ECE 408 GPT Project Milestone 3 Report

Authors: Tianhao Chen, Vasunandan Dar, Raghav Tirumale, Ram Reddy

Department of Electrical and Computer Engineering, University of Illinois

Mentor: Vijay Daita

May 19, 2025

Table of Contents

Introduction

For Milestone 3, our group experimented with several optimizations. They can be grouped up in a few different categories. First, we implemented optimizations which could be considered general practices, such as loop unrolling pragmas, using constant memory, restrict keywords, and using reduction steps. Then we implemented algorithmic optimizations on attention, starting with local attention and moving to Flash Attention 2. We also continued on optimizing the tensor core matrix multiplication to see how close we could get to cuBLAS. With these optimizations, we ran a sweeping script on these kernels to determine the optimal grid sizes to ensure the fastest performance. Following these requirements, we also implemented three proposals, including a simple KV cache, flash attention optimized at a warp level, and kernel fusion. In this report, an in-depth background, explanation of implementation details, analysis of the Nsight reports will be provided for each of these optimizations.

Loop unrolling (req_8)

Background

Loop unrolling is a compiler optimization used to decrease the number of assembly instructions in loops. It is primarily to decrease the overhead of loop control like incrementing counters and loop evaluation. Unrolling is able to decrease the number of instructions by repeating the loop body multiple times in one iteration. For a GPU, this allows more register usage as each thread can utilize more registers per cycle as more of the assembly is defined in one go. Unrolling should be used in moderation as too much unrolling might slow the program as the GPU may not be able to reduce occupancy to meet the unrolled instructions needs. For example, on the NVIDIA A40 GPU, each Streaming Multiprocessor (SM) has 64,000 32-bit registers shared across a maximum of 1,536 threads. If a block uses 1,024 threads, this means each thread can use up to 62 registers before register spilling or reduced occupancy becomes a concern.

Implementation

Implemented unrolling is very simple and only needs a pragma which is a compiler directive. The user only needs to “#pragma unroll <number here>”. There are some rules to use a pragma, like the number that the unroll uses has to be defined in compile time, this means that you cannot use a runtime variable as that does nothing. Some of the more common unrolls are 2, 4 and other multiples of 2. This optimization was done on our base kernels defined in milestone 2.

In our project, we found a couple of locations where there are loops. We had the preatt_kernel, which loops based on T which is the time series. T is always divisible by 4 based on the inputted values of GPT2.cuh. We tested different unroll values and eventually ended up with an unroll number of 8 for the outer two loops. For the inner head size loop HS = C/NH and is always a multiple of 2 and is always above 32 so 8 is also used here for decent unrolling without too much register pressure. We then also used unrolling in the compute_att_kernel, this loops the exact same way as the preatt_kernel and so we used 8 for a good unrolling amount.

Then we found layernorm_forward_kernel which loops based on C which is channels. C is also always divisible by 2 and 128, so we tried different unrolls here upto 32. Lastly, we found one lat kernel with loops: the softmax_forward_kernel which loops based on idx%T. Due to the use of the modulus operator, this loop does not have a predictable iteration count at compile time, making it unsuitable for unrolling.

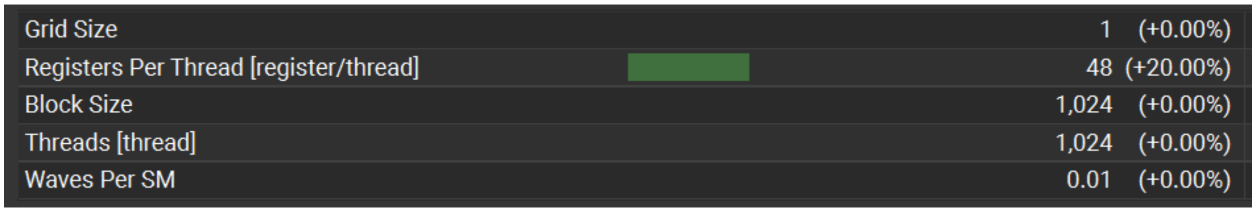
Analysis

After running Nsight systems we got the following times for unrolled kernels. The table shows the times for both the baseline and unrolled versions. Since we unrolled with a moderate amount, the unrolled times always improved performance.

Kernel	Baseline Time (ns)	Unrolled Time (ns)
preeatt_kernel	185,890,995	176,371,761

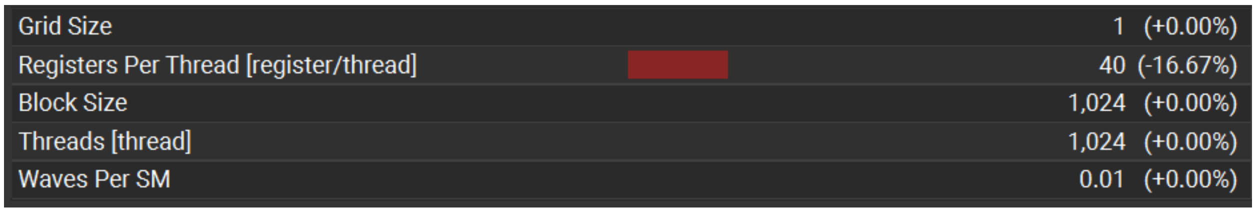
compute_att_kernel	284,895,072	198,717,515
layernorm_forward	7,279,300	7,233,875

Since unrolling does not change much of the kernel itself, there are only a few metrics that change with each call. The most important one is the registers per thread which increased from 40 to 48 registers from baseline to the unrolled versions. This makes sense as we used an unroll amount of 8 for the preatt_kernel and thus it uses 8 extra registers.



Grid Size		1	(+0.00%)
Registers Per Thread [register/thread]	🟩	48	(+20.00%)
Block Size		1,024	(+0.00%)
Threads [thread]		1,024	(+0.00%)
Waves Per SM		0.01	(+0.00%)

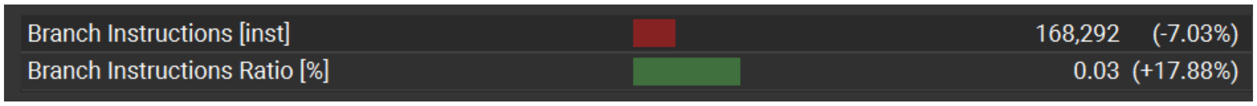
Launch Statistics of Unrolled preatt_kernel



Grid Size		1	(+0.00%)
Registers Per Thread [register/thread]	🟥	40	(-16.67%)
Block Size		1,024	(+0.00%)
Threads [thread]		1,024	(+0.00%)
Waves Per SM		0.01	(+0.00%)

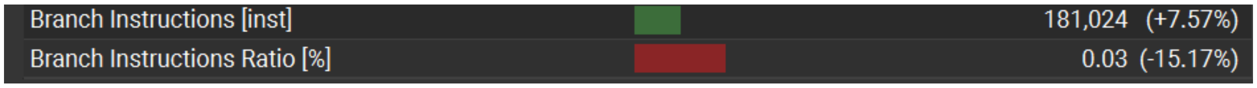
Launch Statistics of Baseline preatt_kernel

Since we do more operations per branch check, the corresponding branch instructions should decrease. Looking at the compute reports statistics, the branch instructions decrease by 7% from the baseline to the unrolled kernel. Due to less branch assembly instructions, the time decreased from 185,890,995 ns to 176,371,761 ns. This also means that branching was likely a significant source of delay for more instructions to run.



Branch Instructions [inst]	🟥	168,292	(-7.03%)
Branch Instructions Ratio [%]	🟩	0.03	(+17.88%)

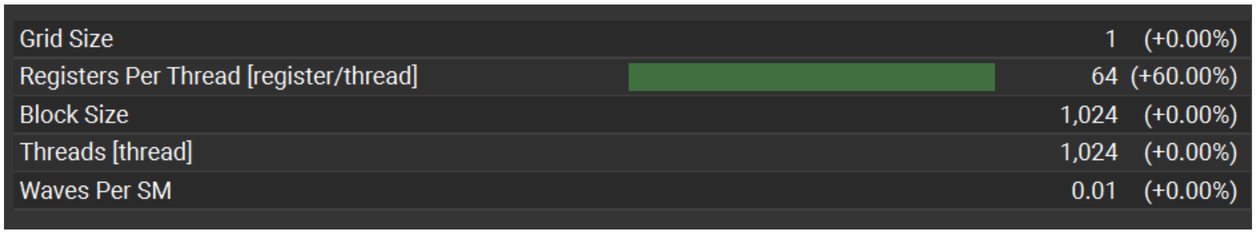
Source Control Statistics of Unrolled preatt_kernel



Branch Instructions [inst]	🟩	181,024	(+7.57%)
Branch Instructions Ratio [%]	🟥	0.03	(-15.17%)

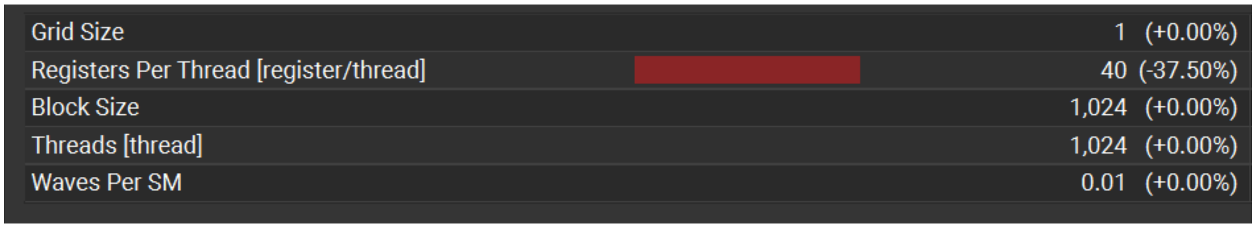
Source Control Statistics of Baseline preatt_kernel

We can also see the same in the compute_att_kernel, however, the compiler was able to perform unrolling on all three loops instead of just one so the register usage went up to 64 registers instead of just 40.



Grid Size	1	(+0.00%)
Registers Per Thread [register/thread]	64	(+60.00%)
Block Size	1,024	(+0.00%)
Threads [thread]	1,024	(+0.00%)
Waves Per SM	0.01	(+0.00%)

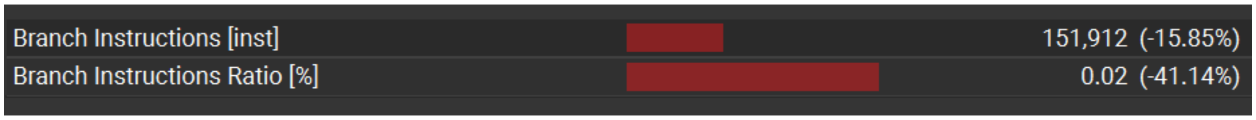
Launch Statistics of Unrolled compute_att_kernel



Grid Size	1	(+0.00%)
Registers Per Thread [register/thread]	40	(-37.50%)
Block Size	1,024	(+0.00%)
Threads [thread]	1,024	(+0.00%)
Waves Per SM	0.01	(+0.00%)

Launch Statistics of Baseline compute_att_kernel

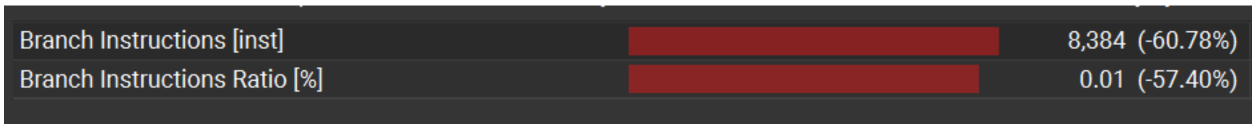
One possible reason that the compiler was able to unroll all three kernels in the compute kernel was because it was used previously, but it is still interesting that the compiler used more registers in the second case. However, this was able to decrease the branch instructions by 15.85% compared to the baseline. Due to using many more registers the time decreased from 284,895,072 to 198,717,515. This likely means that branching was a significant source of delay for more instructions to run.



Branch Instructions [inst]	151,912	(-15.85%)
Branch Instructions Ratio [%]	0.02	(-41.14%)

Source Control Statistics of Unrolled compute_att_kernel

Lastly, we have the layernorm kernel. Although the number of registers per thread increased compared to the base lines and the number of branch instructions decreased by 60% it did not cause a significant time change. The time only decreased from 7,279,300 ns to 7,233,875 ns. This is likely due to branch instructions not causing significant warp delays and threads already doing enough work that unrolling caused minimal changes.



Branch Instructions [inst]	8,384 (-60.78%)
Branch Instructions Ratio [%]	0.01 (-57.40%)

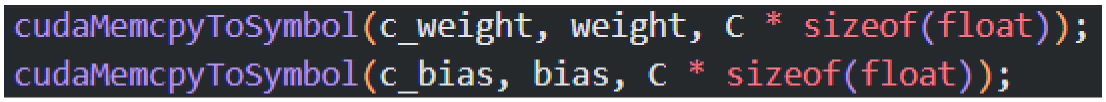
Source Control Statistics of Unrolled layernorm_forward

Constant Memory (req_6)

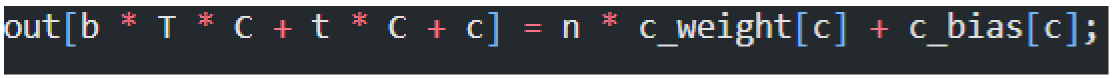
Overview

Constant memory is a small region of read-only memory space, typically 64 KB in size. Like global memory, constant memory is stored in the DRAM. However, because constant memory doesn't change during runtime, that section of memory is aggressively cached into the L1 cache, the fastest of the cache levels. In addition, if a constant variable is accessed by all threads in a warp, that variable can be broadcasted, greatly increasing the bandwidth. Therefore, Constant memory is beneficial for kernels that reuse variables that won't be modified during runtime.

Constant memory was implemented in the layernorm kernel, because the weight and bias arrays were small enough to be placed into constant memory, and because they are size c, each value will be reused for each thread. Implementation was straightforward, as the weight and bias arrays were copied into constant memory, which was used in the kernel.



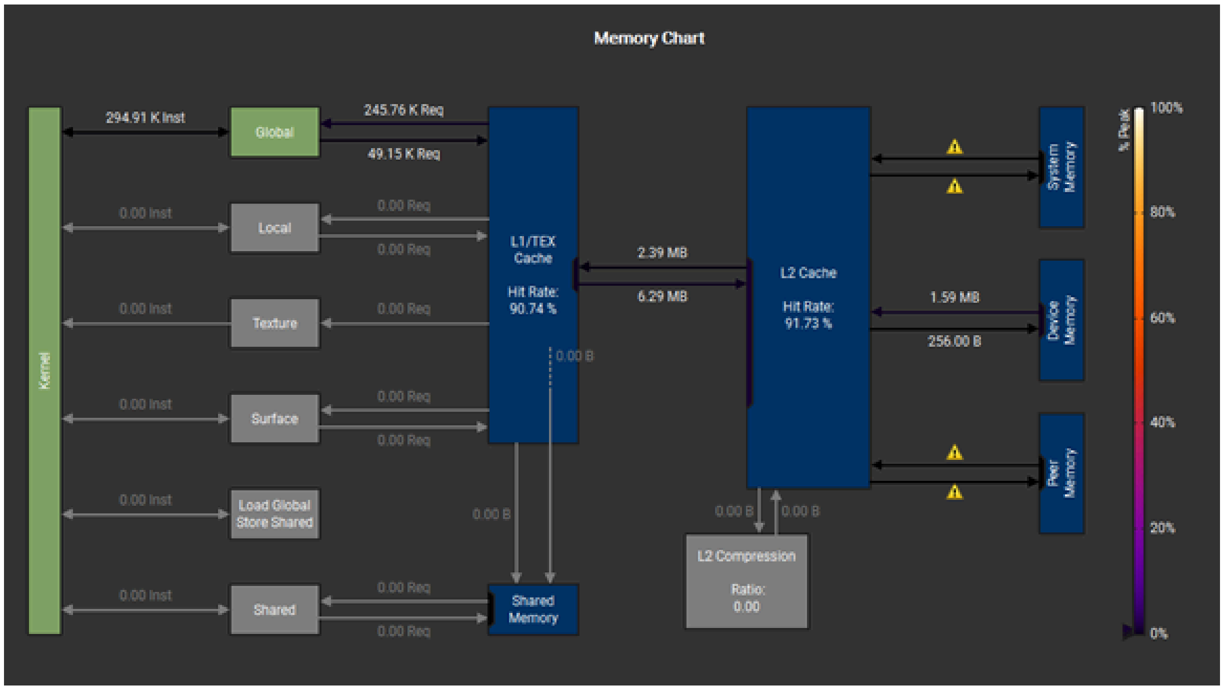
```
cudaMemcpyToSymbol(c_weight, weight, C * sizeof(float));
cudaMemcpyToSymbol(c_bias, bias, C * sizeof(float));
```



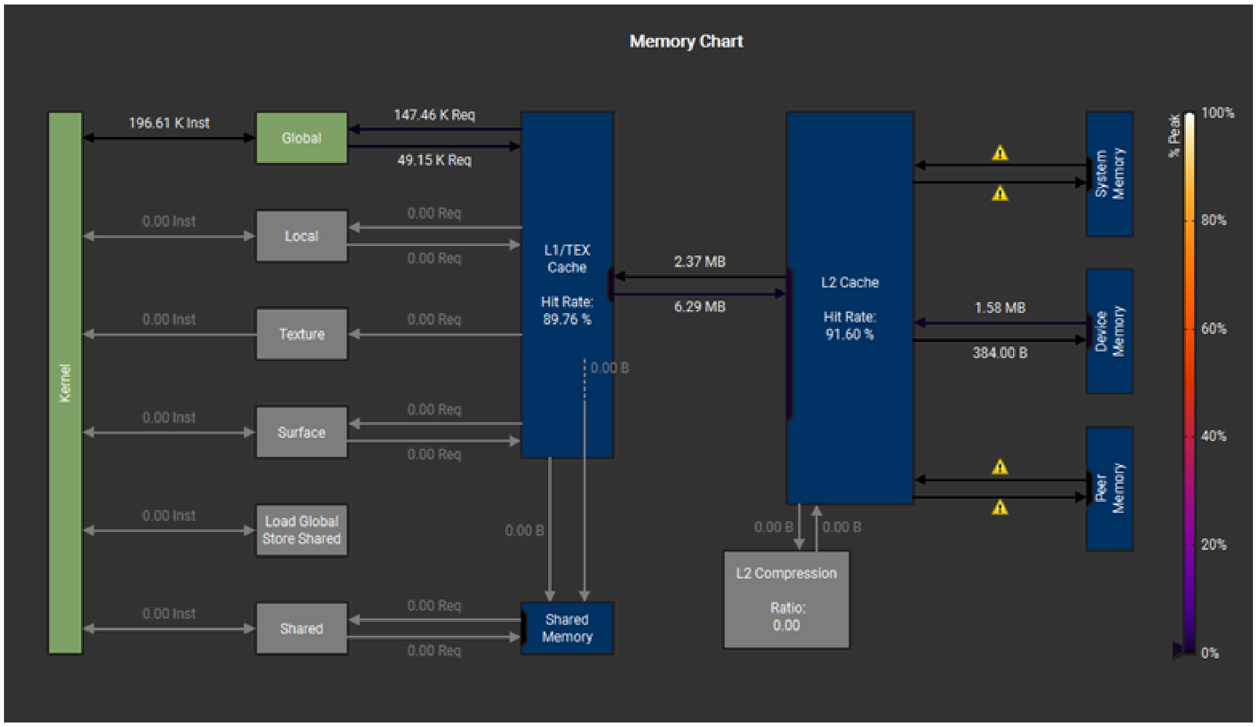
```
out[b * T * C + t * C + c] = n * c_weight[c] + c_bias[c];
```

Analysis

Comparing the Nsight Compute report of the baseline layernorm kernel with the layernorm kernel that's using constant memory, the main difference was the amount of requests and instructions to global memory. While the baseline kernel had 294,910 thousand global memory instructions, 245,760 thousand read requests, and 49,150 thousand write requests, the layernorm kernel with constant memory had significantly less global memory instructions and read requests, with 196,610 thousand and 147,460 thousand respectively. This significant decrease is the result of loading the weights and bias arrays into constant memory because the kernel and L1 cache doesn't have to access global memory to retrieve values in those arrays during execution.

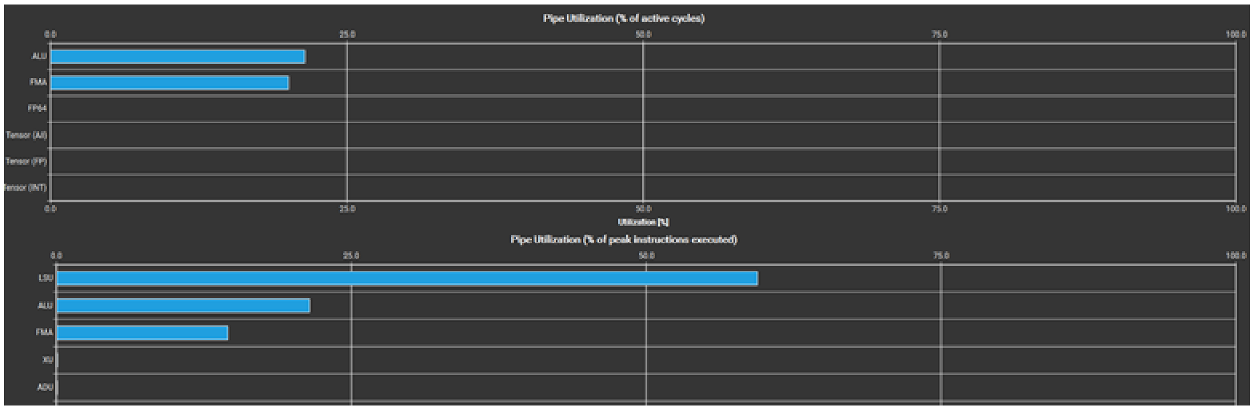


Memory Chart of Baseline Layernorm

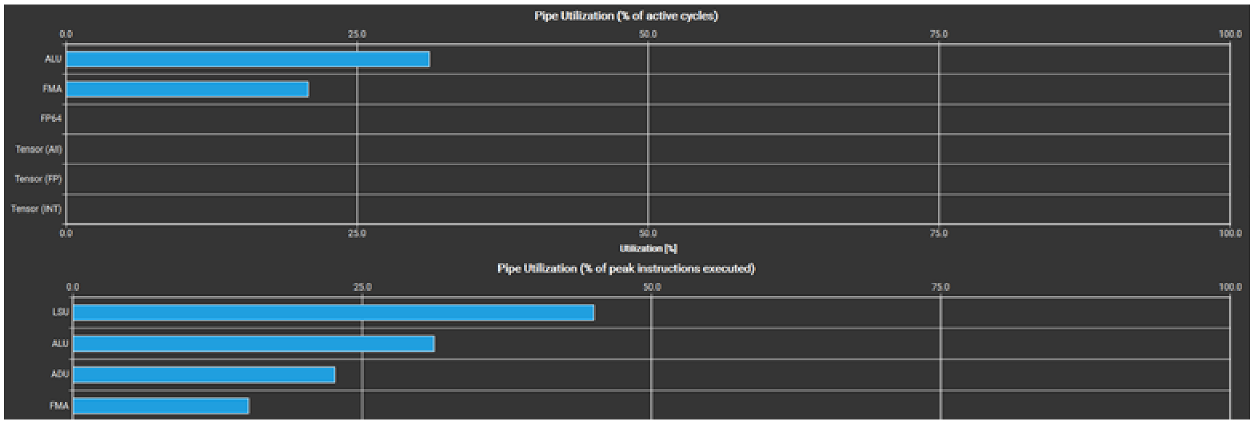


Memory Chart of Baseline Layernorm With Constant Memory

Utilizing constant memory also displayed a change in pipe utilization, as we observed an increase in ALU, a decrease in the LSU utilization, and utilization in the ADU. The reduction in LSU utilization is a result of less global memory access, where the constant memory access goes to the ADU, which means the ADU is now being utilized. Using faster memory access means less stalling for memory access during execution, allowing more arithmetic instructions per cycle, hence the increased utilization of the ALU.



