parallel")
    - Residual: 4.1μs to 463μs (memory overhead dominates)
    - CuBLAS Matmul: 1.788ms to 21.233ms (same as above)

## What Went Wrong?

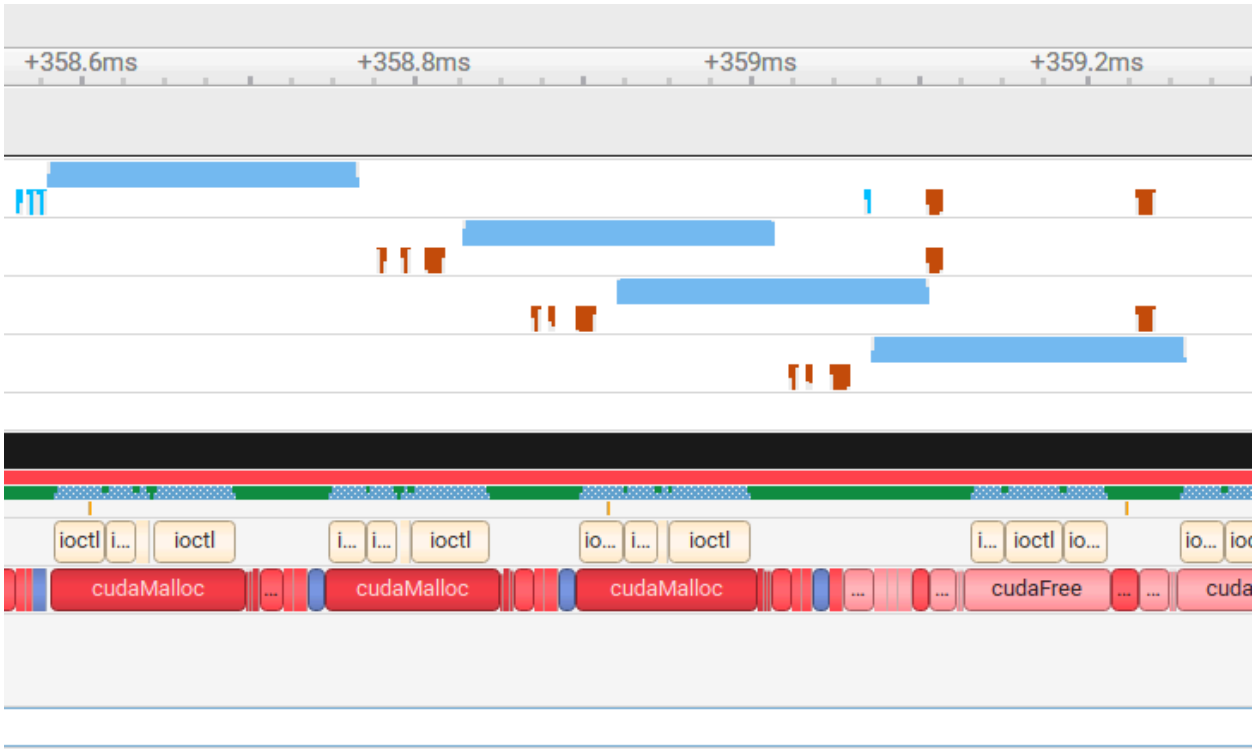
While the multi-GPU implementation provided noticeable speedups for individual kernel execution, several limitations prevented it from fully outperforming the single-GPU baseline. A primary bottleneck stems from the cost of memory operations. For kernels like matmul, which require copying and allocating multiple large buffers (e.g., inp, weight, bias, and out) to each GPU, the overhead from repeated cudaMalloc and cudaMemcpyAsync operations becomes significant. These memory operations add considerable latency and can often dominate the kernel's execution time, especially when done independently on each device. The figure below shows how large memcpy overhead damages the performance of the cuBLAS matmul kernel. Note that each kernel runs in ~670μs, which is ⅓ of the original cuBLAS implementation. If perfectly parallel, the runtime should be cut to a third.




Another example from the residue kernel shows another situation where performance is hampered by malloc. Note that the light blue segment is kernel running.




In contrast, kernels such as layernorm operate on smaller elements and therefore benefit more from parallelization. With reduced memory transfer and allocation costs and greater concurrency across GPUs, these lighter-weight kernels see a net gain in overall runtime. This discrepancy highlights that the effectiveness of multi-GPU parallelism is highly sensitive to memory traffic volume and kernel complexity. Without overlapping memory transfers or persistent allocations, heavier kernels can be dragged down by setup overhead despite shorter actual computation time.




In addition to that, some kernels tend to benefit less from input partitioning. For example, although the input is cut to one-fourth when distributed across 4 GPUs, the flash attention kernel's runtime remains roughly the same. This is because each thread in the kernel already operates independently over the full sequence dimension, and the workload is dominated by nested loops over T and HS, not B. As a result, simply reducing the batch size per GPU does not significantly reduce the total work done per thread or improve warp-level parallelism. Moreover, the memory access pattern and per-thread register usage remain constant, limiting scalability through naive batch slicing.

## Potential Improvements

Although the multi-GPU implementation introduces parallelism and computational gains in many kernels, its effectiveness can be further improved by addressing key inefficiencies and selectively targeting high-impact areas.