Pipe Utilization Without Constant Memory



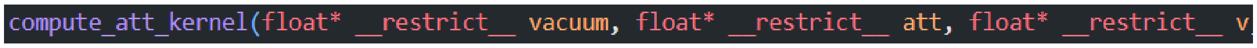
Pipe Utilization With Constant Memory

Overall, using constant memory on the layernorm kernel helped improve its overall runtime performance, as the total runtime went from 7.2 ms to 6.3 ms, displaying a 1.14x speed up.

Restrict (req_7)

Overview

Sometimes, pointers may be aliased, which means the memory regions between these pointers contain overlap. If the compiler cannot be certain that pointers don't alias, then they will be conservative and treat the pointers as if they are aliased. This means that the compiled code will be less optimal because the compiler assumes that changes made to a memory location from one pointer will affect other pointers. To avoid this, the __restrict__ keyword can be used to prevent the compiler from treating pointers as if they are aliased. __restrict__ essentially indicates to the compiler that the pointers are not aliased, meaning memory locations can only be accessed by one pointer. This will enable the compiler to produce more optimal code due to fewer reloads from memory.

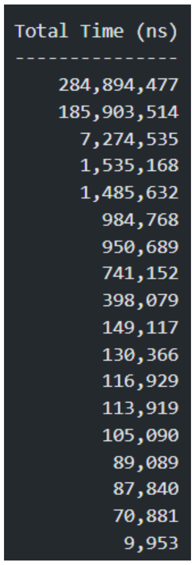


```
compute_att_kernel(float* __restrict__ vacuum, float* __restrict__ att, float* __restrict__ v
```

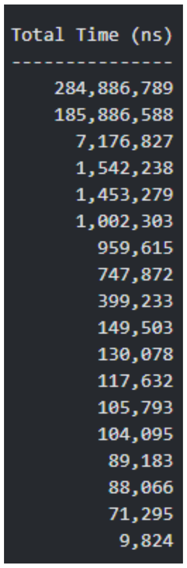
Example of Using __restrict__

Analysis

Comparing the baseline to adding the __restrict__ keyword to method headers, there are no significant changes. Looking at the total time executed before and after adding the keyword, they are almost identical, with no significant time differences.



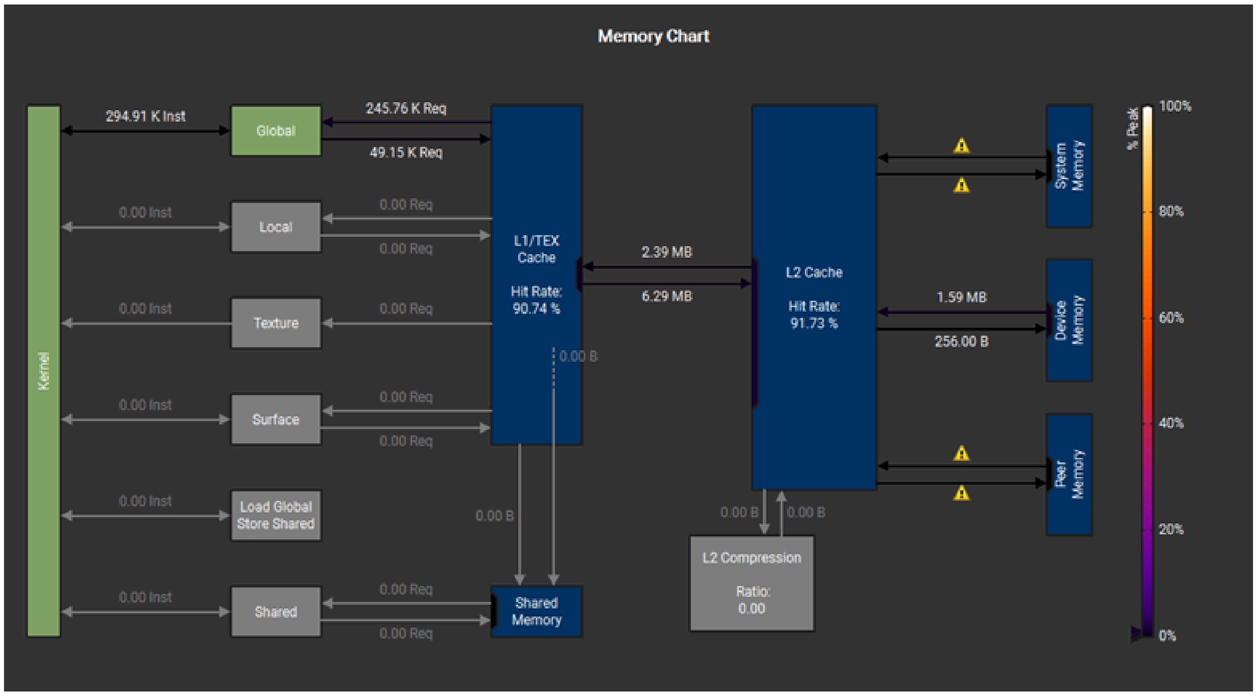
```
Total Time (ns)
---------------
    284,894,477
    185,903,514
      7,274,535
      1,535,168
      1,485,632
        984,768
        950,689
        741,152
        398,079
        149,117
        130,366
        116,929
        113,919
        105,090
         89,089
         87,840
         70,881
          9,953
```



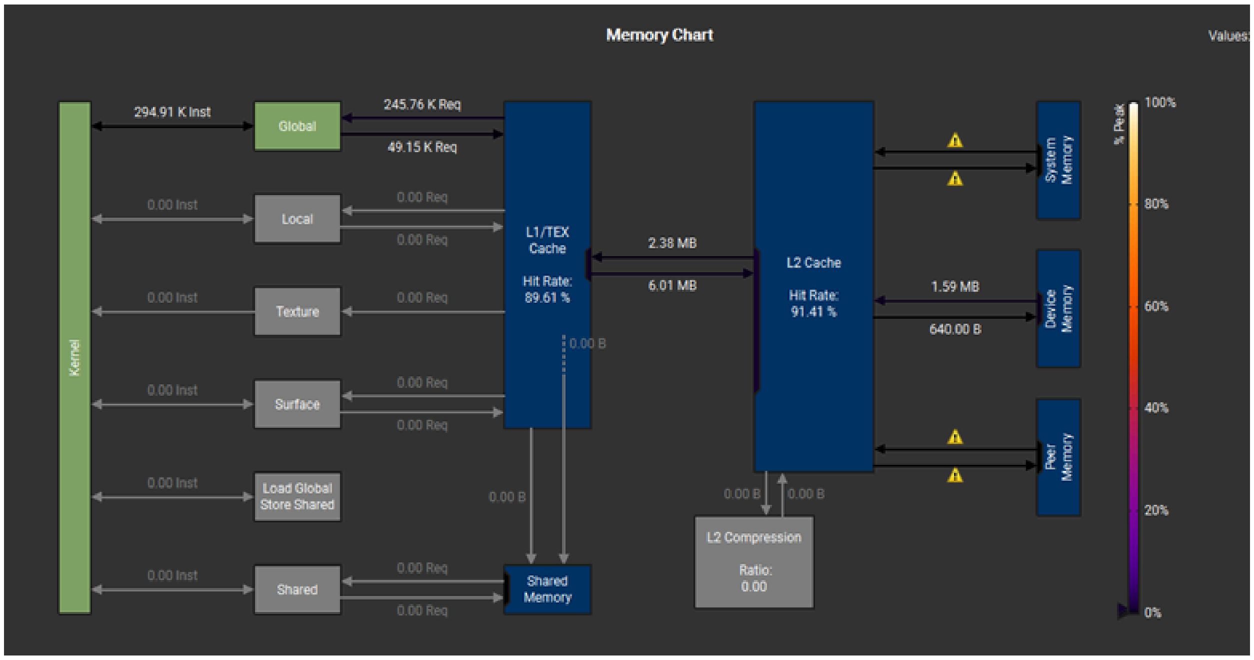
```
Total Time (ns)
---------------
    284,886,789
    185,886,588
      7,176,827
      1,542,238
      1,453,279
      1,002,303
        959,615
        747,872
        399,233
        149,503
        130,078
        117,632
        105,793
        104,095
         89,183
         88,066
         71,295
          9,824
```

Without __restrict__ vs With __restrict__

If we also compare the before and after for the Nsight Compute Report, there are no differences either. For example, in the layernorm kernel, the memory chart is nearly identical with or without the __restrict__ keyword.



Memory Chart Without __restrict__



Memory Chart With __restrict__

The amount of data transferred, cache hit rate, number of memory instructions, and number of memory requests, are identical, meaning that __restrict__ didn't impact NVCC when it was compiling the code to SASS assembly. This is evident as the two version's SASS code compiled by NVCC is the same. The lack of any performance improvement means that the use of __restrict__ is unneeded here. One possible reason for this is the high hit rate for both the L1 and L2 cache. A high hit rate means that most of the accesses to global memory is served from the cache, so any suboptimal code compiled does significantly impact performance because the memory access requests will be serviced directly from the cache.

Reduction (req_4)

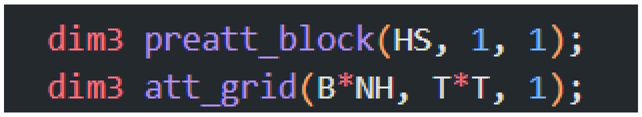
Background

Parallel reduction is an which combines a set of inputs into one value using an operation that's both associative and commutative. Examples of operations reduction can be used to perform are sum, product, min, and max. The purpose of reduction is to reduce computation time by decreasing the number of steps needed to compute the final value and utilize shared memory for faster memory access. Decreasing the number of steps is done by performing the reduction operation on the set of data in parallel instead of sequentially in a loop. In the GPT network, there were several kernels that computed a final value by performing a reduction operation in a for loop, which is inefficient. Below, we'll discuss each of the kernels and why performing a reduction operation can help optimize the kernel.

Kernel Overview

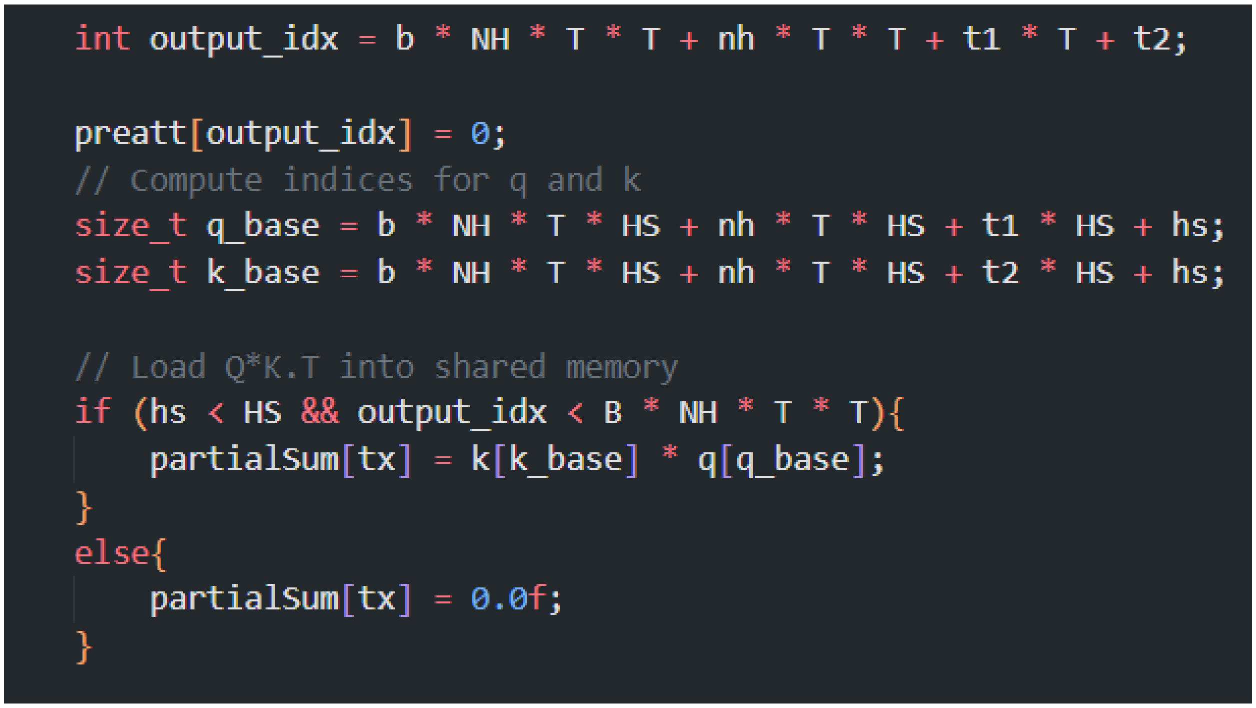
Preattention

The preattention kernel performs Q * K.T for an output matrix of shape (B, NH, T, T), where Q and K are of shape (B, NH, T, HS). For this section, the T dimension of Q will be labeled as T1, and the T dimension of K will be labeled T2. For a given T1 and T2 index, each dot product of Q and K.T over the HS dimension will be summed to produce its final value in the output matrix. In our original code, this accumulation computation was performed in a loop, where each thread would loop over the HS dimension to calculate the output value. This is inefficient, as the accumulation is done sequentially in parallel. To optimize this, reduction was applied in the HS dimension so that the sum can be computed in parallel. In this optimized kernel, each block is configured to perform reduction for a singular output value, and the grid is set up so each index in the output matrix is accounted for.



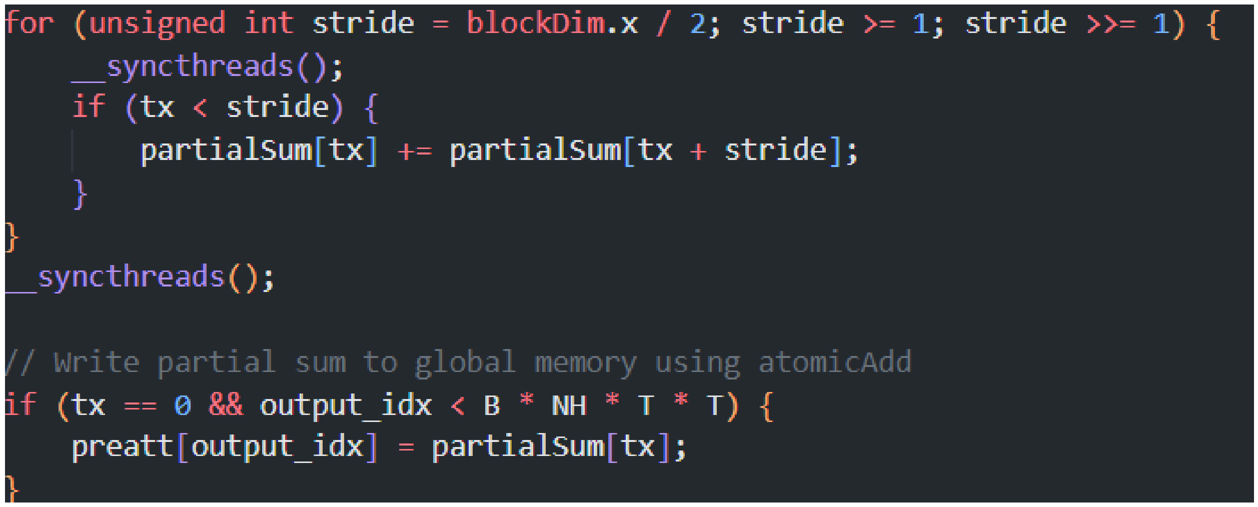
```
dim3 preatt_block(HS, 1, 1);
dim3 att_grid(B*NH, T*T, 1);
```

In the kernel, the block and grid configurations are thus used to determine the hs, b, nh, t1, and t2 indexes. Each thread calculates one Q* K.T value and loads it into shared memory.



```
int output_idx = b * NH * T * T + nh * T * T + t1 * T + t2;

preatt[output_idx] = 0;
// Compute indices for q and k
size_t q_base = b * NH * T * HS + nh * T * HS + t1 * HS + hs;
size_t k_base = b * NH * T * HS + nh * T * HS + t2 * HS + hs;

// Load Q*K.T into shared memory
if (hs < HS && output_idx < B * NH * T * T){
    partialSum[tx] = k[k_base] * q[q_base];
}
else{
    partialSum[tx] = 0.0f;
}
```

Once all values are loaded, the reduction operation is performed. The reduction performed in this kernel is similar to the improved reduction algorithm introduced in the course, where the stride starts at blockDim.x /2 and halfs each time. If a thread's index is less than the stride value, it will sum its corresponding value in shared memory with threadIdx + stride's corresponding value. The final output value, found in index 0 of the shared memory, is then copied into the output tensor.



```
for (unsigned int stride = blockDim.x / 2; stride >= 1; stride >>= 1) {
    __syncthreads();
    if (tx < stride) {
        partialSum[tx] += partialSum[tx + stride];
    }
}
__syncthreads();

// Write partial sum to global memory using atomicAdd
if (tx == 0 && output_idx < B * NH * T * T) {
    preatt[output_idx] = partialSum[tx];
}
```

Compute Attention

The Compute Attention Kernel is very similar to the preattention kernel. It computes att * v for a matrix of dimension (B, NH, T1, HS), where att has the dimension (B, NH, T1, T2) and v has the dimension (B, NH, T2, HS). For a given T1 and HS index, each product of att and v over the T2 dimension will be summed to produce its final value in the output matrix. Like the preattention kernel, each result is summed using a loop, which is inefficient because it's performed sequentially. Instead, reduction should be applied so that the summation operation can be parallelized. The block and grid dimensions for this kernel is set up similarly to the preattention kernel, where each block has T2 threads each block performs reduction for one output value, and the grid is of shape (B*NH, T*HS, 1) to account for all indexes in the output matrix.

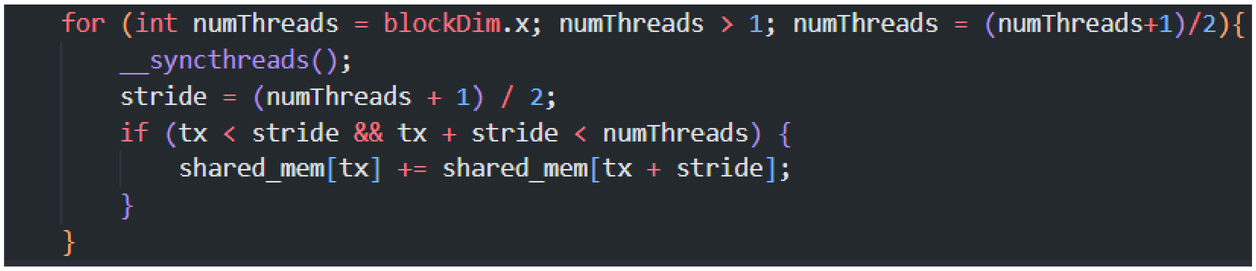
Similarly to the preattention kernel, the block and grid configurations are used to determine its b, nh, t1, hs, and t2 index. Each thread calculates its corresponding att * v value, and loads it into shared memory. Once every thread in the block finishes loading the value into shared memory, a reduction algorithm is applied to sum all the values. This reduction algorithm is the same as the algorithm implemented in the preattention kernel.

Layernorm

The layernorm kernel performs layer normalization for a matrix of dimension (B, T, C) over the c dimension. The calculation is performed as follows:
output = weight * ((input - mean) / sqrt(variance + epsilon)) + bias

In order to find the mean and sqrt for a B, T row in the matrix, the kernel iterates through the C dimension to perform a summation. Because this iteration is sequential, implementing a reduction algorithm would be beneficial to reduce the number of steps to compute the sum over the C dimension. However, unlike the previous two kernels, layernorm must perform reduction multiple times, and the result of the previous reduction is used for the following reduction operations, so it is important to keep the C dimension within one block. This means that for the optimized kernel, the block shape will be (C, 1, 1). To cover all the B, T indexes, the grid dimension will be (T, B, 1).

The indexes of b, t, and c are computed using the block and grid dimensions. To calculate the mean, each thread loads its corresponding input into shared memory. Once all values are loaded, a reduction operation is performed on the input values. The algorithm is similar to the preattention and compute attention kernels, but it also considers that C is not a multiple of 1024, which means additional checks are required to properly perform the reduction. To perform the reduction, the number of active threads and the stride length is stored so the proper boundary checks can be performed. The number of active threads begins at the same value as blockDim.x, and gets halved in each iteration, and stride begins as the half of blockDim.x. When both values are halved, a ceiling division is performed to ensure that no values are skipped when performing the reduction. An additional upper boundary check is performed to ensure that the thread + stride is less than the number of active threads to prevent extraneous additions performed when the number of elements isn't a power of 2.



```
    for (int numThreads = blockDim.x; numThreads > 1; numThreads = (numThreads+1)/2){
        __syncthreads();
        stride = (numThreads + 1) / 2;
        if (tx < stride && tx + stride < numThreads) {
            shared_mem[tx] += shared_mem[tx + stride];
        }
    }
```

The same implementation is done for the second reduction operation to calculate the variance. The only difference is that the value loaded into shared memory is the (input value – mean) ^ 2.

Softmax

The softmax kernel computes the softmax to the preattention scores, converting them into a probability distribution for each T row, where each output value is between 0 and 1, and all values sum to 1. For a given position in a T row in the preattention matrix, the thread only processes elements up to that position, where it will find the max value, apply an exponential function and sum all values up to that position, and normalize the output. To find the max and compute the sum of all exponential values, the kernel iterates from the beginning of the row up to the current position given by the thread index. To optimize this, reduction can be applied so the process can be parallelized. The optimized kernel has a block dimension of (T, 1, 1), where each thread will represent a position in a row of the preattention score. The grid dimension is (T, NH, B) to index into a row for each batch and head.

To calculate the max value, each thread of index up to T loads the corresponding input value into shared memory. Reduction is then performed with the same algorithm as preattention and compute attention, but instead of summing them together, the max value is saved. After the max value is found, the exponential is computed for each value up until index T, and the sum is calculated by performing the same reduction algorithm as the preattention and compute attention kernels. This implementation also eliminates the final loop that normalizes the output for each index up until T, as each thread now handles one value in a row of the preattention matrix.

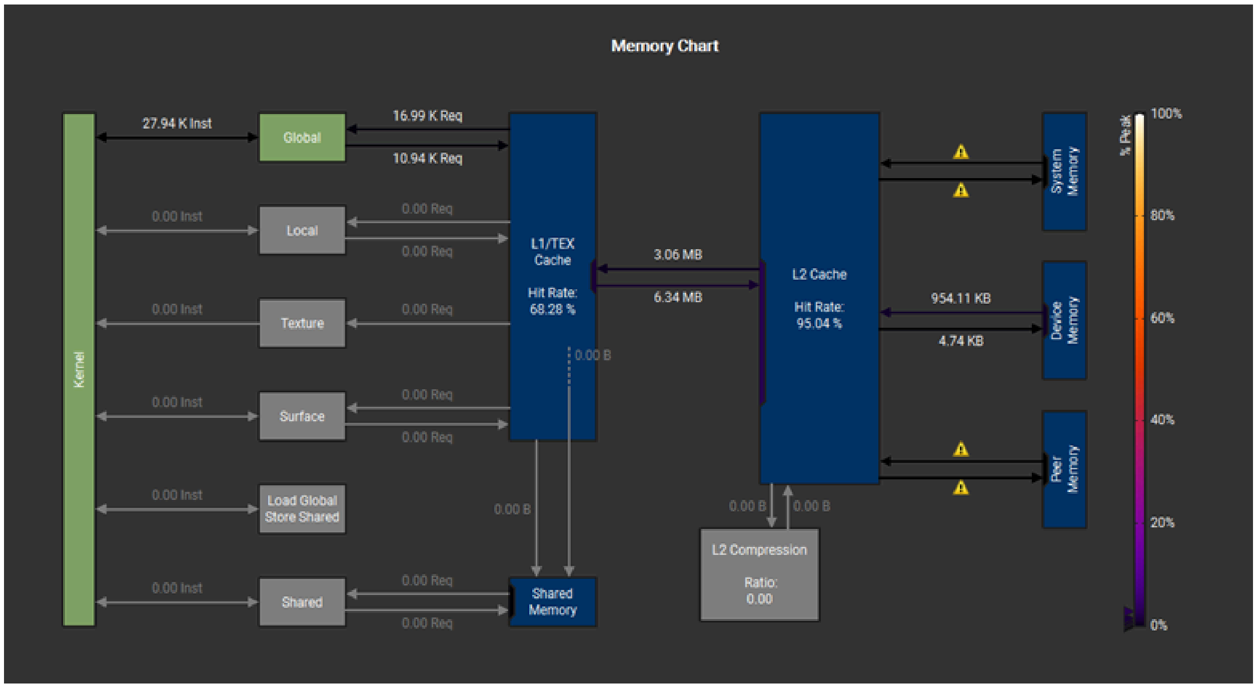
Analysis

When analyzing the Nsight Compute reports of all four kernels, the first major improvement observed was their compute and memory throughputs. For each of their baseline implementations, the kernels saw single digit percentages in their compute and memory throughput, sometimes being below 1%, which signifies low utilization of the GPU's hardware. After implementing reduction across all four kernels, their compute and memory throughput increased to at minimum in the 40% range, and at maximum in the 70% range. Part of this massive improvement is due to having very inefficient baseline kernels that didn't utilize the parallel workload that GPUs provided; for example, preattention and attention had nested loops. However, implementing reduction was certainly beneficial, as the algorithm allowed for threads to concurrently calculate and compute the partial sum/max across a dataset instead of sequentially, which allows for more of the GPUs resources to be utilized. Additionally, reduction's use of shared memory also decreases the global memory traffic. These optimizations allowed for more computations and memory transactions to occur per second, resulting in a higher throughput in both metrics.

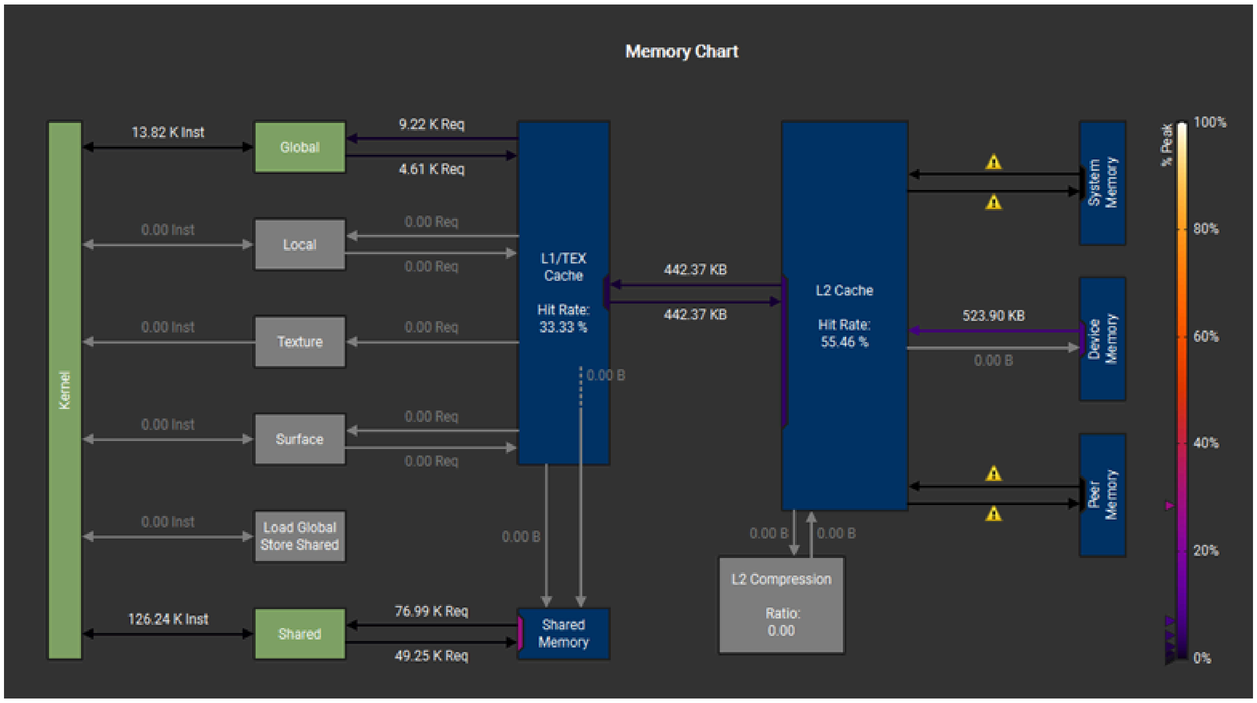
Comparison of Compute and Memory Throughput Before and After Reduction Implementation

Kernel	**Baseline**		**Reduction**	
Throughput	**Compute**	**Memory**	**Compute**	**Memory**
Preattention	0.19%	1.12%	75.43%	75.43%
Compute Attention	0.12%	0.73%	55%	79.71%
Layernorm	1.41%	2.12%	46.41%	46.41%
Softmax	0.38%	4.85%	51.14%	51.14%

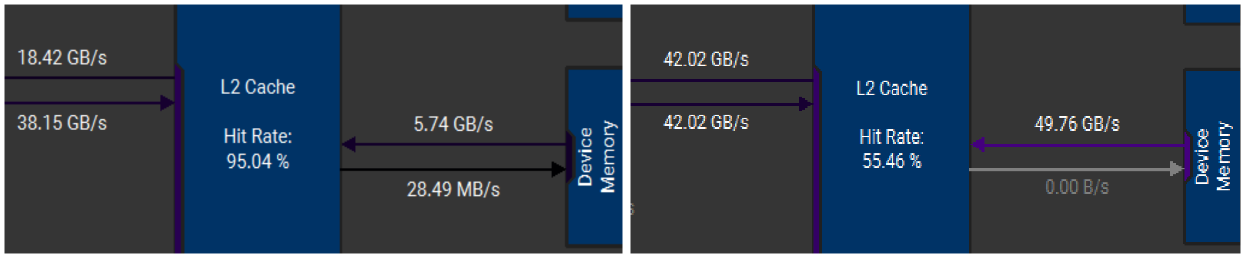
The use of shared memory in reduction also leads to improvements in memory usage. Across all four kernels, the amount of data transfer between the caches and global memory decreases as a lot of memory usage for these kernels is off loaded into the shared memory. For example, the softmax kernel has a reduced amount of global memory instructions and requests when reduction is implemented compared to the baseline. One interesting thing to note is that the hit rate for the L1 and L2 cache decreases when reduction was implemented. This could be due to the use of shared memory, so with fewer global memory accesses, the chances of the data being in the cache decreases, so therefore the cache has to fetch the data from global memory more often. However, the trade off of less cache hits for more shared memory usage is worth it due to the kernel using faster memory. The reduction of global memory use also means the kernel stalls less while accessing global memory, resulting in higher throughput and utilization of GPU's memory.



Memory Chart of Baseline Softmax

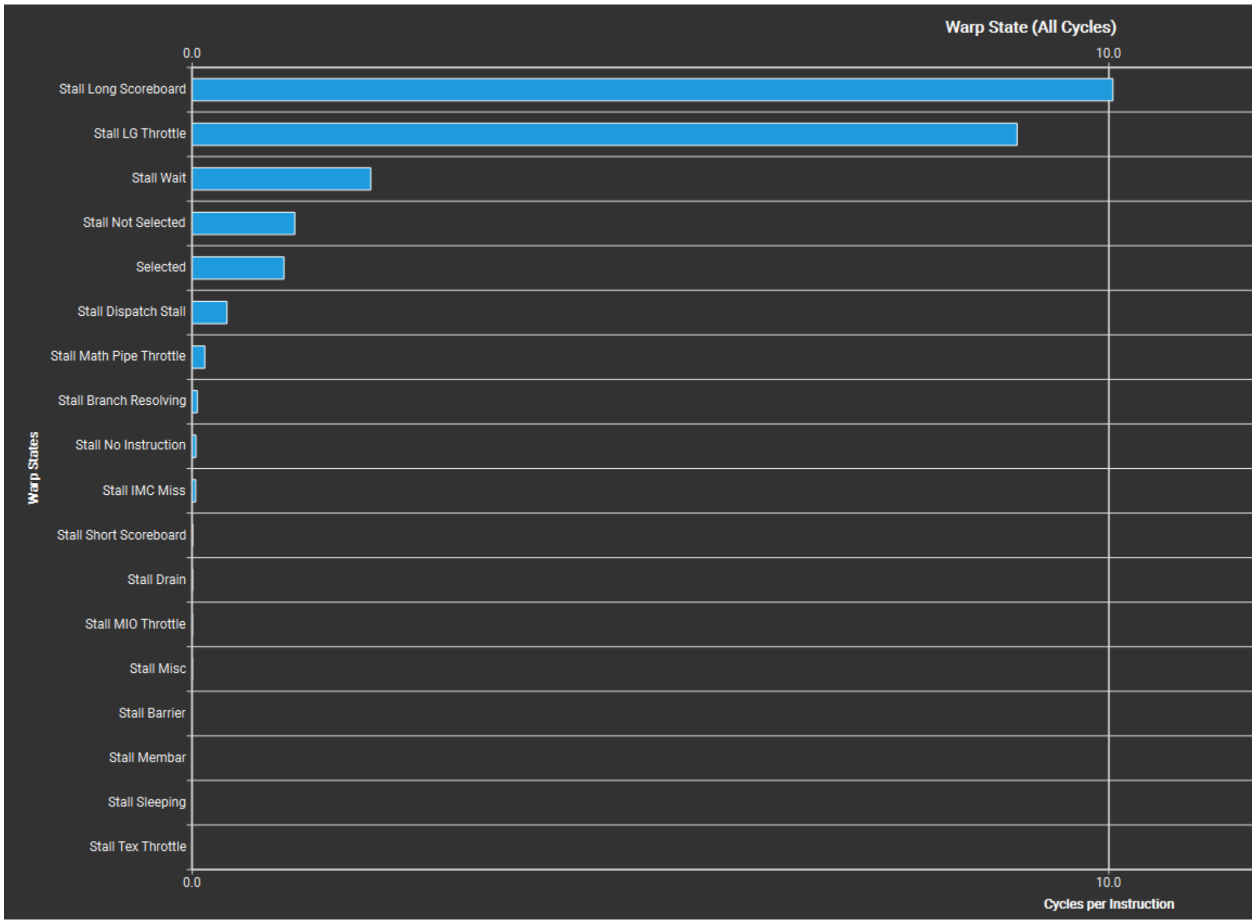


Memory Chart of Reduction Softmax



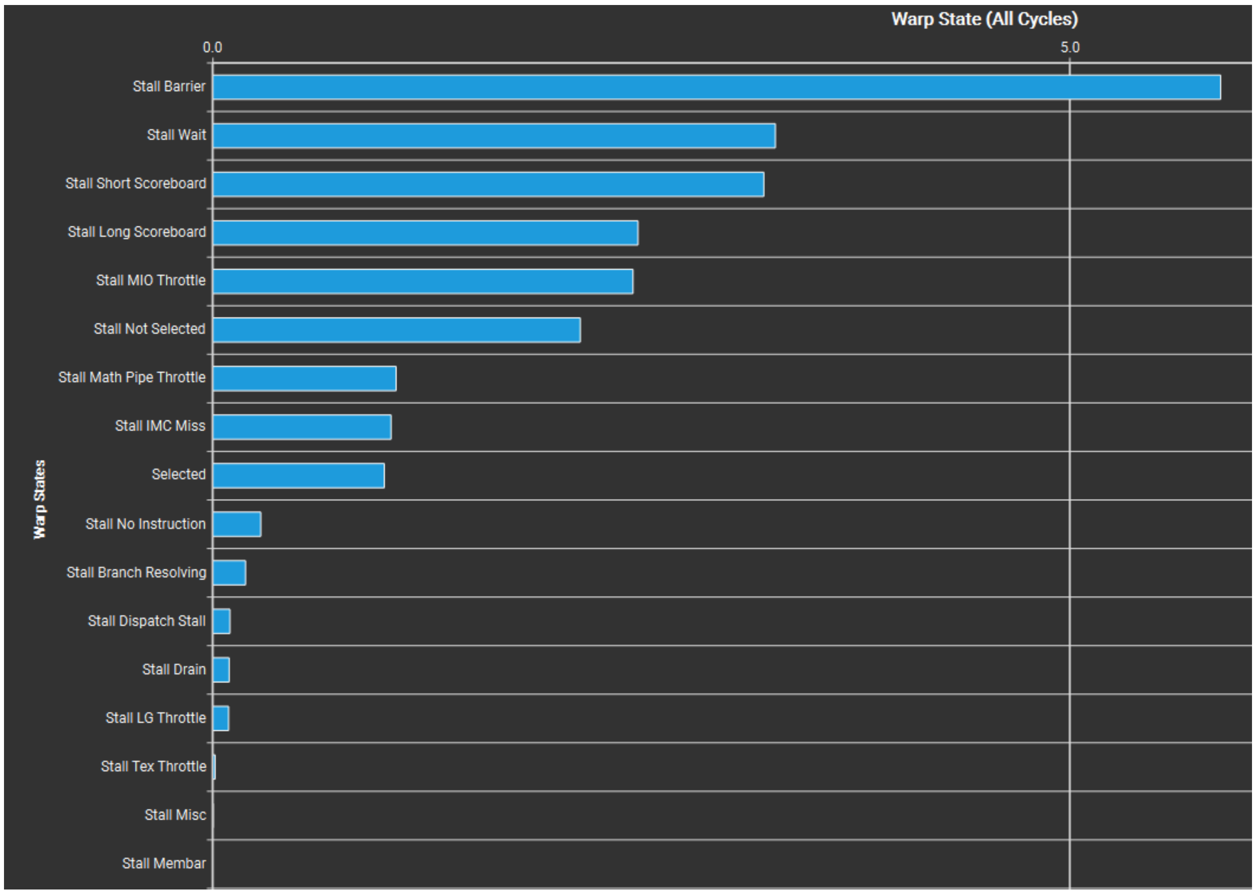
Throughput Differences Without Reduction vs With Reduction

The reduction of global memory use is reflected in the warp states. Looking at the Warp State chart, the baseline implementations of all four kernels experience a lot of stalling in certain areas. In the preattention and compute attention kernels, they experience a very high long scoreboard stall, which is when a warp is waiting on data from global memory. The reason for this stall in the preattention and compute attention kernel is due to constantly having to access global memory to load data to compute the final result. The other two kernels, Layernorm and Softmax, in addition to having a high long scoreboard stall, also had a long lg_throttle stall, which is when the warp has to wait for the L1 instruction queue to be not full. The lg_throttle stall is due to redundant global memory accesses occurring within the kernel, which is what is happening for both layernorm and softmax, as both access the same index from input multiple times.



Layernorm Stall Statistics

By implementing reduction on these kernels and using shared memory to store the values that will be used for reduction, the kernels show no large stalling in any category. There is still stalling occurring in several categories, but the stalls are typically no more than 5 cycles per instruction, demonstrating improved performance from the baseline implementation.



However, all four reduction kernels are still experiencing thread divergence, which makes sense, because as the stride length decreases for each reduction iteration, eventually it either not be a multiple of 32, which is the warp size, or it will decrease to less than the size of the warp, which introduces thread divergence.

Overall, due to the increased throughput, the use of faster memory, and the reduction in stall cycles from implementing reduction in these four kernels, the runtime for each had drastically increased.

Total Execution Time Per Kernel (µs)

Kernels	**Preattention**	**Compute Attention**	**Layernorm**	**Softmax**
Baseline	185903 µs	284894 µs	7274 µs	1485 µs
Reduction	2134 µs	3180 µs	355 µs	99 µs
Speed Improvement	87	89	20	15

Overview of Self-Attention Layer

Mathematical Operation: $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_w}})V$

Background

The query matrix encodes the relationship the model is searching for between the current token to other tokens. The key matrix holds the information of all the previous tokens in the sequence. The key matrix is given an upper triangular mask, which is needed so that the query matrix only looks at past data in sequence, not future data, which would defeat the purpose of GPT-2 being a predictive model. The multiplication QK^T can be thought of as the weights of the relationships- the extent to which the past tokens match the relationships queried. This is scaled by $\frac{1}{\sqrt{d_w}}$, which helps stabilize the matrix product. d_w is the attention head size, which corresponds to the dimensionality of the Q,K,V matrices. The larger the dimensionality leads to very large variance, which results in unstable softmax output. By scaling with $\frac{1}{\sqrt{d_w}}$, this variance is reduced resulting in a smooth and stable softmax output. The softmax function turns the input into a probability distribution, encoding the relationships that have the highest likelihood of being present. The value matrix can be interpreted as the 'values' of the relationships. The overall attention output is the product between the softmax output and the value matrix, which can be interpreted as the direction that the input embedding should be modified in, to account for the enhanced meaning given by the relationships.

Brief Review of Problems with Baseline

Our baseline attention involved 5 kernels:

Permute- obtained the QKV matrices with correct dimensions (total time: 123,969 ns)

Preatt- computed the QK preattention scores (total time: 185,912,317 ns)

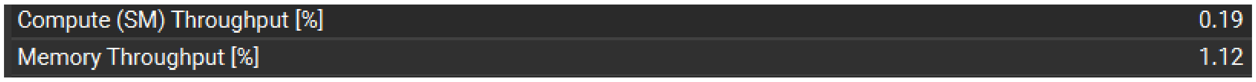
Softmax- Scaled the preattention scores and apply safe softmax (subtract the max row value before applying scaling for stability) to each row (total time: 1,483,140)

Compute_att- compute final scores by multiplying with V matrix (total time: 284,997,523)

Unpermute- Permute output to correct dimension, flip NH and T dimensions (total time: 70,978)

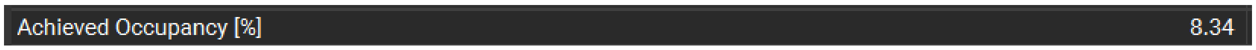
Since the preatt and compute_att kernels dominated the runtime, our initial analysis focused on them. Below, I outline our logic for determining the root cause behind why our initial preatt kernel was so slow, which turned out to be the same issue plaguing compute_att.

The first red flag our team noticed, which was mentioned in the reduction section, was unusually low throughput:



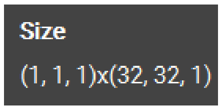
Compute (SM) Throughput [%]	0.19
Memory Throughput [%]	1.12

Which was due to very low achieved occupancy:



Achieved Occupancy [%]	8.34

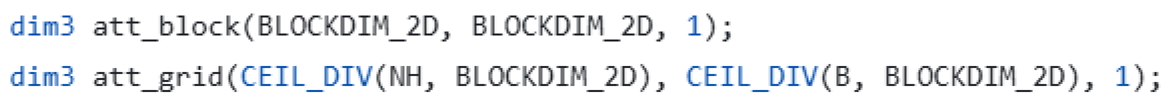
Due to the fact that we only launched one block in the entire grid:



Size

(1, 1, 1)x(32, 32, 1)

This was because we **severely overestimated** the parallelism that could be achieved over the B (batch) and NH (number of heads) dimension. Our original grid was



```
dim3 att_block(BLOCKDIM_2D, BLOCKDIM_2D, 1);
dim3 att_grid(CEIL_DIV(NH, BLOCKDIM_2D), CEIL_DIV(B, BLOCKDIM_2D), 1);
```

which only resulted in one block because the forward pass of GPT2 used 4 batches and 12 heads, while our chosen block dimension was 32x32. Doing more research, our team realized that, especially for inferencing, batch size was normally small, in the order of 10s, or even single digits. Additionally, we looked through open source models on Hugging Face and determined that even modern llms used at most a couple hundred heads. Given this information, our group decided that to better utilize the A40’s computational resources, it would make sense to parallelize over more dimensions like sequence length, which we implemented in the Local and Flash Attention optimizations.

Another issue we determined was that within the SMs that were running, warps seemed to be stalling. Nsight Compute listed the cause as *long scoreboard*: on average, each warp of preattention spent 14 cycles being stalled, likely due to waiting for a memory read. One of the reasons this may have been happening is because while the preatt and compute attentions are matrix multiplications that access the same values multiple times, we did not utilize any form of memory reuse such as shared memory, instead making every memory access global. Global reads are slower than shared reads, and too many concurrent global reads was likely what caused the warps to stall.

The key takeaways we had were that parallelization over batch and number of heads is insufficient to fully utilize the A40, and in attention, some form of tiling is needed to avoid warp stalling from waiting on global reads.

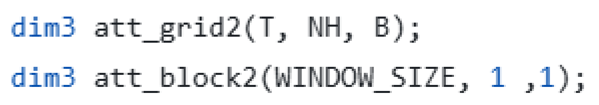
Local Attention (req _9)

Background

A key drawback of attention is that it scales quadratically with sequence length. An easy way to see why is to look at the QK matrix, which is sequence length by sequence length. If the sequence length is doubled, the number of values to be computed in the QK matrix will be quadrupled. This means that both computations and intermediate storage needed can grow unsustainably large for large context windows. This bottlenecks the performance of LLMs during a lengthy conversation where context which was thousands of tokens earlier could be relevant for an optimal response. The Longformer Paper (2020) found a solution to this problem through combining a local windowed attention with a sparse global attention. Windowed attention only computes the attention scores for a token with the past [window_size] tokens. Visually, this would mean that every row i of the QK matrix would be a strip of computed values from max(0, i - window_size) to i. This maintains a significant portion of performance because more recent tokens are more likely to have a greater impact on the current token than older ones. To account for other cases, a sparse global attention is used on several preselected input tokens. This approach helps stop long context models from degrading over time without excessive compute requirements. In this requirement, we just chose to implement local windowed attention for GPT2. GPT2 has a max context window of 1024 (HuggingFace), so it is a relatively short model, and there would be a minimal advantage in implementing a complementary sparse global attention.