1. Pre-Allocation of Memory: One major overhead in the current approach stems from repeated cudaMalloc and cudaMemcpyAsync calls on each GPU before every kernel execution. This is especially costly in compute-heavy kernels like matmul or attention, where large parameter tensors must be copied to every

device. A more efficient strategy would be to pre-allocate and persist memory on each GPU during model initialization (e.g., in gpt2.cuh), and reuse these buffers throughout the forward pass. This would eliminate redundant memory allocations and transfers, significantly reducing latency.

2. Focus on Compute-Heavy Kernels: Multi-GPU execution is most beneficial for kernels with long runtimes or large workloads. In our case, many kernels operate on relatively small inputs and already execute quickly on a single GPU. For such cases, the overhead of synchronization and data transfer outweighs any benefits. Therefore, multi-GPU strategies should be reserved for slow, compute-intensive kernels, such as large matmuls or long-sequence attention, where the workload justifies the distribution cost.

3. Target Kernels That Respond Well to Input Partitioning: Not all kernels benefit equally from slicing the batch dimension. For example, while matmul and layernorm scale well with input partitioning, flash attention showed limited runtime improvement even when the batch was divided across GPUs. This suggests that certain kernels are more sensitive to sequence length or thread-level parallelism rather than batch size. Future optimization efforts should focus on kernels that are empirically responsive to input partitioning, and avoid complicating fast or non-scaling kernels with unnecessary multi-GPU logic.

# Proposed Optimization 2: KV Cache

## Motivation

In practice, generative LLMs like GPT-2 are most effective when used in long conversations or with much larger context windows (e.g. ChatGPT). However, processing more tokens requires computing increasingly larger Q, K, and V matrices. Traditionally, Q, K, and V are computed on demand by multiplying the input tokens with the weight matrices. For every new token that's generated, the matrices grow larger and larger. In fact, the runtime increases by a quadratic factor for every new token!

The KV cache aims to tackle this problem by avoiding redundant computation, and processing a single token at a time rather than the full input sequence.

## Implementation Details

The K and V caches are stored in the GPT2 struct as float* arrays. We will maintain a cache of size (L, B, max_seq_len, C) for K and V, where L corresponds to layers, B corresponds to batch, max_seq_len is the size of the output, and C is the channel size, or the size of each token vector. We will also have an int cache_len that indicates whether the cache is initialized or not and the size of the cache. The cache is initialized if the cache_len != 0.

At a high level, the forward pass will need to be modified. During the first pass, the forward pass will process the entire input sequence as normal and copy K and V to the cache. Once the cache is warm, the forward pass will need to process only a single token and utilize the KV cache correctly.

A few changes were made when handling the output of the forward pass as well. Since only one token is processed, only one token is produced instead of the full sequence.

Overall, the following files were modified:

- `next_token_generation.cu`
- `gpt2.cuh`
- `attention.cuh`
- `softmax.cuh`

## Forward Pass Modification - Overview

If the cache is not initialized:

- retrieve QKV from the matmul output like before.
- Split K and V from the QKV shared matrix
- cudaMemcpy the K and V splits directly to the cache.
- Compute attention like before
- Continue the rest of the forward pass

If the cache is initialized:

- Instead of computing the full Q, K, V matrices, only compute the newly generated token
- Split Q, K, and V
- Append K and V to the cache and increment cache_len
- Pass in Q, K_cache, and V_cache to the modified attention kernel.
- The attention kernel is modified to use a single query vector instead of a matrix
- Pass the output through the rest of the forward pass as normal

## Attention

A new attention function handles the computation for a single query. The following was modified:

- Permute kernels
    - The dimensions of Q, K, and V are not the same anymore. Q is a single vector whereas K and V contain the full input sequence. The original permute kernel did not account for this.
- Matrix multiplication
    - Instead of naive matrix multiplication, cuBLAS was used. Additionally, the operation changes from matrix-matrix multiplication to matrix-vector multiplication.
    - `cublasSgemmStridedBatched` is used to process each batch and each attention head independently.
- Softmax
    - The original softmax kernel assumed the input to be a matrix, which is not the case anymore.

## Notable Metrics

We observed a significant speedup compared to baseline.

- Baseline generated an average **9-10** tokens per second
- KV cache generated an average **24-25 (+150%)** tokens per second

The following parameters were used for testing:

- Max sequence length (T): 1024
- Vocabulary size (V): 50,257
- Padded vocabulary size (Vp): 50,304
- Number of layers (L): 12
- Number of attention heads (NH): 12
- Hidden size / Channels (C): 768
- Total number of parameters: 124,475,904
- Number of input tokens: 4

## What Went Wrong?

In the end, the KV cache implementation remained incomplete. The majority of the significant modifications were made, however, the output is not readable. There's likely a bug in the code somewhere. Potential issues:

- Softmax implementation
- Permute kernel
- memory allocation/indexing issues
- Layernorm and gelu kernels might need modification as well

## Potential Improvements

The current KV cache implementation has a few inefficiencies that could be optimized:

1. The full input sequence is encoded every time. This is not needed when the kv_cache is warm, only the last token should be encoded.
2. Similarly, the first layernorm call is always using the full input sequence
3. The various activation pointers are allocated with the full sequence length for simplicity. These could be dynamically managed when processing a single query for efficiency.