Kernel Overview

Learning from our baseline implementation, we parallelized not only over batch and number of heads, but sequence length. We also realized that since local attention is computing small strips for each row, head, and batch, every block could compute this strip (window size is expected to be less than 1024). The strip is up to window_size in length, so every thread can compute every element in the strip. Local attention makes it possible for this 4th degree of parallelism, since for a large QK matrix threads would have to compute more than one element in a row since the row size would likely be significantly larger than 1024. Since every block computes a strip, our group determined that we could also combine the softmax kernel with preattention and compute_attention kernels, since all of these computations can be serially done at the block granularity. This is how we determined the launch configuration to be:



```
dim3 att_grid2(T, NH, B);
dim3 att_block2(WINDOW_SIZE, 1 ,1);
```

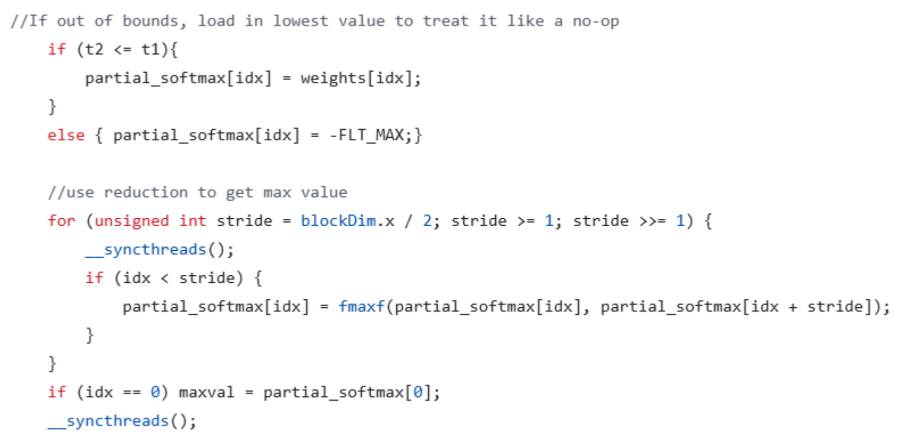
Each block is assigned to its respective token, head, and batch, with each token being assigned to position within the strip to compute. The size of the strip is limited by the position of the current token- if it’s near the start of the row, less than window_size tokens will be computed. All threads with an idx less than t1-ts will not compute a value. This hints at potential branch divergence, though it will only be problematic near edges of large matrices.

```
__global__ void local_attention_forward(float* vacuum, float inv_temperature, float* q, float* k, float* v, int B, int NH, int T, int HS) {
    int t1 = blockIdx.x;
    int nh = blockIdx.y;
    int b = blockIdx.z;
    int idx = threadIdx.x;

    //window start
    int ts = max(0, t1 - WINDOW_SIZE);

    int t2 = idx + ts;
```

Since the same strip will be reused multiple times, it makes sense to use shared memory to hold the strip of “weights”, to reduce global loads and stores. This weights strip is loaded with the QK values, obtained from the dot product of the relevant token vectors. For the softmax computation, reduction is necessary, since the max value of the strip and the sum of exponentials must be obtained. A sum reduction is a textbook example, while max reduction is also possible because max(a,b) is a binary operation, associative and commutative, and has an identity element (min_val). The shared partial_softmax array was declared as a temporary array for both reductions, while the max and sum values were also declared shared for each thread to reference when computing the final softmax value.

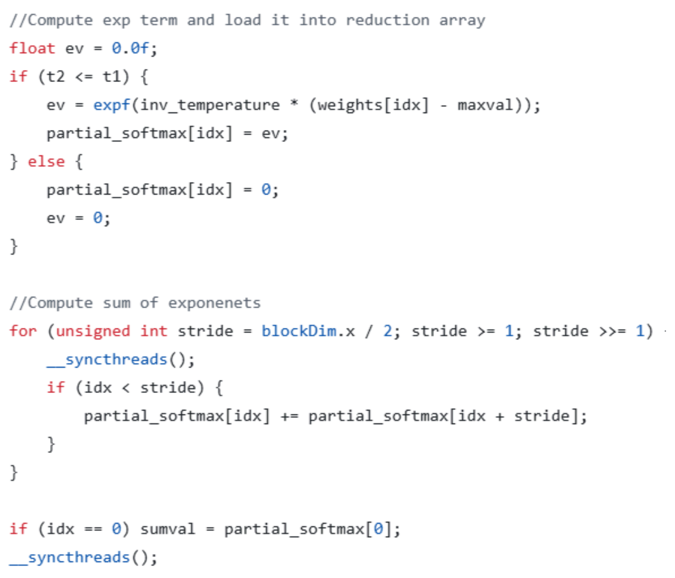
```
__shared__ float weights[WINDOW_SIZE];

//Load weights array from start to end with the KV matrix
if (b < B && nh < NH && t1 < T && t2 <= t1){
    float sum = 0.0f;
    for (int hs = 0; hs < HS; hs++) {
        sum += k[b * NH * T * HS + nh * T * HS + t2 * HS + hs] *
            q[b * NH * T * HS + nh * T * HS + t1 * HS + hs];
    }
    weights[idx] = sum;
}

//Partial softmax computation
__shared__ float partial_softmax[WINDOW_SIZE];
__shared__ float maxval;
__shared__ float sumval;
```

Below is the max reduction, following the improved version in lecture to minimize divergence.



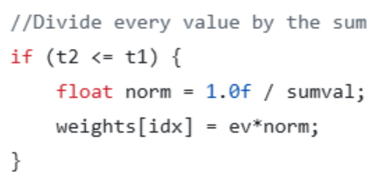
```
//If out of bounds, load in lowest value to treat it like a no-op
    if (t2 <= t1){
        partial_softmax[idx] = weights[idx];
    }
    else { partial_softmax[idx] = -FLT_MAX;}

    //use reduction to get max value
    for (unsigned int stride = blockDim.x / 2; stride >= 1; stride >>= 1) {
        __syncthreads();
        if (idx < stride) {
            partial_softmax[idx] = fmaxf(partial_softmax[idx], partial_softmax[idx + stride]);
        }
    }
    if (idx == 0) maxval = partial_softmax[0];
    __syncthreads();
```

Following this is the sum reduction, done in a similar format. Each thread also keeps a copy of their respective exponential term at the register level variable ev.



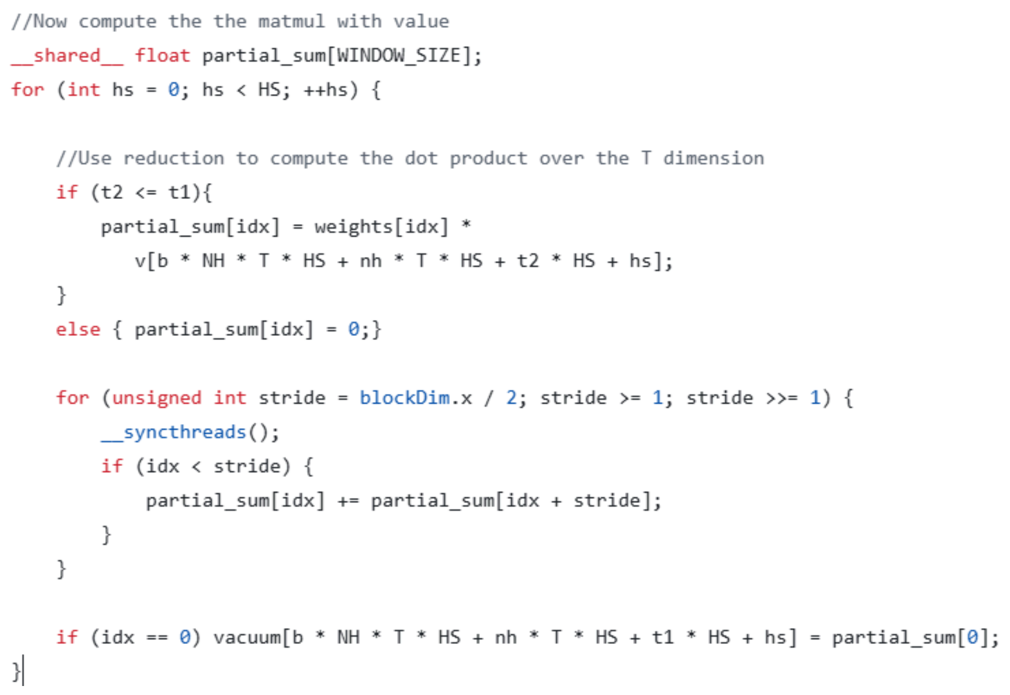
```
//Compute exp term and load it into reduction array
float ev = 0.0f;
if (t2 <= t1) {
    ev = expf(inv_temperature * (weights[idx] - maxval));
    partial_softmax[idx] = ev;
} else {
    partial_softmax[idx] = 0;
    ev = 0;
}

//Compute sum of exponenets
for (unsigned int stride = blockDim.x / 2; stride >= 1; stride >>= 1)
    __syncthreads();
    if (idx < stride) {
        partial_softmax[idx] += partial_softmax[idx + stride];
    }
}

if (idx == 0) sumval = partial_softmax[0];
__syncthreads();
```

Then, the final softmax output is computed by each active thread.



```
//Divide every value by the sum
if (t2 <= t1) {
    float norm = 1.0f / sumval;
    weights[idx] = ev*norm;
}
```

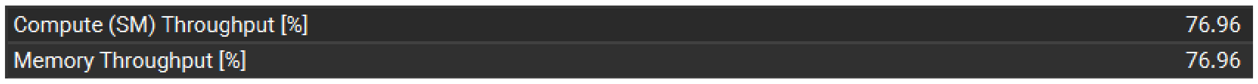
Finally, the final attention score strip that corresponds to the current block is computed. Since the sum in this multiplication is being done over the token dimension, a third reduction will be needed to compute this sum, using a new shared array called partial_sum.



```
//Now compute the the matmul with value
__shared__ float partial_sum[WINDOW_SIZE];
for (int hs = 0; hs < HS; ++hs) {

    //Use reduction to compute the dot product over the T dimension
    if (t2 <= t1){
        partial_sum[idx] = weights[idx] *
           v[b * NH * T * HS + nh * T * HS + t2 * HS + hs];
    }
    else { partial_sum[idx] = 0;}

    for (unsigned int stride = blockDim.x / 2; stride >= 1; stride >>= 1) {
        __syncthreads();
        if (idx < stride) {
            partial_sum[idx] += partial_sum[idx + stride];
        }
    }

    if (idx == 0) vacuum[b * NH * T * HS + nh * T * HS + t1 * HS + hs] = partial_sum[0];
}
```

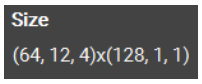
Analysis

The total runtime of the local attention kernel was 4,443,121 ns, which compared to the baseline, which included preattention, softmax, and compute_att, was a **106.32x** speedup. While the name of this optimization is local attention, this improvement cannot uniquely be attributed to windowing, but also to the fusion of kernels and increased parallelism. A key difference between the baseline implementation and local attention is the overall Speed of Light throughput being an orders of magnitudes higher:



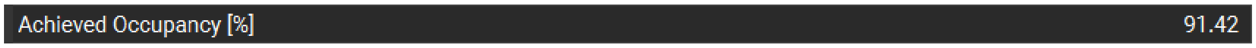
Compute (SM) Throughput [%]	76.96
Memory Throughput [%]	76.96

Memory throughput increased by more than 50x and compute throughput by over 400x (specifically compared to preattention, which was the lowest throughput of the baseline). This is because SMs are able to properly utilize warps due to a larger grid of size:



Size
(64, 12, 4)x(128, 1, 1)

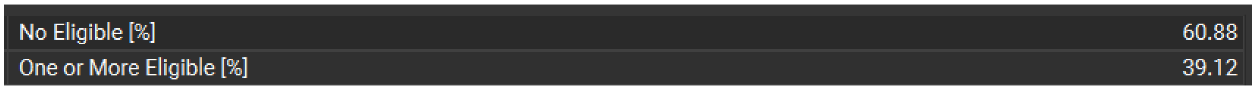
which increased achieved occupancy from 8.34% in the baseline to above 90%!



Achieved Occupancy [%]	91.42

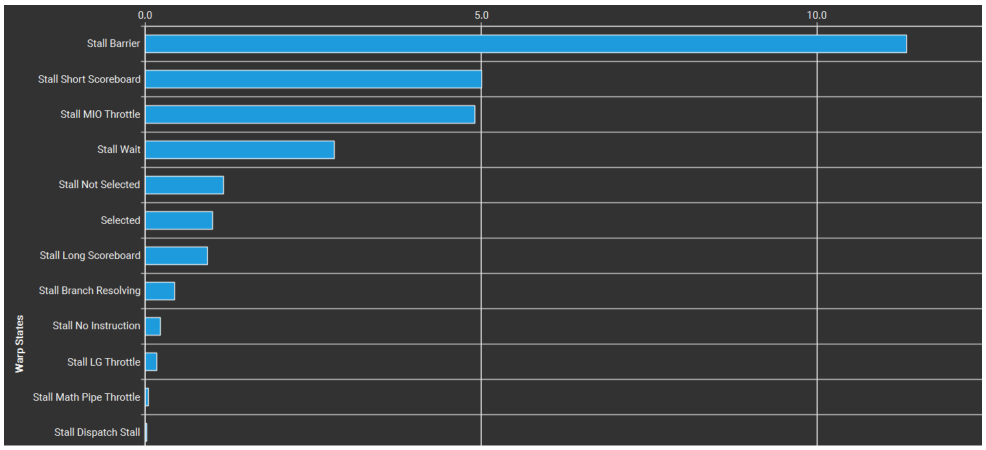
The Nsight Compute details page explains that there seem to be no noticeable memory or compute bottlenecks.

Our group still found noticeable performance issues to improve through the analysis, as there was still a significant amount of warp stalling, with 60% of cycles having no eligible warps, so the scheduler which is capable of issuing an instruction every cycle only issued one every 2.6 cycles.



No Eligible [%]	60.88
One or More Eligible [%]	39.12

According to Nsight Compute, this due to synchronization barriers causing threads in a block to spend on average 11.3 idle cycles waiting for slower threads to catch up. This is likely due to the number of __synchthreads statements in the kernel, for each iteration of the three reductions. Further performance improvements would come from reducing the number of barrier statements, possibly doing multiple warp level reductions in a block. Overall though, this is still an improvement from the previous long scoreboard stalling which led to 14 stall cycles.



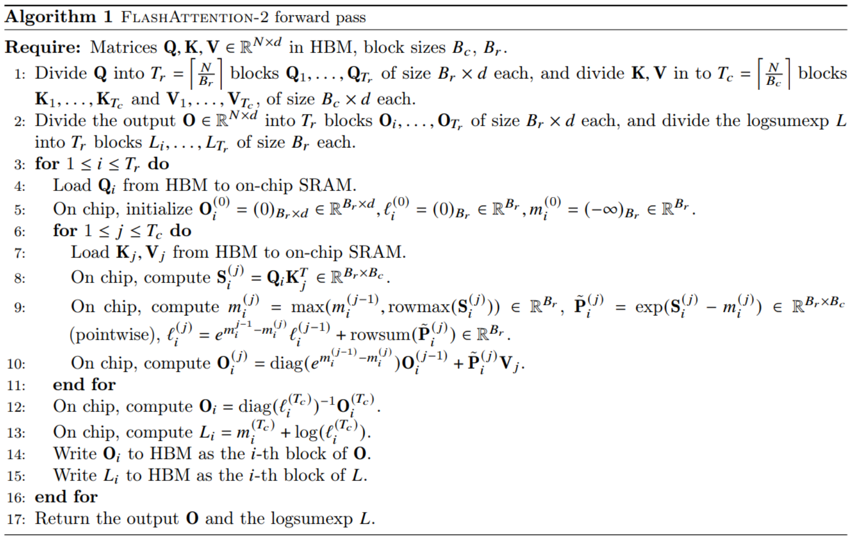
Another possible reason for this that Nsight Compute uncovered, which was hinted at in the kernel overview, was branch divergence, which lowered the average of 31.5 threads per warp active in a cycle to 19.8. There are two reasons why this could be happening. First, the window is being truncated for early tokens, which causes threads in a warp that correspond to an index below zero to stay inactive. The other reason is due to a fundamental flaw with our reduction approach. In textbook reduction, each thread loads two elements of the input array, and stays active for the first iteration. However, since the number of threads is the same as the number of elements in the input array due to how our launch configuration was designed, each thread loads a single thread, which means that even in the very first reduction step half of threads are inactive. This could be alleviated by launching blocks of size window_size/2 and having every thread compute two elements of the strip.

The key takeaways from this optimization were that parallelization over the sequence length is a must for low-latency attention kernels and to be careful with the implementation of reduction for small arrays, as barrier synchronization and branch divergence may actually make the reduction slower than a naive approach.

Flash Attention (req_3)

Background

Flash attention tries to tackle the same problem as local windowed attention: how to make attention more scalable with longer sequence lengths (quadratic to linear scaling). In attention, latency is normally limited by memory bandwidth, as seen in the baseline analysis. This is because, according to the original Flash Attention paper, GPU compute power has been outpacing bandwidth speeds. The core idea of flash attention is to maximize reuse of data to minimize the number of memory read/writes. Even though more floating point operations are computed, since attention is memory limited, there is still a significant speedup. Specifically, our implementation is based off of the Flash Attention-2 algorithm proposed by Tri Dao.



Algorithm 1 FlashAttention-2 forward pass

```
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM, block sizes $B_c$, $B_r$.
1: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
2: Divide the output $\mathbf{O} \in \mathbb{R}^{N\times d}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp $L$ into $T_r$ blocks $L_i, \dots, L_{T_r}$ of size $B_r$ each.
3: for $1 \le i \le T_r$ do
4:   Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.
5:   On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}, \ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}, m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.
6:   for $1 \le j \le T_c$ do
7:     Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8:     On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
9:     On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \mathrm{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{j-1} - m_i^{(j)}} \ell_i^{(j-1)} + \mathrm{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.
10:    On chip, compute $\mathbf{O}_i^{(j)} = \mathrm{diag}(e^{m_i^{(j-1)} - m_i^{(j)}}) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.
11:  end for
12:  On chip, compute $\mathbf{O}_i = \mathrm{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$.
13:  On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.
14:  Write $\mathbf{O}_i$ to HBM as the $i$-th block of $\mathbf{O}$.
15:  Write $L_i$ to HBM as the $i$-th block of $L$.
16: end for
17: Return the output $\mathbf{O}$ and the logsumexp $L$.
```

Here's a breakdown of how the forward pass works. Steps 1 and 2 define the size of the shared memory blocks to be used in each block of the kernel. Br is the height of the Q and output tiles. Bc is the height of the K and V tiles. All 4 tiles are length d (head size). Br and Bc are parameters that are maximized by the amount of shared memory available in a single grid block. It is important to note that the logsumexp matrix doesn't need to be kept for a forward pass- it is used in the backward pass to calculate the gradient. Tr is defined as T/Br, or the total number of Q and O blocks needed to iterate over the entire TxD matrices. Similarly, Tc is defined as T/Bc.

The next step of the algorithm is to iterate through each of the Q and O blocks from 1 to Tr. In each iteration, a new shared tile Br x d tile of Q should be loaded in. This tile contains all the information from Q that is needed to compute a Br x d output tile. Then, the output tile is initialized to zero, the 1d tile L holding the sum of exponentials for each row is initialized to 0, and the 1d tile holding the running max in each row is initialized to -inf. This is necessary to compute the partial softmax for each output row, and update past results if necessary.

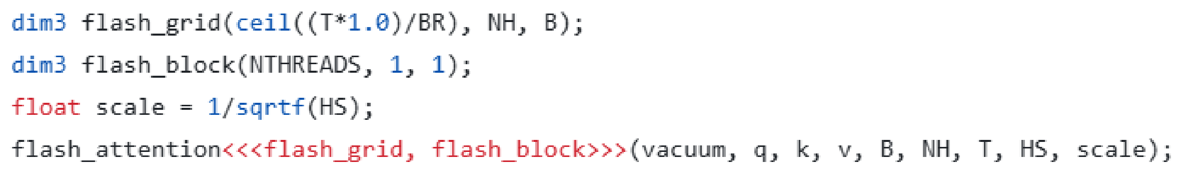
To obtain an output tile of Txd, the input Q tile needs the entire K and V matrices for the 2 matrix multiplications in attention. Since loading the entire matrices into shared memory is not feasible, an inner loop iterates over all the KV tiles of size Bc x d from 1 to Tc. In each iteration, the respective tiles are loaded into shared memory. Now, it is possible to compute the preattention scores of size Br x Bc through multiplying the Q tile and transpose of the K tile. These scores are stored in another shared buffer called S. After this, the max of each row of length Bc is computed, and compared to the running max. Then, the safe softmax numerator is computed and stored in the buffer P. The running sum of exponentials (softmax denominator) array is also updated. Each row of P is summed and added to the past array, which is multiplied by exp(past_max-current_max), which updates the denominator terms to match the current max. Then the old output tile is multiplied by the same scaling factor, which updates the numerator term to match the current max, and added with the matrix product of P and V, which is the contribution of this iteration to the output tile.

After the final inner loop iteration, the full softmax denominator has been computed. Every term in the output tile is divided by this value. It is fine to do this after the multiplication with V because multiplying by a scalar before or after a matrix multiplication will not alter the product since they are associative operations. This is the complete output tile which can finally be loaded back to global memory. After all Tc chunks are done, the attention computation is finished. This algorithm takes advantage of **online softmax**, which is the idea that instead of computing softmax for an entire row at a time, it can be dynamically computed and updated for smaller chunks.

Now we will walk through how we used this algorithm as a blueprint for our kernel.

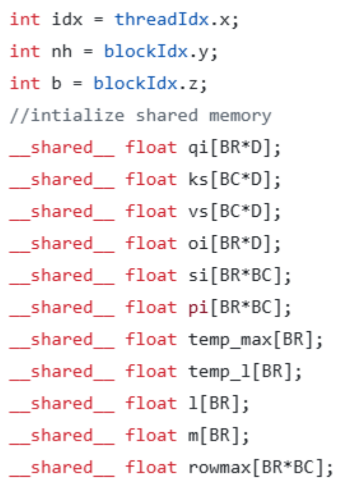
Kernel Overview

The algorithm earlier is for a single attention head and batch. To parallelize over them, the simple solution was to have two of our grid dimensions encode the number of batches and heads. Our x-dimension of the grid encoded the Tr chunks, parallelizing the outer loop of the algorithm. The inner loop is inherently sequential due to the nature of online softmax so it could not be part of the grid. The blocks contained NTHREADS threads, a macro that was set to 256. This is because we determined that the largest values of BR and BC that a block could handle was 16 x 16, and since the inner loop was computing BR x BC tiles, including more than 256 threads would leave a portion mostly inactive.



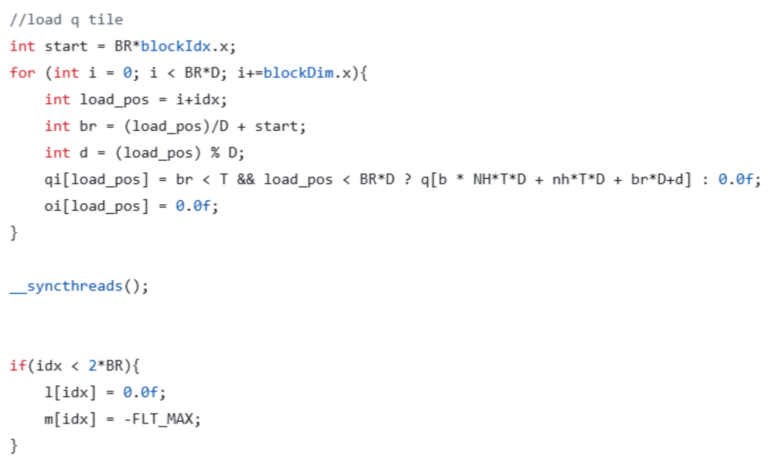
```
dim3 flash_grid(ceil((T*1.0)/BR), NH, B);
dim3 flash_block(NTHREADS, 1, 1);
float scale = 1/sqrtf(HS);
flash_attention<<<flash_grid, flash_block>>>(vacuum, q, k, v, B, NH, T, HS, scale);
```

The idx variable corresponds to the relative position of the thread, which was used to determine the relative tile loading position. The rest of the buffers are similar to what the algorithm uses. We created an additional temp max and temp l buffer to hold the max and exponential sum for the current inner iteration, and used rowmax as a temporary buffer for reduction.



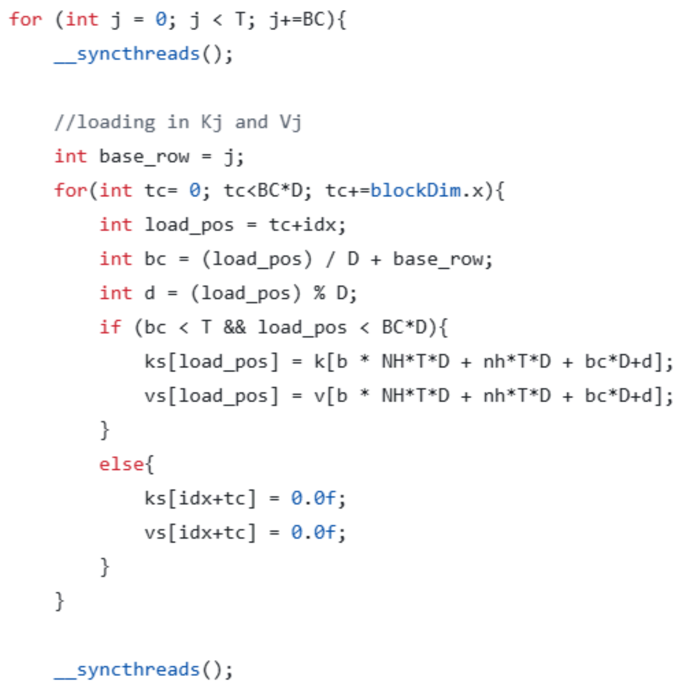
```
int idx = threadIdx.x;
int nh = blockIdx.y;
int b = blockIdx.z;
//intialize shared memory
__shared__ float qi[BR*D];
__shared__ float ks[BC*D];
__shared__ float vs[BC*D];
__shared__ float oi[BR*D];
__shared__ float si[BR*BC];
__shared__ float pi[BR*BC];
__shared__ float temp_max[BR];
__shared__ float temp_l[BR];
__shared__ float l[BR];
__shared__ float m[BR];
__shared__ float rowmax[BR*BC];
```

Next, the q tile is loaded into the shared buffer, with the output tile also being set to 0. Basic bounds checks are done to prevent out of bounds accesses. The current block in the x dimension determines where tile starts. The running max and sum of the exponential array is also initialized for each row.



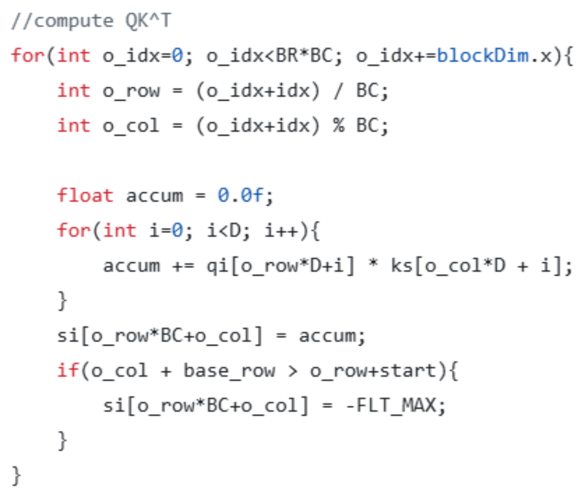
```
//load q tile
int start = BR*blockIdx.x;
for (int i = 0; i < BR*D; i+=blockDim.x){
    int load_pos = i+idx;
    int br = (load_pos)/D + start;
    int d = (load_pos) % D;
    qi[load_pos] = br < T && load_pos < BR*D ? q[b * NH*T*D + nh*T*D + br*D+d] : 0.0f;
    oi[load_pos] = 0.0f;
}

__syncthreads();


if(idx < 2*BR){
    l[idx] = 0.0f;
    m[idx] = -FLT_MAX;
}
```

Then, the inner loop over the KV chunks can begin. The first step is similar to the loading of the Q tile. The start of each chunk is determined by the jth iteration, which corresponds to the starting row of the tile.



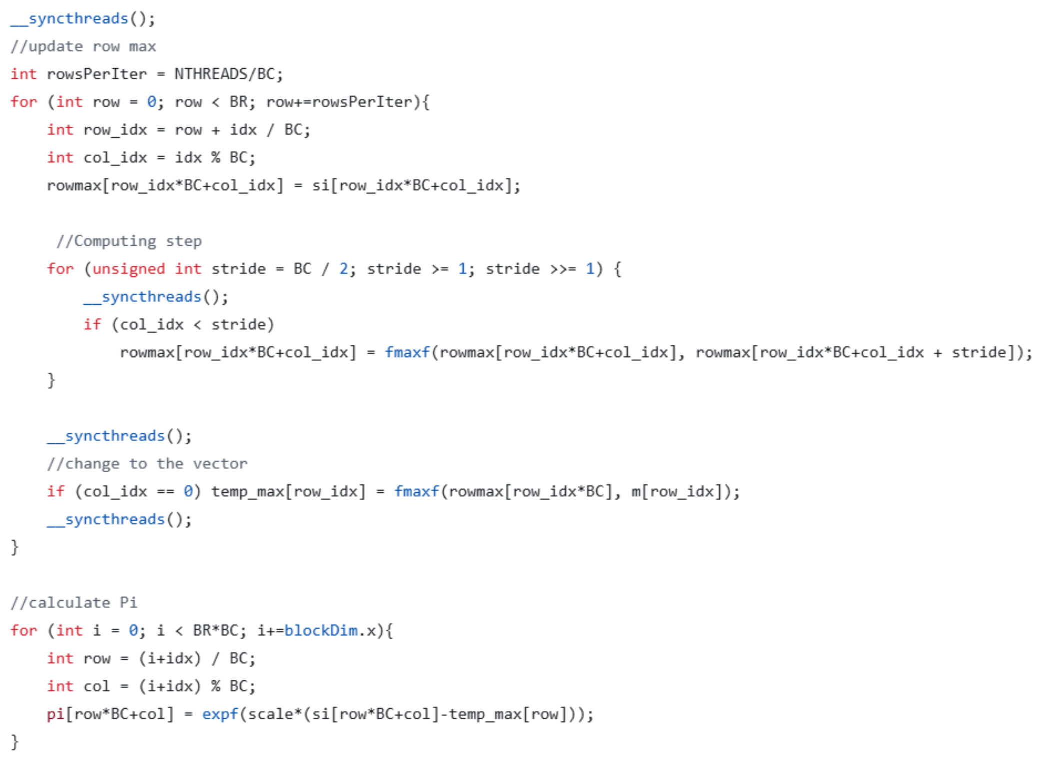
```
for (int j = 0; j < T; j+=BC){
    __syncthreads();

    //loading in Kj and Vj
    int base_row = j;
    for(int tc= 0; tc<BC*D; tc+=blockDim.x){
        int load_pos = tc+idx;
        int bc = (load_pos) / D + base_row;
        int d = (load_pos) % D;
        if (bc < T && load_pos < BC*D){
            ks[load_pos] = k[b * NH*T*D + nh*T*D + bc*D+d];
            vs[load_pos] = v[b * NH*T*D + nh*T*D + bc*D+d];
        }
        else{
            ks[idx+tc] = 0.0f;
            vs[idx+tc] = 0.0f;
        }
    }

    __syncthreads();
```

After everything has been loaded, the computation of the QK matrix can happen. We are implementing causal attention, which means every index in the QK matrix where the absolute column is greater than the absolute row will be set to -inf. This eliminates the contribution of these elements since the softmax of these elements will evaluate to essentially zero. This causes branch divergence, but the degree of branch divergence will be limited since it will only happen for warps that cross the matrix diagonal.



```
//compute QK^T
for(int o_idx=0; o_idx<BR*BC; o_idx+=blockDim.x){
    int o_row = (o_idx+idx) / BC;
    int o_col = (o_idx+idx) % BC;

    float accum = 0.0f;
    for(int i=0; i<D; i++){
        accum += qi[o_row*D+i] * ks[o_col*D + i];
    }
    si[o_row*BC+o_col] = accum;
    if(o_col + base_row > o_row+start){
        si[o_row*BC+o_col] = -FLT_MAX;
    }
}
```

The next step involves determining the max element of the current row, similar to local attention’s reductions. A difference was that there were multiple rows that reduction had to be computed for simultaneously, so they were computed in an iterative manner, with the maximum number of rows the threads could handle being computed in a single iteration until every row was reduced. The maximum of each row strip was compared to the global row maximum and stored in the temp_max buffer. It is critical that the current max isn’t updated yet. This is because we need to keep the old max (current) and new max (temp) so that we can apply the scaling factor to the past output chunks. The softmax numerators were calculated after this and stored in the pi buffer. Another important distinction from the original algorithm is that scaling is being applied to the exponents, which will be applied here and in all the updates to these values.

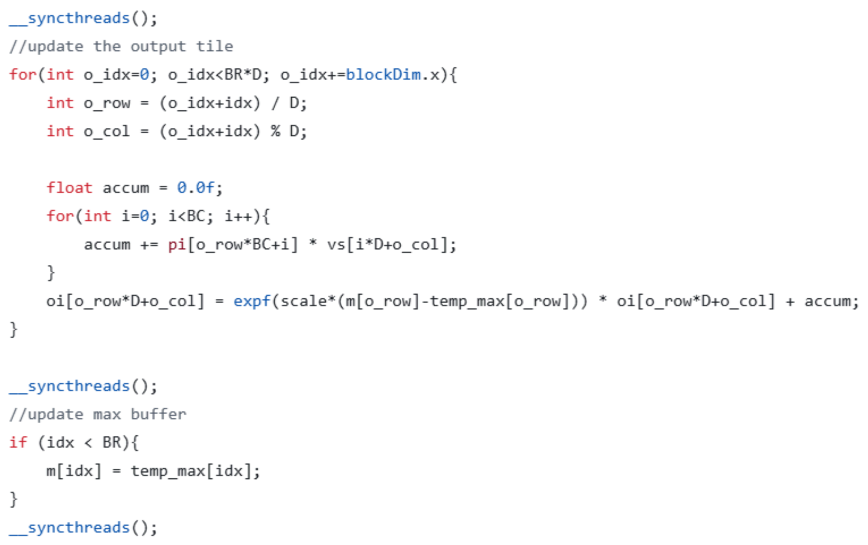


```
__syncthreads();
//update row max
int rowsPerIter = NTHREADS/BC;
for (int row = 0; row < BR; row+=rowsPerIter){
    int row_idx = row + idx / BC;
    int col_idx = idx % BC;
    rowmax[row_idx*BC+col_idx] = si[row_idx*BC+col_idx];

     //Computing step
    for (unsigned int stride = BC / 2; stride >= 1; stride >>= 1) {
        __syncthreads();
        if (col_idx < stride)
            rowmax[row_idx*BC+col_idx] = fmaxf(rowmax[row_idx*BC+col_idx], rowmax[row_idx*BC+col_idx + stride]);
    }

    __syncthreads();
    //change to the vector
    if (col_idx == 0) temp_max[row_idx] = fmaxf(rowmax[row_idx*BC], m[row_idx]);
    __syncthreads();
}

//calculate Pi
for (int i = 0; i < BR*BC; i+=blockDim.x){
    int row = (i+idx) / BC;
    int col = (i+idx) % BC;
    pi[row*BC+col] = expf(scale*(si[row*BC+col]-temp_max[row]));
}
```

After computing the partial softmax, the array holding the running sums of exponentials for each row needs to be updated. The reduction is done on pi in a similar manner as the max reduction, over groups of rows at a time. The current sums are directly stored into the temp_l buffer, used to hold the sums of the current row strip. L, the array that holds the old running sum is updated, first through being scaled to account for a new maximum, and then getting added to by the current chunk's contribution.

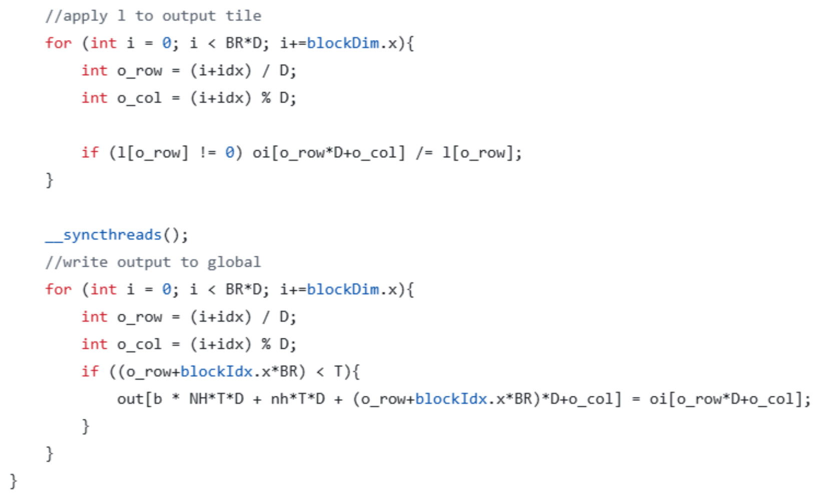
```
__syncthreads();
//sum reduction
for (int row = 0; row < BR; row+=rowsPerIter){
    int row_idx = row + idx / BC;
    int col_idx = idx % BC;
    if (col_idx + base_row <= row_idx+start){
        rowmax[row_idx*BC+col_idx] = pi[row_idx*BC+col_idx];
    }
    else{
        rowmax[row_idx*BC+col_idx] = 0;
    }
    for (unsigned int stride = BC / 2; stride >= 1; stride >>= 1) {
        __syncthreads();
        if (col_idx < stride)
            rowmax[row_idx*BC+col_idx] += rowmax[row_idx*BC+col_idx + stride];
    }
    //update temporary l buffer
    __syncthreads();
    if (col_idx == 0) temp_l[row_idx] = rowmax[row_idx*BC];
    __syncthreads();
}

//update the l buffer
if (idx < BR){
    l[idx] = expf(scale*(m[idx]-temp_max[idx]))*l[idx]+temp_l[idx];
}
```

To end the j loop over the Tr chunks, the new part of the output tile must be computed, through a standard matrix multiplication of the partial softmax buffer with the value buffer. This is added to the old value stored at the output, which was scaled to account for a new maximum, just like the sum array was earlier. This means the max array can finally be updated since the old max isn’t needed anymore for rescaling.



```
__syncthreads();
//update the output tile
for(int o_idx=0; o_idx<BR*D; o_idx+=blockDim.x){
    int o_row = (o_idx+idx) / D;
    int o_col = (o_idx+idx) % D;

    float accum = 0.0f;
    for(int i=0; i<BC; i++){
        accum += pi[o_row*BC+i] * vs[i*D+o_col];
    }
    oi[o_row*D+o_col] = expf(scale*(m[o_row]-temp_max[o_row])) * oi[o_row*D+o_col] + accum;
}

__syncthreads();
//update max buffer
if (idx < BR){
    m[idx] = temp_max[idx];
}
__syncthreads();
```

Outside the j loop, there are two final steps. First, the output tile will need to be divided by the accumulated sum of exponentials, to obtain the full output. We check if the sum is zero to avoid a division by zero error. Then, this output can be written back to global memory, with the appropriate bounds checks.

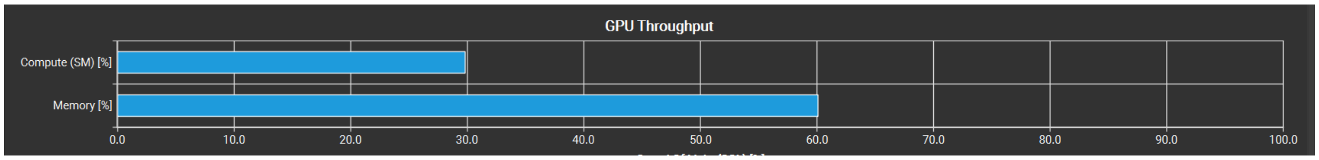


```
    //apply l to output tile
    for (int i = 0; i < BR*D; i+=blockDim.x){
        int o_row = (i+idx) / D;
        int o_col = (i+idx) % D;

        if (l[o_row] != 0) oi[o_row*D+o_col] /= l[o_row];
    }

    __syncthreads();
    //write output to global
    for (int i = 0; i < BR*D; i+=blockDim.x){
        int o_row = (i+idx) / D;
        int o_col = (i+idx) % D;
        if ((o_row+blockIdx.x*BR) < T){
            out[b * NH*T*D + nh*T*D + (o_row+blockIdx.x*BR)*D+o_col] = oi[o_row*D+o_col];
        }
    }
}
```

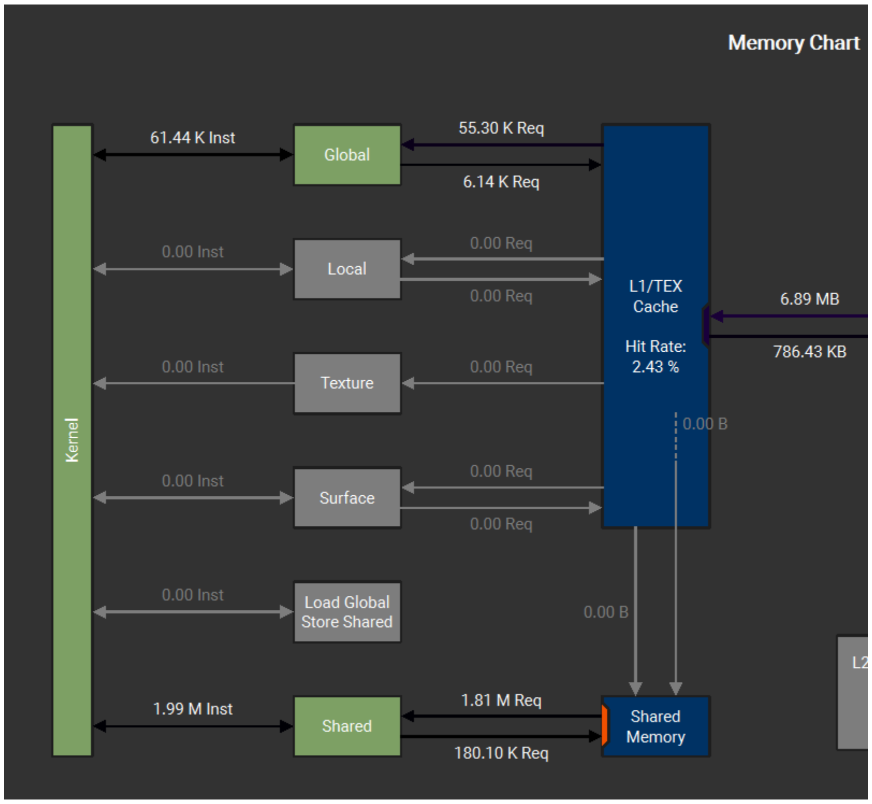
Analysis

Flash attention had a total runtime of 1,134,588 ns with, a 3.91x speedup over local attention and a 416.35x speedup over the baseline.

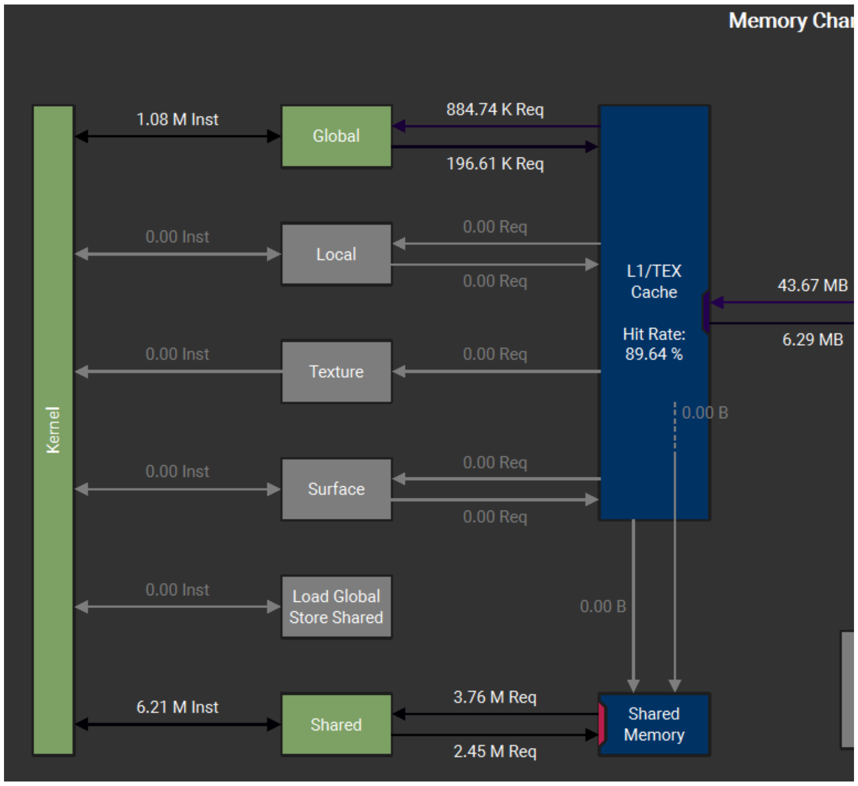
The first thing we noticed in the compute report was that, likely due to reliance on large shared buffers, memory throughput was disproportionately higher than compute throughput.



This may be due to our heavy shared memory usage. By limiting global read/writes as much as possible, most of the data transfer happens through shared memory, which is why the kernel is significantly faster. Since we have more shared buffers to transfer between, there are still overall more memory IO operations which is why memory throughput is significantly higher than compute. Only about 3% of read/writes were global.

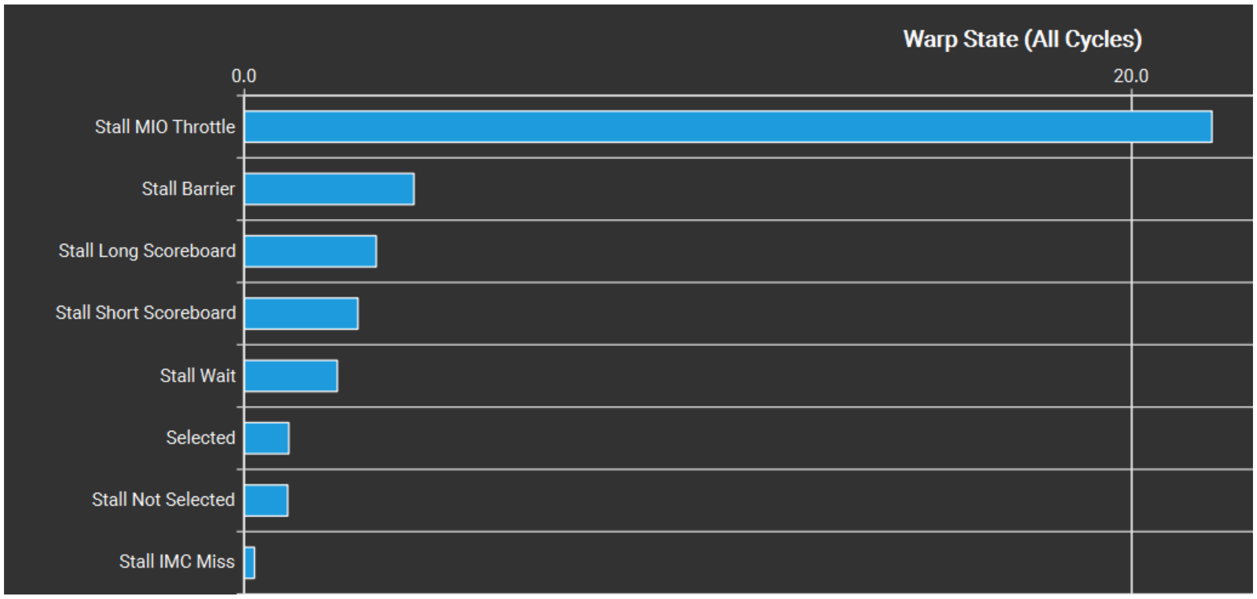


Compare this to local attention- overall, while the throughput was a lot higher, that isn't necessarily a good thing, because a GPU doing more computations and loads/stores per second isn't necessarily faster, due to the fact that different algorithms have different levels of efficiency. Local attention relied on memory IO significantly more than flash attention due to only holding small strips of a row in shared buffers and loading entire block sized chunks of Q,K, and V from global memory to compute these chunks. Here is a memory chart for local attention:



Compared to 3%, local attention has global read/writes account for about 15% of memory IO. This data validates the idea of flash-attention- since global read/writes are the largest bottleneck, reducing them will lead to a speedup.

For this low percentage of global reads, the tradeoff is that there is worse warp stalling than with local attention, or even baseline attention. Nsight Compute determines warps stalled for an average of 21.8 cycles, far higher than the 14 cycles in baseline attention. MIO throttle is usually due to too many concurrent shared memory loads/stores. The throttling results in less SM utilization, which is why our achieved occupancy was only 32.75%. This is a tradeoff that is necessary to minimize global accesses, however our team determined that to further improve flash attention, we could find ways to use memory more optimally, maybe through utilizing register arrays to substitute small buffers like temp_max and temp_l.



Additionally, looking at this chart, stall barrier is the second most significant cause for warp inactivity. This could be because the reduction steps done in flash attention were similar to local attention, and suffer from the same issues. Additionally, there are many synchronization barriers used in the inner j loop that may also contribute to this problem.

Overall, the flash attention kernel was a significant improvement to local attention and baseline attention. Further improvements to alleviate the issues discussed above are implemented in proposal 2.

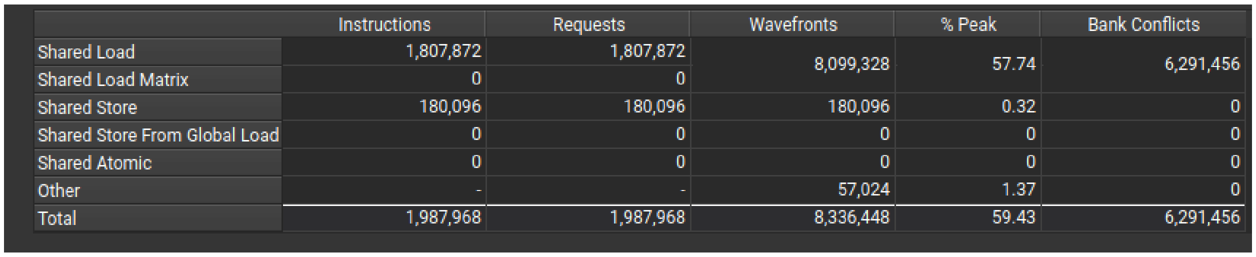
Flash Attention Improvements (Prop_2)

Introduction

While flash attention at its baseline implementation represents some of the intelligent and creative ways to improve performance by relying on less global memory operations, we decided to investigate techniques that could be applied to our implementation of the algorithm that would lead to additional inference time speed ups. In the following sections we go through these changes and how they work along with an analysis of these changes allowing for improved performance over a baseline implementation of flash attention.

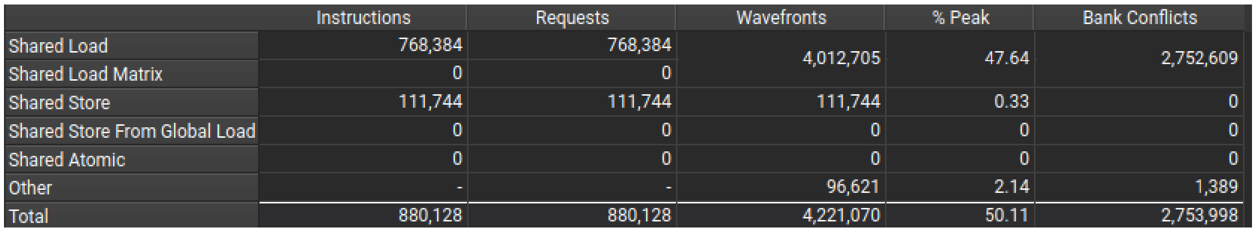
Shared Memory Usage

In our original script we use static shared memory allocations for each individual portion required for computing flash attention such as the q,k,v tiles, output tiles, and the exponential sum and max sum buffers. While this technique allows for easier indexing, having these distinctive shared memory chunks led to over 6.2 million bank conflicts which led to performance degradation:



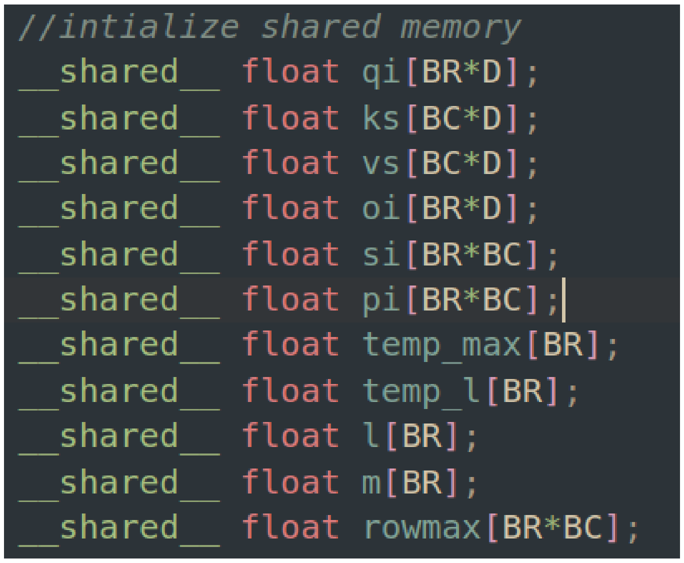
	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	1,807,872	1,807,872	8,099,328	57.74	6,291,456
Shared Load Matrix	0	0			
Shared Store	180,096	180,096	180,096	0.32	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	57,024	1.37	0
Total	1,987,968	1,987,968	8,336,448	59.43	6,291,456

Instead of doing this we instead optimizing our loading pattern so that one large contiguous chunk of shared memory was allocated instead allowing for a much more coalesced access pattern that allowed for efficient data transfer. After making this change we found that the number of bank conflicts significantly decreased to only 2.7 million bank conflicts:



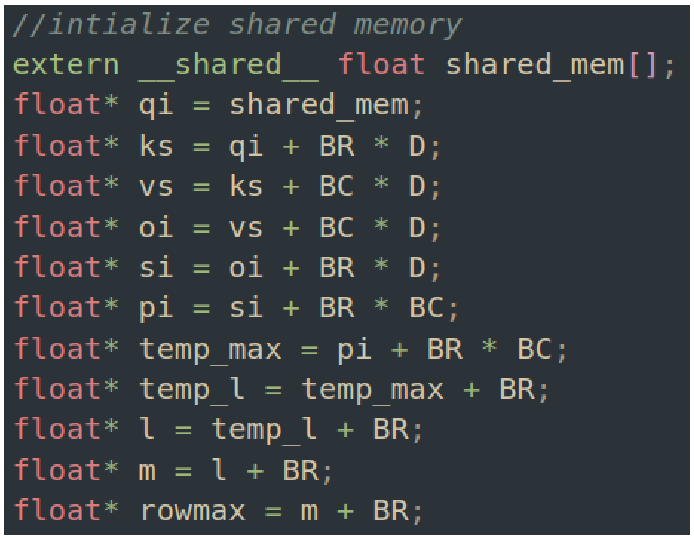
	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	768,384	768,384	4,012,705	47.64	2,752,609
Shared Load Matrix	0	0			
Shared Store	111,744	111,744	111,744	0.33	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	96,621	2.14	1,389
Total	880,128	880,128	4,221,070	50.11	2,753,998

What this looks like in practice is the following change in the code, where pointers are assigned to different points in one common chunk of dynamically allocated shared memory rather than multiple statically defined chunks.



```
//intialize shared memory
__shared__ float qi[BR*D];
__shared__ float ks[BC*D];
__shared__ float vs[BC*D];
__shared__ float oi[BR*D];
__shared__ float si[BR*BC];
__shared__ float pi[BR*BC];
__shared__ float temp_max[BR];
__shared__ float temp_l[BR];
__shared__ float l[BR];
__shared__ float m[BR];
__shared__ float rowmax[BR*BC];
```

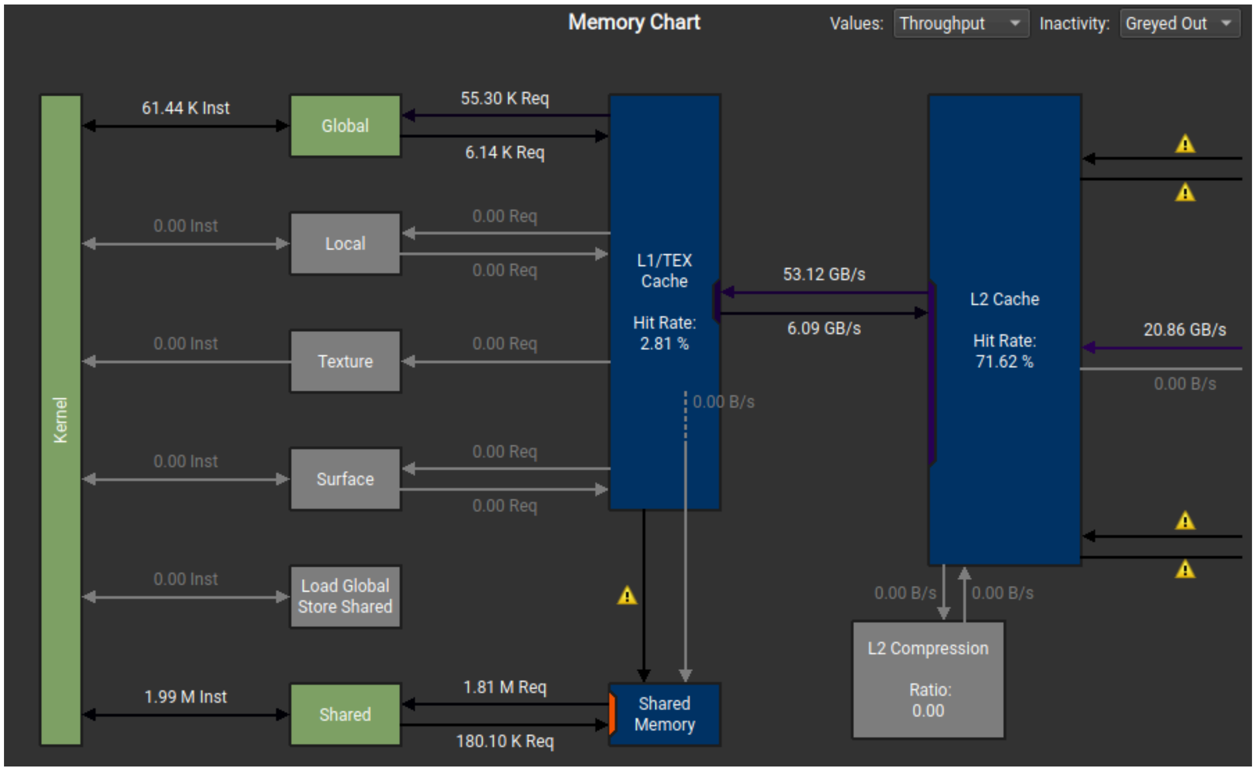
Original Shared Memory Pattern



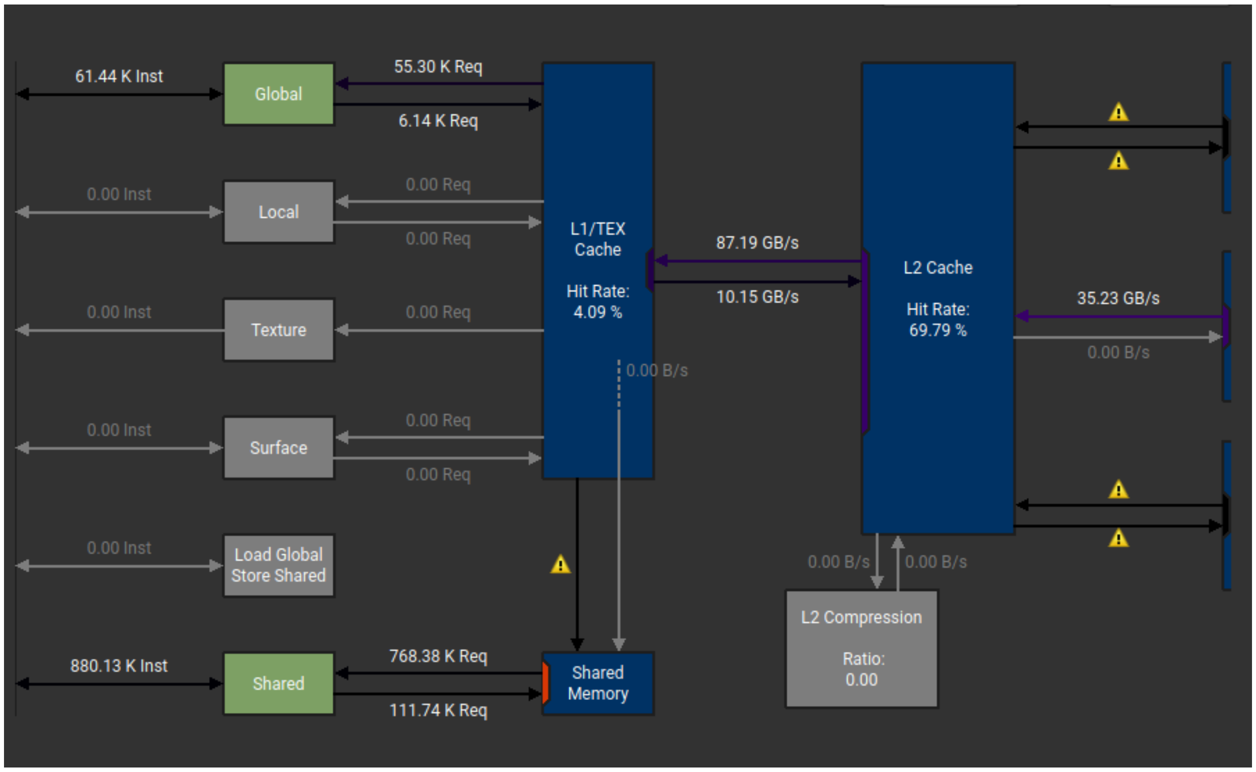
```
//intialize shared memory
extern __shared__ float shared_mem[];
float* qi = shared_mem;
float* ks = qi + BR * D;
float* vs = ks + BC * D;
float* oi = vs + BC * D;
float* si = oi + BR * D;
float* pi = si + BR * BC;
float* temp_max = pi + BR * BC;
float* temp_l = temp_max + BR;
float* l = temp_l + BR;
float* m = l + BR;
float* rowmax = m + BR;
```

Optimized Shared Memory Pattern

Each required memory chunk is offset by the size of the previous entry allowing for future parts of code to index in essentially the same format while in practice having much more efficient usage of shared memory. We can confirm that this is the case by looking at the memory throughput charts between our original flash attention kernel and our optimized flash attention kernel.

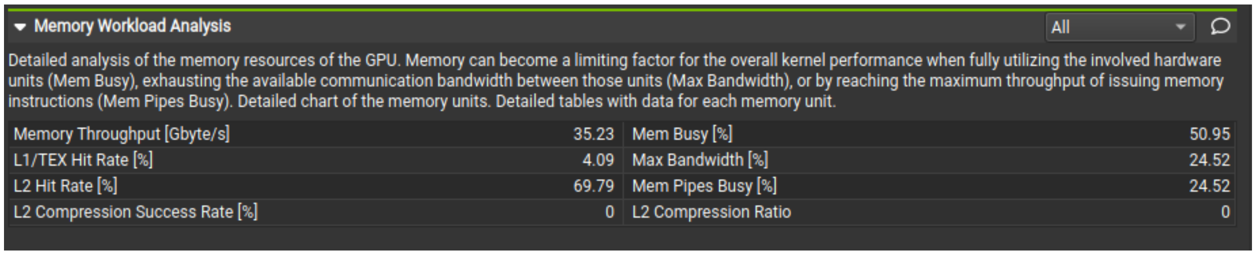


Original Flash Attention Memory Chart



New Flash Attention Memory Chart

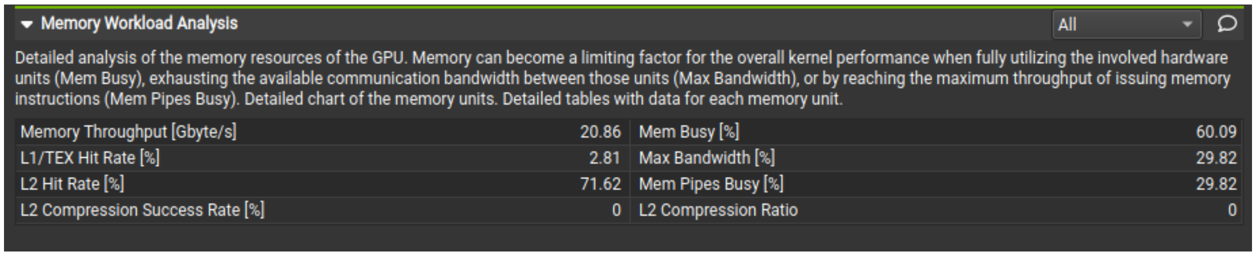
As seen in these charts, the bandwidth of operations is greatly improved as the access pattern is much more standardized. In the original flash attention implementation we have a memory throughput of 20.86 GB/s while in our optimized kernel we are now able to achieve a memory throughput of 35.23 GB/s


Memory Workload Analysis All

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/s]	35.23	Mem Busy [%]	50.95
L1/TEX Hit Rate [%]	4.09	Max Bandwidth [%]	24.52
L2 Hit Rate [%]	69.79	Mem Pipes Busy [%]	24.52
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Optimized Kernel


Memory Workload Analysis All

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

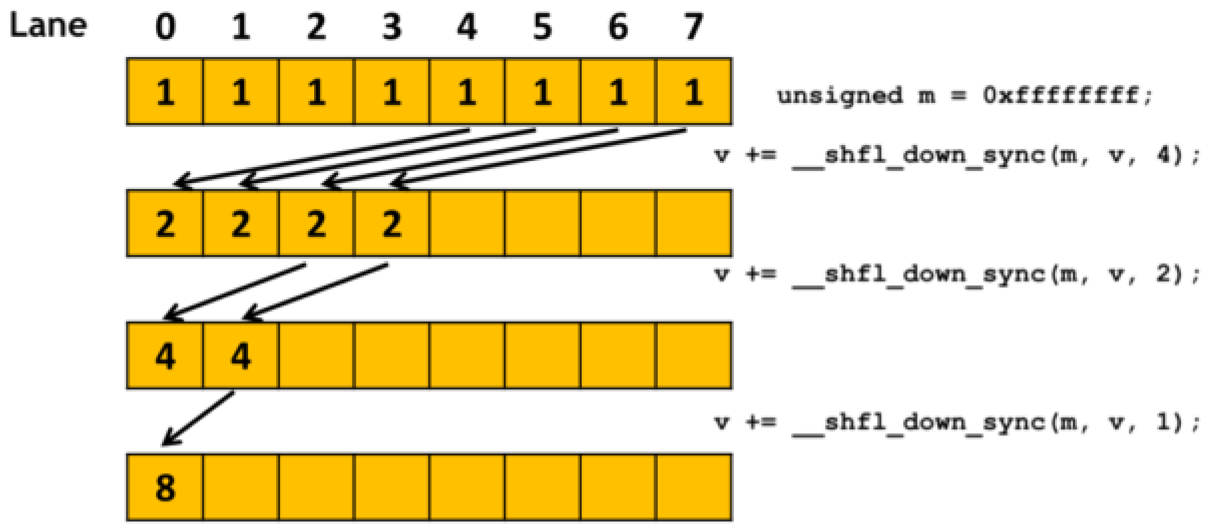
Memory Throughput [Gbyte/s]	20.86	Mem Busy [%]	60.09
L1/TEX Hit Rate [%]	2.81	Max Bandwidth [%]	29.82
L2 Hit Rate [%]	71.62	Mem Pipes Busy [%]	29.82
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Original Flash Attention Kernel

This allows for warps to become available for scheduling much faster as they aren't stalled by memory for as long. Other techniques that contributed to this improvement are discussed in the section below.

Warp Shuffle Operations

One of the most fundamental operations in modern day GPU algorithms is parallel scans and reductions which we spent some time discussing in lecture. Some common algorithms include Brett Kung and Kogge-Stone which attempt to do reductions through phases of loading, associative operations, and writes to global memory requiring careful synchronization between levels to ensure that threads in a block have properly synchronized before moving to the next phase. Recognizing the importance of these kinds of computation patterns in several algorithms, NVIDIA introduced warp level primitive operations which allow threads in a warp to work together to do operations like reductions which can greatly increase performance and reduce synchronization requirements. For example to perform a sum reduction over a warp we can use the __shfl_down_sync operation as shown in this example from the NVIDIA Developer's Blog


Lane
0 1 2 3 4 5 6 7
1 1 1 1 1 1 1 1
unsigned m = 0xffffffff;
v += __shfl_down_sync(m, v, 4);
2 2 2 2
v += __shfl_down_sync(m, v, 2);
4 4
v += __shfl_down_sync(m, v, 1);
8

Example of Warp-Level Primitives from the NVIDIA Developer Blog

How this works is that m indicates all the threads that should be active in the reduction while the third parameter indicates how many threads should be active during that iteration. With a mask of all Fs we are essentially saying we want a reduction over all threads in this warp and as we halve the third parameter on each iteration we reduce the number of active threads for that iteration. In our case, as described in the above section on flash attention, we must compute the temp_l and temp_max buffers in order to perform an online softmax. This is perfectly suited for a reduction operation as each row Si where we have to perform a max reduction is BC elements and each row of Pi where we have to perform a sum reduction is also BC elements. BC is less than the size of a warp allowing us to use these warp primitives without block level primitives leading to a lot of efficiency potential. Below is a snippet of how the max of each row of si can be computed using the warp-level shuffle operation. A similar technique is used for computing the row sum associated with the pi matrix.

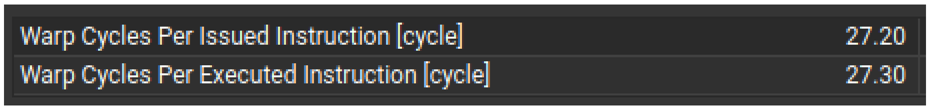
```
C/C++
float max = si[row_idx * BC + col_idx];

//max reduction
for (int mask = BC/2; mask > 0; mask >>= 1) {
        float val = __shfl_down_sync(0xFFFFFFFF, max, mask, BC);
        max = fmaxf(max, val);
}

if (col_idx == 0) {
        temp_max[row_idx] = fmaxf(max, m[row_idx]);
}
```

While this operation is harder to trace down within the Nsight Compute Report there are some metrics that indicate that these operations are more efficient besides the reduction in latency. One observation that can immediately be made is that the intermediary values associated with computing the max are now stored in a register rather than in the rowmax shared memory buffer. Below is the original which uses a row_max buffer in shared memory:

```
C/C++
int row_idx = row + idx / BC;
int col_idx = idx % BC;
rowmax[row_idx*BC+col_idx] = si[row_idx*BC+col_idx];

 //Computing step
for (unsigned int stride = BC / 2; stride >= 1; stride >>= 1) {
        __syncthreads();
        if (col_idx < stride)
        rowmax[row_idx*BC+col_idx] = fmaxf(rowmax[row_idx*BC+col_idx],
rowmax[row_idx*BC+col_idx + stride]);
}

__syncthreads();
//change to the vector
```

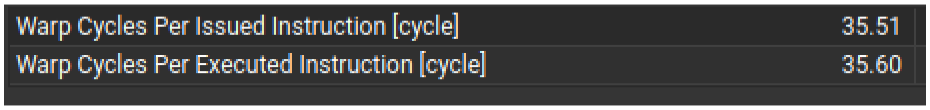
```
if (col_idx == 0) temp_max[row_idx] = fmaxf(rowmax[row_idx*BC], m[row_idx]);
__syncthreads();
```

This directly reduces the number of calls made to shared memory which is apparent when looking at the Nsight Compute Memory Charts included above. Looking at the number of instructions issued to shared memory it can be seen that the implementation that uses warp level primitives issues only 880k requests to shared memory while the original flash attention kernel issued 1.99M shared memory requests. This leads to less stalling as the SM can now schedule warps more frequently as they don't have to wait for data as long. This conclusion is supported by the data on stall cycles below:



Warp Cycles Per Issued Instruction [cycle]	27.20
Warp Cycles Per Executed Instruction [cycle]	27.30

Optimized Flash Attention



Warp Cycles Per Issued Instruction [cycle]	35.51
Warp Cycles Per Executed Instruction [cycle]	35.60

Unoptimized Flash Attention

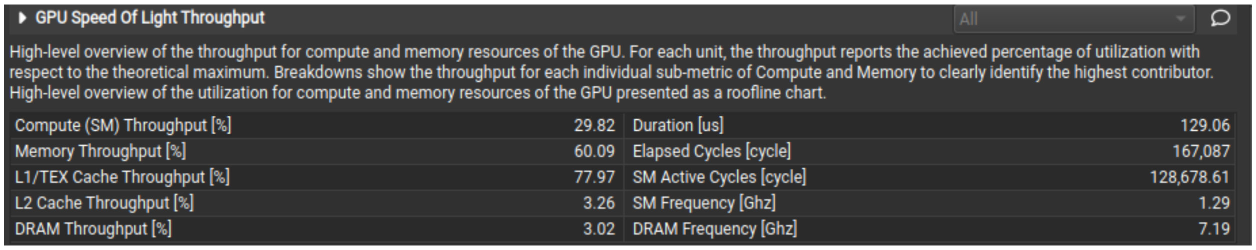
As shown in the images above, these changes along with the dynamic allocation of a large buffer for shared memory allow for higher memory transfer rates in GB/s which allow the warp cycles per instruction to be reduced. This is one of the primary factors contributing to the reduced latency kernel we were able to develop.

Micro Optimization

One of the final smaller changes we made was to do some operation fusing. In the Flash Attention 2 algorithm they mention that the l vector which stores the sum of softmax values for each row must be applied to the output tile on chip and then written to the HBM in the proper location. When we initially implemented the algorithm we implemented these steps separately however it can be seen that we can do a loop fuse operation to reduce the number of total operations. This didn't have a significant effect on performance, but was a cleaner way to perform the operation.

Analysis

Summing these changes together we were able to achieve a kernel for computing flash attention that performed significantly better compared to our baseline implementation. While many of the statistics have been analyzed above we include some additional important findings here.

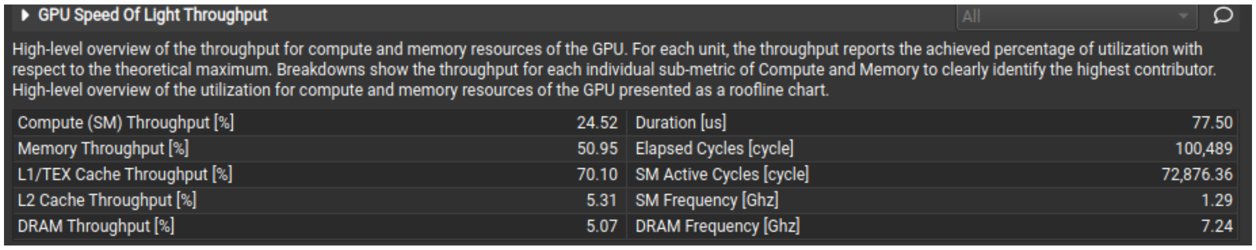


GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	29.82	Duration [us]	129.06
Memory Throughput [%]	60.09	Elapsed Cycles [cycle]	167,087
L1/TEX Cache Throughput [%]	77.97	SM Active Cycles [cycle]	128,678.61
L2 Cache Throughput [%]	3.26	SM Frequency [Ghz]	1.29
DRAM Throughput [%]	3.02	DRAM Frequency [Ghz]	7.19

Original Speed of Light Throughput



GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	24.52	Duration [us]	77.50
Memory Throughput [%]	50.95	Elapsed Cycles [cycle]	100,489
L1/TEX Cache Throughput [%]	70.10	SM Active Cycles [cycle]	72,876.36
L2 Cache Throughput [%]	5.31	SM Frequency [Ghz]	1.29
DRAM Throughput [%]	5.07	DRAM Frequency [Ghz]	7.24

Optimized Kernel Speed of Light Throughput

As seen in this diagram our optimized kernel achieves a lower overall duration despite having lower compute and memory throughput. The reason this occurs is for a number of reasons that we already touched on in previous sections. For the memory throughput we decreased the number of overall shared memory operations allowing us to decrease throughput, and for our compute throughput we used the warp-level instructions and loop fusing to reduce total operations. This shows that when trying to optimize for inference speed baseline throughput is not always a good measure. In most cases looking more deeply into Nsight Compute can reveal more specific and valuable regions where changes can be made. As seen in the duration section, our new kernel is roughly **1.66x faster** than our original flash attention representing a fairly significant improvement in speed.

Matrix Multiplication Optimization (Req_10)

Background

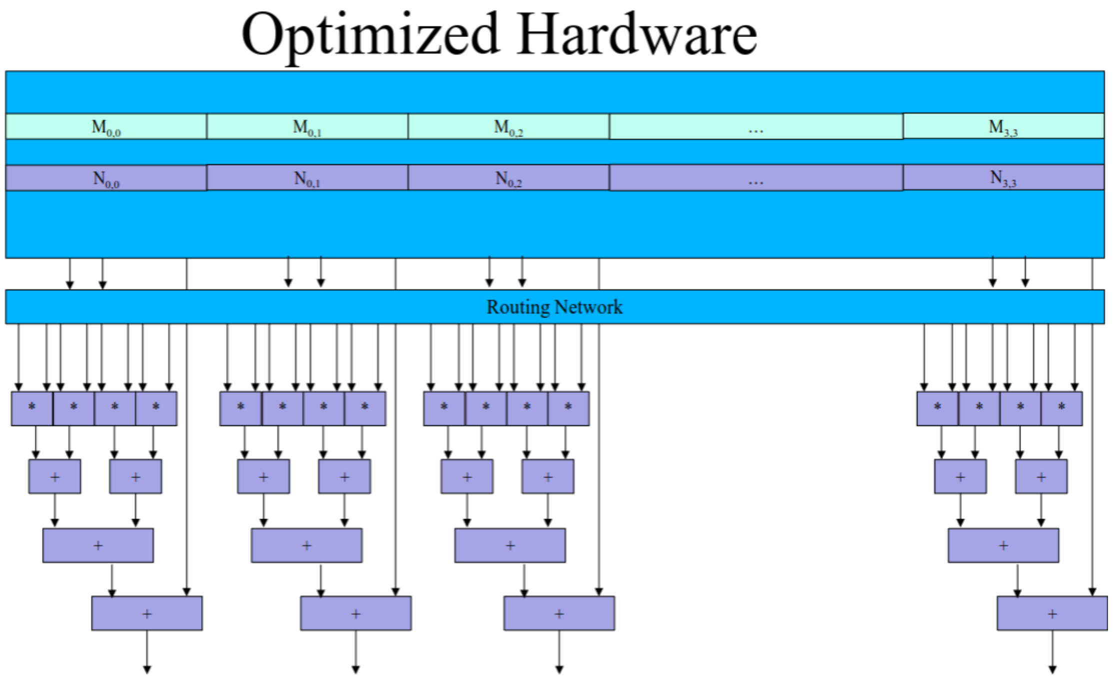
Matrix Multiplication is a fundamentally important operation in Transformer inference often accounting for a large portion of the runtime. This is due to the large weight matrices that often reach 4,000,000+ bytes for models such as those we tested on like GPT-2. From our initial Nsight Systems report which profiled the runtimes of our initial kernels we found that matrix multiplication had a total operation time of over one hundred million nanoseconds, or over 17% of the total runtime, only being surpassed in runtime by our attention kernels. This is a significant problem as both our attention kernels and matrix multiplication kernels were playing a large role in the long runtimes. Looking more clearly at why this was the case we looked at the Nsight Compute report which revealed that the compute throughput is poor with a Speed of Light compute percentage of 28.14% and memory percentage of 95.64%. The reason for this is fairly straightforward in that each location in our output required pulling 2*C values from global memory and sequentially performing C total multiplications. There is a lot of data being loaded from global memory hence the high memory throughput but proportionally not as much computation. The reader can see how performing uncoalesced global memory accesses and sequential computation afterwards is an inefficient way of performing matrix multiplication. This leads to stall cycles while the memory is loaded into registers reducing the throughput of each streaming multiprocessor.

Potential Solutions

There are various strategies to mitigate this that we investigated such as using register tiling which essentially uses both shared memory and registers to load tiles which when summed leads to higher overall throughput. These strategies did improve performance but not as substantially as advanced kernels provided in CuBLAS. While the internals of CuBLAS aren't fully available online, it can reasonably be assumed that CuBLAS takes advantage of the advanced Tensor Core hardware available on our Ampere based GPUs available in the Delta cluster. While we couldn't quite get CuBLAS performance we were able to generate a significant speedup which we analyze more clearly in a future section of the report.

Tensor Core Introduction

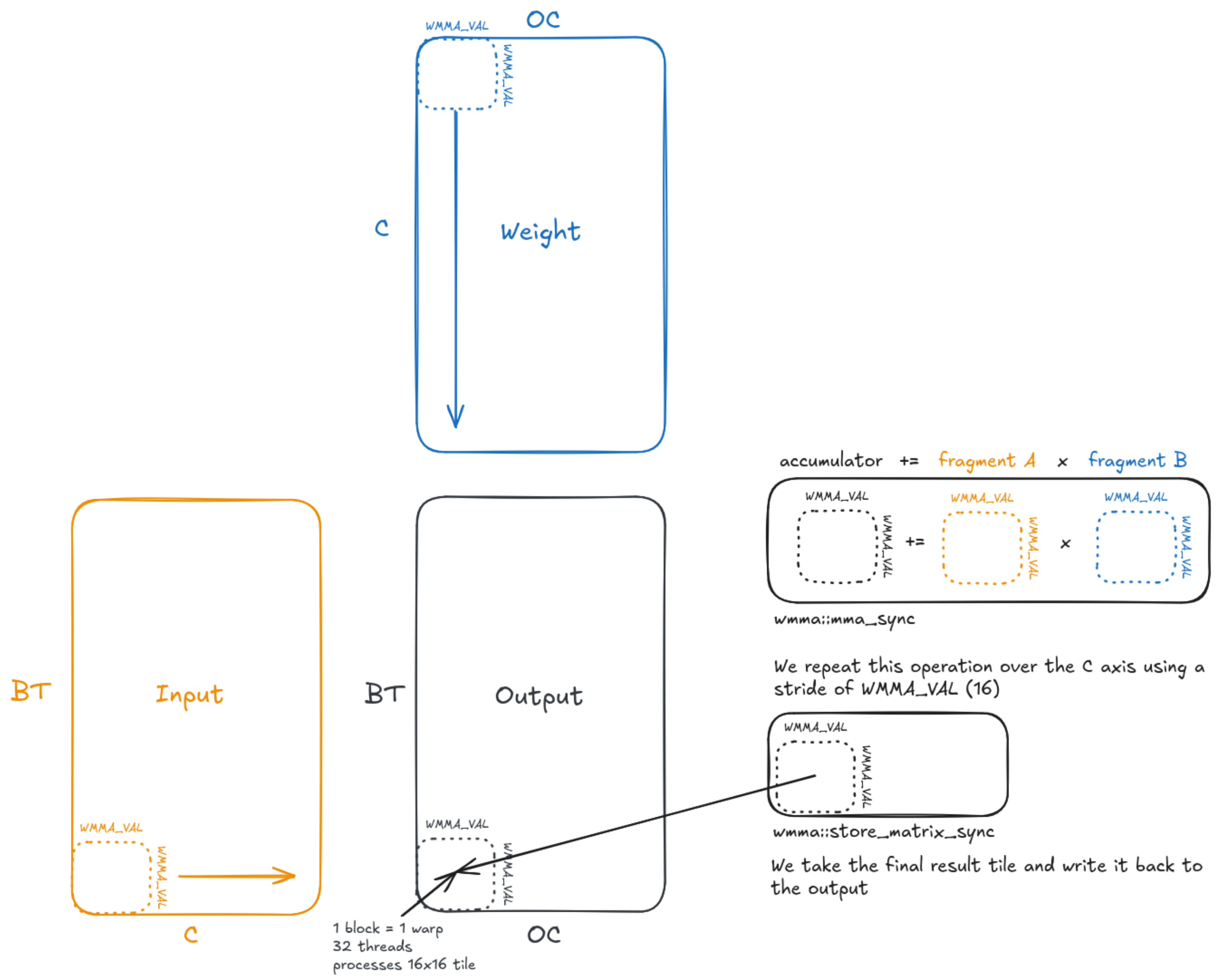
Tensor Cores, first introduced in the Volta architecture GPUs, serve as powerful hardware units for performing matrix multiplication operations with very high throughput as they have custom pathways for loading data from global memory in optimized patterns into what is called a fragment which are then multiplied and accumulated. By creating a hardware pathway that integrates both the loading phase into registers, the multiplications, and accumulations, we are able to improve the overall throughput of our kernels.



Tensor Core Hardware Layout, Source Lecture 23: Tensor Operations - Professor Sanjay Patel

In the above figure, we see this layout in action. Suppose we have matrices M and N, then given a warp which is of size 32 threads we can load 2 tiles each of size 4x4 which yields a total of 32 loads. We then pass these values through a routing matrix which appropriately routes the data to the appropriate arithmetic units which perform multiplications and accumulations. In this scheme, for a given output we require 4 multiplication units and 4 addition units. From the diagram we can see that instead of distributing these multiplications across a sequential loop we instead distribute across hardware improving the throughput through parallelization. To perform these calculations we require our input matrices to follow some guidelines in order to ensure proper loading and that we stay within the limits of the register file. To address these two challenges we align our arrays to the total tile size or 16, and reduce the precision to FP16 in order to avoid clobbering the register files. The reason we have to avoid clobbering the register files is that if we try to put too much data in the register file we encounter a problem known as register spilling where some of the data goes into the global memory. This will again lead to stall cycles as loading data from the global memory can take hundreds of clock cycles.

Implementation & Mechanism

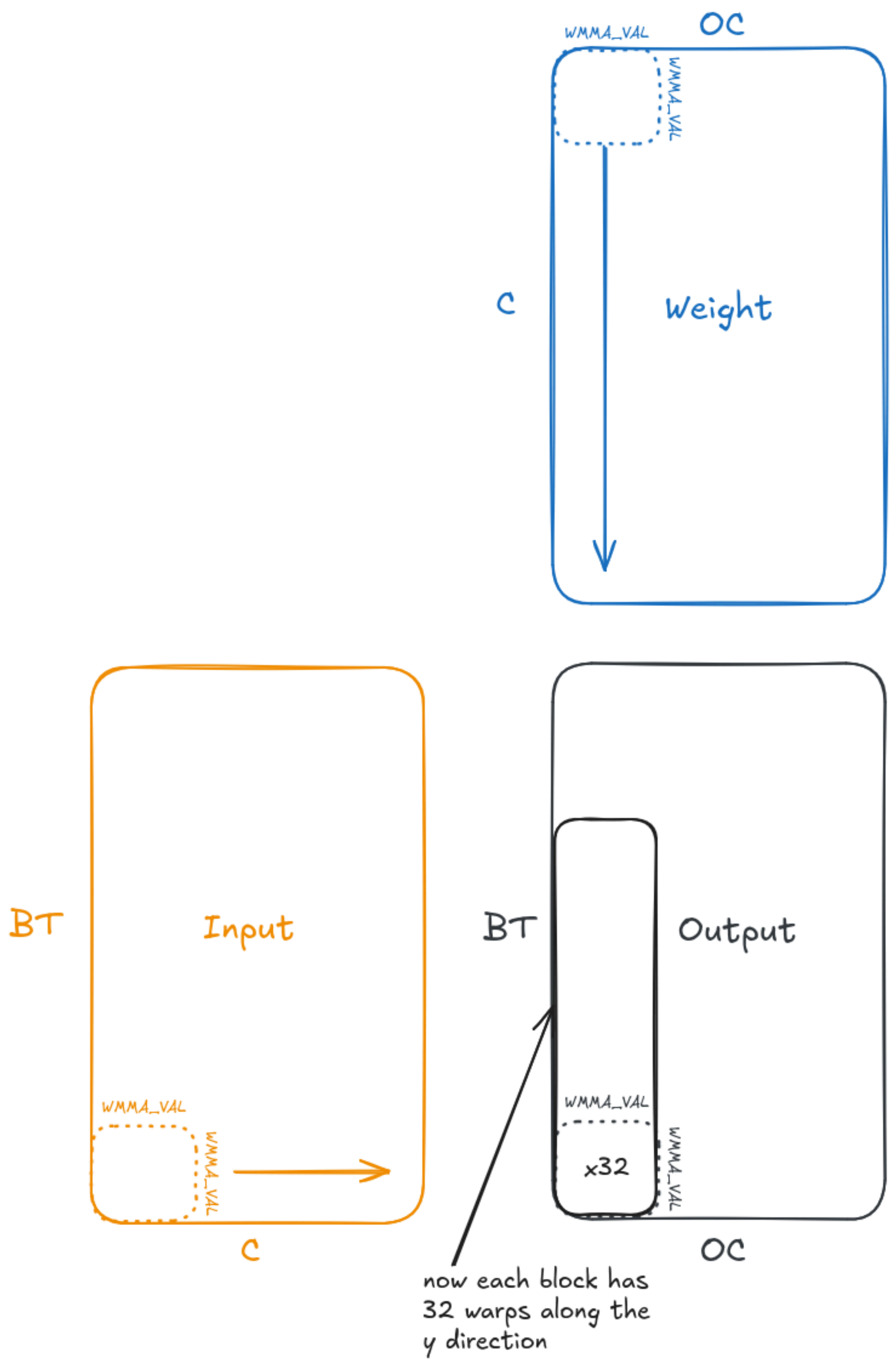


Basic Mechanism of Tensor Core Matrix Multiplication, Illustrated by Raghav Tirumale

As an improvement to the basic matrix multiplication kernel developed in the first phase of the project we developed an initial version of a tensor core multiplication kernel that uses an arrangement where each block has 1 warp. In the tensor core operation mechanism each warp does the computation for one tile of the output. There are various supported sizes however we choose the 16x16 square tile as it makes the indexing scheme easier. Since each block does the computation for a 16x16 output tile, the grid can be assigned as ceil((1.0*BT)/16) * ceil((1.0*OC)/16). Then each warp will load the appropriate data from the input matrix and weights matrix and perform repeated mma_sync operations which when complete can be written to the output matrix via a store_matrix_sync operation. While this method does on average improve the performance by roughly 21x compared to our base matrix multiplication kernel as we showed in phase 2 of the project, there are multiple areas to investigate potential optimization strategies.

Larger Block Sizes

During the final phase of the project we tried out many interesting approaches to optimization, however in the end we found that our particular application case the best optimizations were fairly straightforward. We discuss some of those other approaches in a future section along with other challenges to optimizing tensor core in a future section. While the common advice is to use one warp per block when working with tensor cores, one particularly powerful approach is to arrange multiple warps into a single block. This requires changing the indexing strategy slightly within the kernel in order to determine what operation each thread should do and also adjusting the grid dimensions. As a recap, each block used to have one warp where all threads were along the x dimension of the block (blockDim.x = 32). In this change we modified the blockDim.y to also have a dimension of 32 thus increasing the effective warps per block along the y direction. With this change we can adjust the grid to have dimensions ceil((1.0*BT)/16) * ceil((1.0*OC)/(16*32)) as along the OC dimension we have a reduction of the number of blocks by a factor of 32. The diagram below highlights how this changes the operation:



Mechanism of Large Block Tensor Core Matrix Multiplication, Illustrated by Raghav Tirumale

As one can see from this sketch, we now need to account for this change in the block size while calculating our indexing within the kernel. This is done in the following scheme.

```
None
const int WARP_X = blockIdx.x;
const int WARP_Y = threadIdx.y + blockIdx.y * blockDim.y;
```

Each block deals with one warp along the x direction and multiple warps along the y direction. The indexing remains largely the same as we already constructed expressions to get the appropriate positions within the output. We calculate these output positions as the following, where WMMA_VAL is our tile size of 16 described earlier giving us the right positions for fragment loading.

```
C/C++
const int OUT_IDX_X = WARP_X * WMMA_VAL;
const int OUT_IDX_Y = WARP_Y * WMMA_VAL;
```

Then we can load and perform matrix multiplication in the same way as milestone 2 on a per warp basis, where each warp corresponds to a 16x16 tile in the output moving along the C axis with a stride of WMMA_VAL (16).

Kernel Fusing

For inferencing the GPT 2 model we have to perform multiplication between the input and weight matrices. In the tensor core implementation we have this input and weight matrix as float matrices. Additionally the weight matrix is passed with the dimensions OC*C which must be transposed in order to properly be multiplied. In our initial implementation these operations were performed separately which led to unnecessary launch overheads that could be easily reduced. To address this we wrote a kernel called transposeToFP16 for the weight matrix specifically to be transposed and converted in one kernel call. Looking at the Nsight Systems Report we see that this preprocessing time goes from 4.461 million nanoseconds down to 2.533 million nanoseconds or roughly 1.76x faster. This is a highly critical step that could also be fused directly into the matrix multiplication process for even faster performance.

Micro Optimizations

Some other microptimizations that improved performance include including launch bounds on our kernel and using double buffering. For the matrix multiplication kernel, we have a __launch_bounds__(1024, 1) statement which gives more discrete instructions to the compiler on how the registers must be distributed on launch allowing for some slight optimization. A discussion of this is available here. Additionally, we tried to do a double buffering of the mma fragments by swapping between two buffers on each iteration across the C axis which shaved off an insignificant amount of runtime.

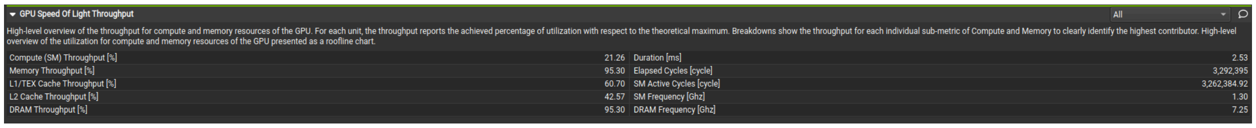
Analysis

In this section we analyze the performance of our optimized tensor core matrix multiplication kernel against our baseline tensor core implementation and cuBLAS. The main performance improvement is visible for larger matrix multiplications so we will focus on that section as performance for other calls is similar or only slightly faster. Looking at the largest matrix multiplication kernel call shows the following results:

Implementation	**Runtime (ms)**	**Speed Improvement**	**% of CuBLAS Performance**
CuBLAS (with Tensor Core)	0.507	61.83x	100%
Tensor Core (Milestone 3)	1.208	25.95x	41.9%
Tensor Core (Milestone 2)	2.484	12.62x	20.4%

Naive Matrix Multiplication	31.35	1x	1.6%

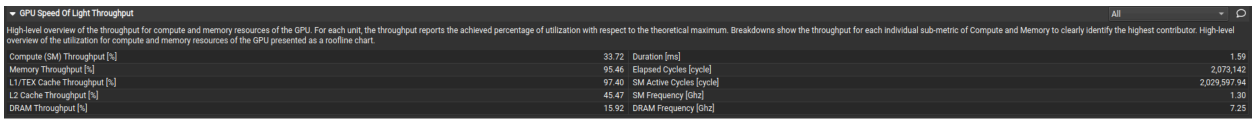
Compared to our initial Tensor Core implementation, we see quite an increase in the performance of our optimized kernel for milestone 3, however we also see that there is still significant improvement scope in achieving a kernel that approaches the tuned kernels provided by the CuBLAS library. Looking at the Nsight Compute report can help us understand why the new implementation is much more efficient compared to our approach from Milestone 2.


GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	21.26	Duration [ms]	2.53
Memory Throughput [%]	95.30	Elapsed Cycles [cycle]	3,292,395
L1/TEX Cache Throughput [%]	60.70	SM Active Cycles [cycle]	3,262,384.92
L2 Cache Throughput [%]	42.57	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	95.30	DRAM Frequency [Ghz]	7.25

Milestone 2 - Speed of Light Throughput

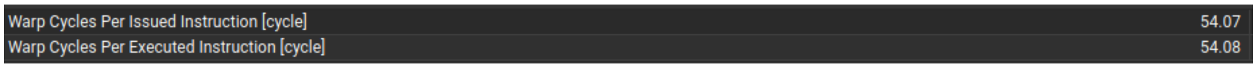

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

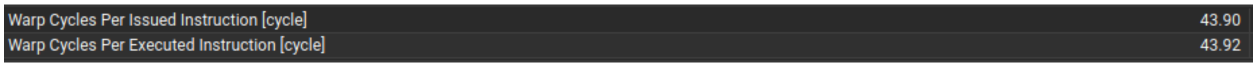
Compute (SM) Throughput [%]	33.72	Duration [ms]	1.59
Memory Throughput [%]	95.46	Elapsed Cycles [cycle]	2,073,142
L1/TEX Cache Throughput [%]	97.40	SM Active Cycles [cycle]	2,029,597.94
L2 Cache Throughput [%]	45.47	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	15.92	DRAM Frequency [Ghz]	7.25

Milestone 3 - Speed of Light Throughput

From these figures we see some interesting behavior that gives us a lense into why the performance of our Milestone 3 approach to matrix multiplication may be faster. Starting with the compute throughput we see that the M3 kernel gets a Speed of Light throughput of 33.72% percent as opposed to the 21.26% that we see with milestone 2. This is likely due to the shift in throughput also visible in the images. For milestone 3, we drastically increased the L1 Cache throughput from 60.7% to 97.4% which would allow for reads from cache that are much faster from direct access from the DRAM banks. The reason for this shift in workload is due to the fact that with larger block sizes adjacent warps within a block become more likely to get cache hits allowing for higher overall throughput as each warp can spend less cycles waiting on data to become available. The following portion of the Nsight Compute Report shows this to be true:


Warp Cycles Per Issued Instruction [cycle]	54.07
Warp Cycles Per Executed Instruction [cycle]	54.08

Milestone 2 - Scheduler Statistics



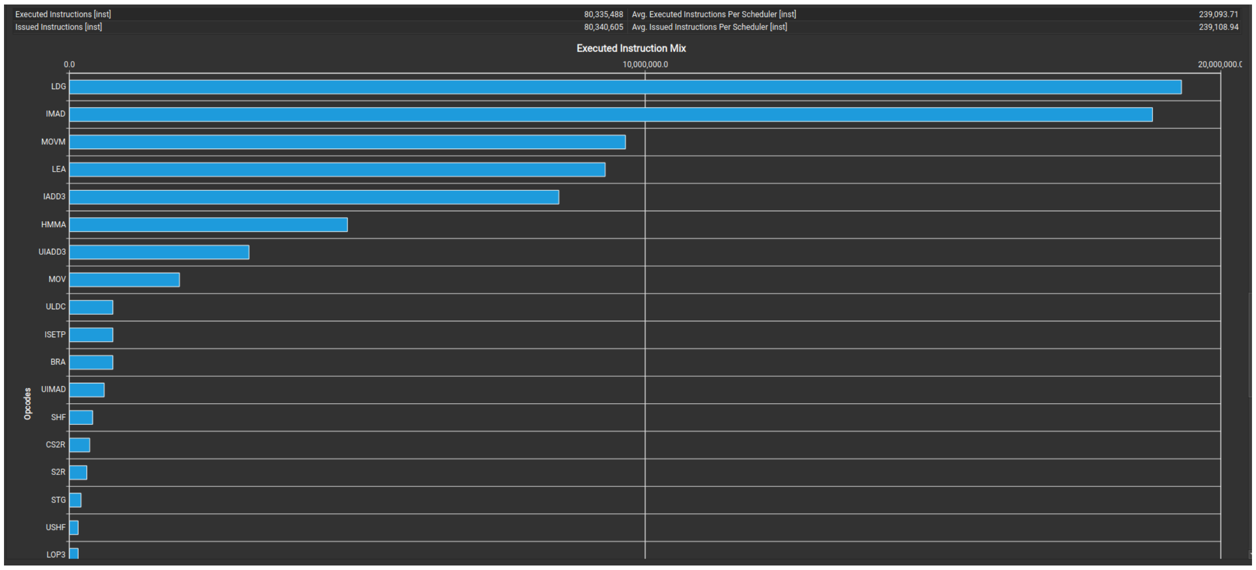
Warp Cycles Per Issued Instruction [cycle]	43.90
Warp Cycles Per Executed Instruction [cycle]	43.92

Milestone 3- Scheduler Statics

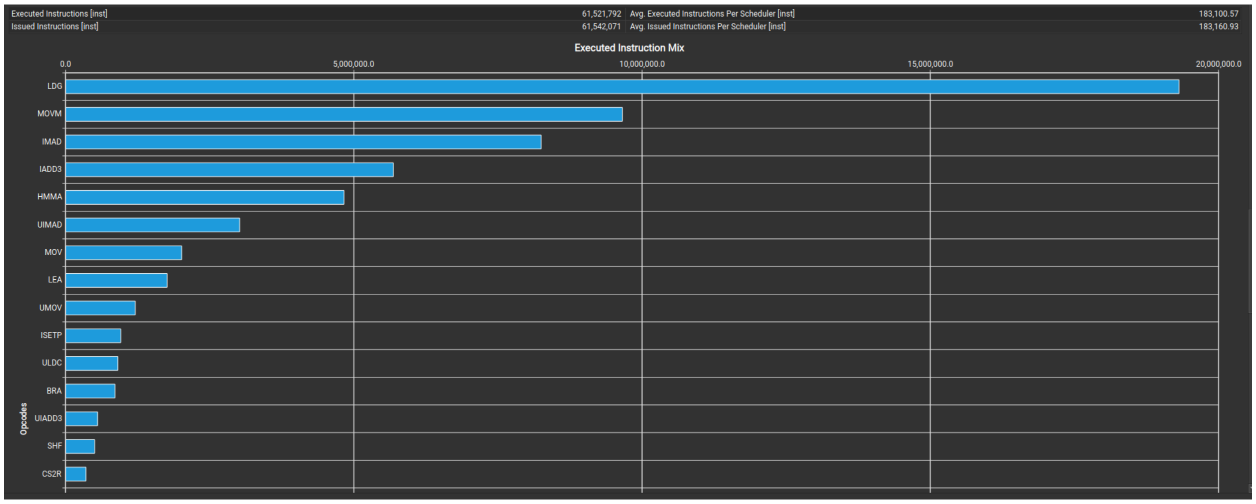
As explained above the scheduler is better able to schedule warps for execution with our milestone 3 implementation as cache hits are much higher. This explains the lower warp cycles per issued instruction. Compared to the L2 cache hit rate in milestone 2 of 11.89%, in milestone 3 we achieve an L2 cache hit rate of 94.33% which serves as a strong explanation for the performance improvement. Another interesting way that we reduced the latency of our M3 kernel is by using an offset to keep track of the start positions within the weight and input matrices that a certain output tile requires. These look like the below:

C/C++
```
half const* is = inp + OUT_IDX_Y*CPAD;
half const* ws = weight + OUT_IDX_X;
```

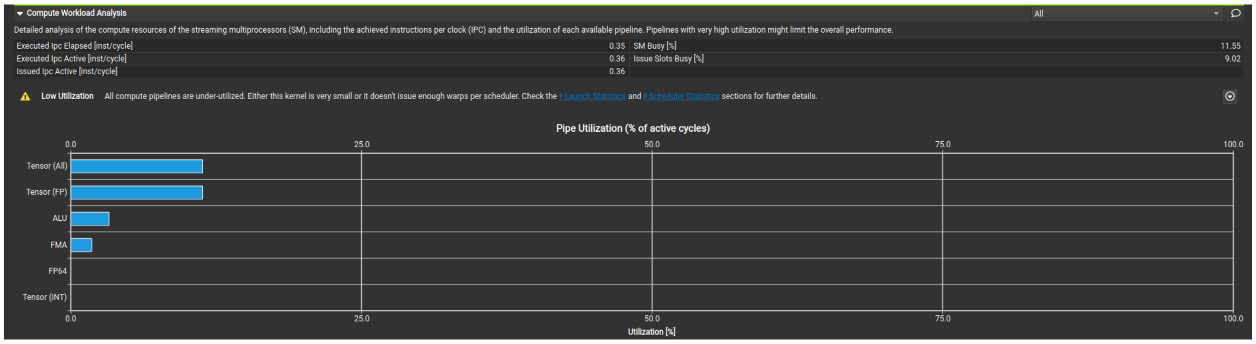
This seems to have reduced the total number of Executed Instructions specifically for the IMAD operation which is used for doing indexing calculations. Compared to milestone 2 which had 80.3M operations total, in milestone 3 we bring this number down to 61.5 Million operations which improved the overall runtime slightly.



Executed Instructions [inst]	80,335,488	Avg. Executed Instructions Per Scheduler [inst]	239,093.71
Issued Instructions [inst]	80,340,605	Avg. Issued Instructions Per Scheduler [inst]	239,108.94


Executed Instructions [inst] 61,521,792
Avg. Executed Instructions Per Scheduler [inst] 183,100.57
Issued Instructions [inst] 61,542,071
Avg. Issued Instructions Per Scheduler [inst] 183,160.93

On the surface these look like fairly decent optimizations, however there are some fundamental reasons why our kernels still need a lot of work. Despite tensor cores having very high throughput through their custom routing network and specialized hardware targeted at computing matrix multiplication operations, we must find better ways of improving performance by trying to get better SM occupancy. Since each warp has to wait 40+ cycles to be scheduled even in our improved implementation, the SM will be idling for long periods of time which reduces throughput. Compared to CuBLAS which has a SM Busy Percentage of 37.35, our kernel only achieves 11.5% utilization. This is likely due to our memory patterns. Other interesting comparisons can be made between our kernel and cuBLAS. The cuBLAS throughput is 35.35% while our implementation has a compute throughput of 33.72%, which is an interesting point of analysis as cuBLAS clearly has better performance. Looking more deeply we see some interesting data:

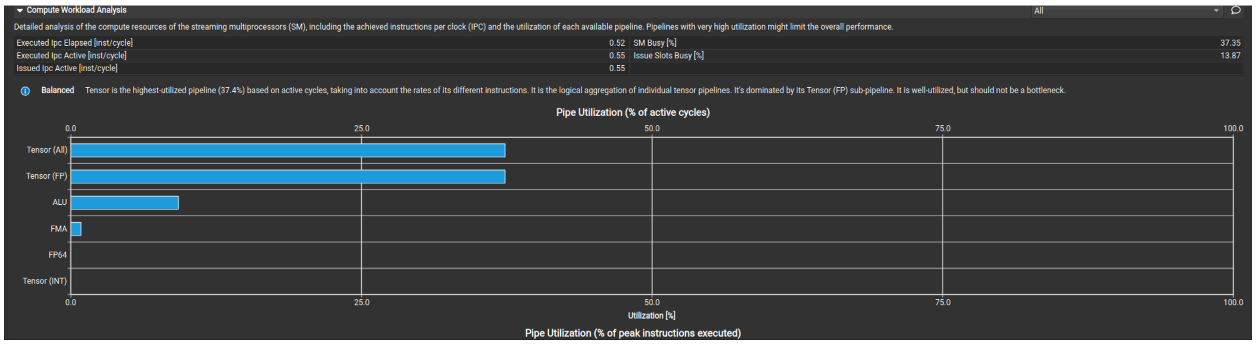

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle] 0.35
SM Busy [%] 11.55
Executed Ipc Active [inst/cycle] 0.36
Issue Slots Busy [%] 9.02
Issued Ipc Active [inst/cycle] 0.36

Low Utilization All compute pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.

Milestone 3 - Workload Analysis



Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

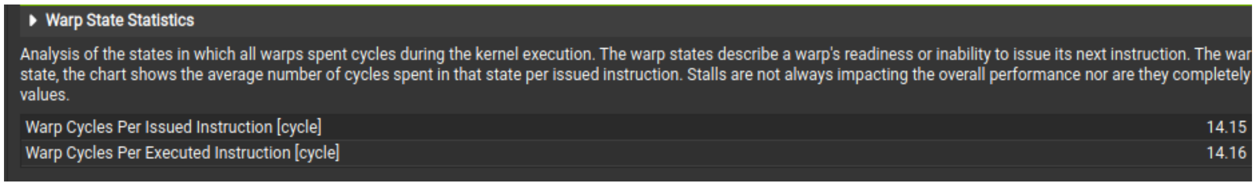
Executed Ipc Elapsed [inst/cycle]	0.52	SM Busy [%]	37.35
Executed Ipc Active [inst/cycle]	0.55	Issue Slots Busy [%]	13.87
Issued Ipc Active [inst/cycle]	0.55		

Balanced Tensor is the highest-utilized pipeline (37.4%) based on active cycles, taking into account the rates of its different instructions. It is the logical aggregation of individual tensor pipelines. It's dominated by its Tensor (FP) sub-pipeline. It is well-utilized, but should not be a bottleneck.

All
Pipe Utilization (% of active cycles)
0.0
25.0
50.0
75.0
100.0
Tensor (All)
Tensor (FP)
ALU
FMA
FP64
Tensor (INT)
Utilization [%]
Pipe Utilization (% of peak instructions executed)

cuBLAS - Workload Analysis

Even though cuBLAS seemingly has similar compute throughput, the percent of cycles where the tensor cores are active is significantly higher. This indicates that the cuBLAS kernel likely has a more optimized method of loading data to ensure that the tensor cores can be active as much as possible. This fact is further solidified when we see that the cuBLAS kernel is able to have the SM schedule in warps much more frequently:

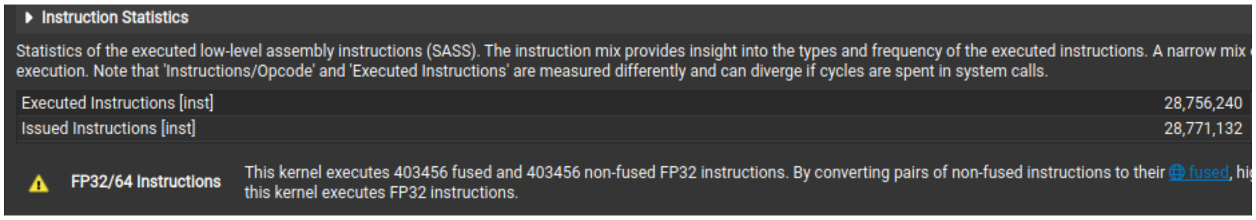


Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The war state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely values.

Warp Cycles Per Issued Instruction [cycle]	14.15
Warp Cycles Per Executed Instruction [cycle]	14.16

The cuBLAS kernel also seems to be doing a lot of fused operations which is something very interesting to investigate in a future effort to improve matrix multiplication performance:



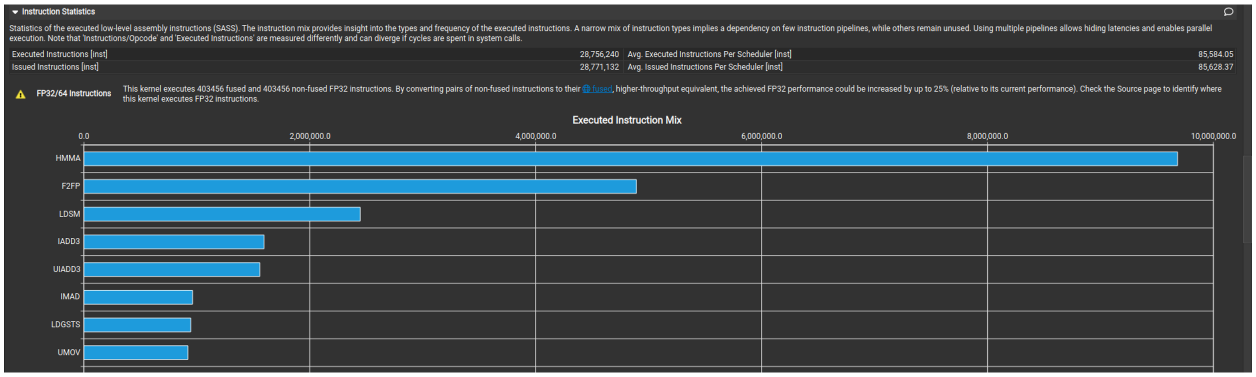
Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	28,756,240
Issued Instructions [inst]	28,771,132

FP32/64 Instructions This kernel executes 403456 fused and 403456 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused, hi this kernel executes FP32 instructions.

From this image we also see that the operations are also greatly reduced as there are only 28M operations as opposed to our kernel from milestone 3 which uses 61M operations. While these changes don't directly indicate a plan of action for competing with this kernel, the higher Tensor Utilization could indicate some advanced loading techniques of the data related to the GPU's architecture that would require further investigation. Another interesting point of comparison is the number of HMMA operations. Looking at the cuBLAS kernel we see that the number of HMMA matrix operations is much higher than our kernel at almost 10 million as opposed to 5 million. The exact reasoning behind this isn't directly apparent just from looking at the report:


Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	28,756,240	Avg. Executed Instructions Per Scheduler [inst]	85,584.05
Issued Instructions [inst]	28,771,132	Avg. Issued Instructions Per Scheduler [inst]	85,628.37

FP32/64 Instructions — This kernel executes 403456 fused and 403456 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP32 performance could be increased by up to 25% (relative to its current performance). Check the Source page to identify where this kernel executes FP32 instructions.

Future Work

While attempts were made at using collaborative shared loading to load in a logical chunk and then make a load into the registers during the fragment loading process, additional work is needed for properly loading large logical tiles so that the overhead of using shared memory outweighs the setup overhead. We could also take a cuBLAS inspired approach and create multiple kernels based on the matrix size in order to ensure techniques involving overhead are only used for larger matrix sizes. Similarly we could also set grid and block sizes based on the matrix sizes to avoid trying to create large configurations for small operations. Other techniques that could potentially be useful include using asynchronous loading to better pipeline our fragment loading process beyond the simple synchronous double buffering approach that was attempted during the time frame of the project. This would be done through NVIDIA's cooperative groups feature, however some minimal testing indicates that the overhead of using this feature may outweigh the potential benefit it creates. Additional techniques could be optimizing the fragment loading pattern to be more optimal however this is quite tricky as the pattern is not very straightforward and differs based on the GPU architecture. Newer generations of GPUs like the Hopper series offer interesting avenues for optimization as well such as new operations like mma_async. These experiments are super exciting and we hope to explore them in the near future. Some interesting articles that served as insightful references while thinking about matrix multiplication included this wonderful article from Alex Armbruster, this blog from an Anthropic researcher, and this medium article by Andy Lo among tons of other interesting papers. We also learned some interesting things about this area by talking with our TA Howie Liu, professor Sanjay Patel, and professor Vlodymyr Kindratenko.

Configuration Sweep (req_5)

Background

Due to the nature of having many different files, kernel implementations, and individual variables in each file, it becomes difficult to test different configurations like block sizes or thread counts. Due to those reasons, it is important to be able to decrease the amount of time taken to test in order to get the most optimized version of a specific kernel. In order to do this, we developed an auto configuration sweep tool that will automatically submit individual jobs for each configuration given. It will make a separate folder environment for each job, so all jobs will not affect each other, and also automatically save profiles into that sub configuration folder. This script, however, does not automatically analyze results due to time constraints.

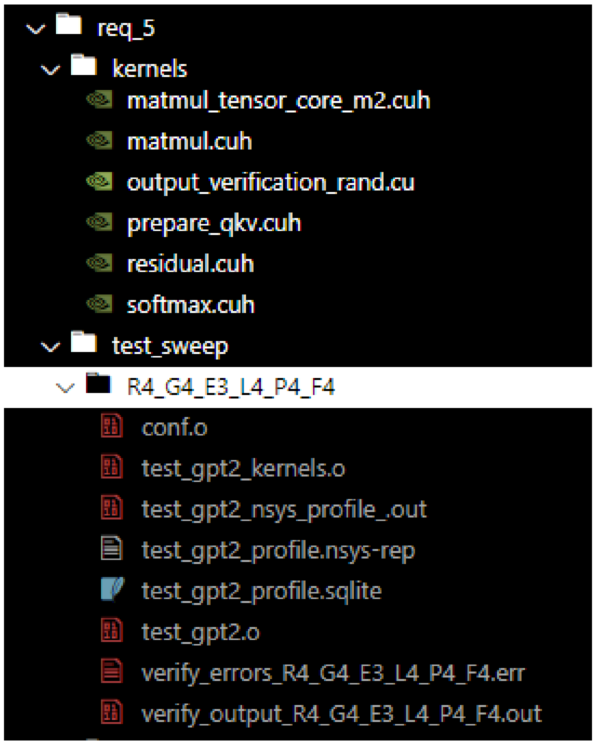
Implementation

The full script is contained in a python file called "config_sweep.py." The script can be called with a regular python3 call with no additional arguments required. There are three main steps in this configuration sweep script.

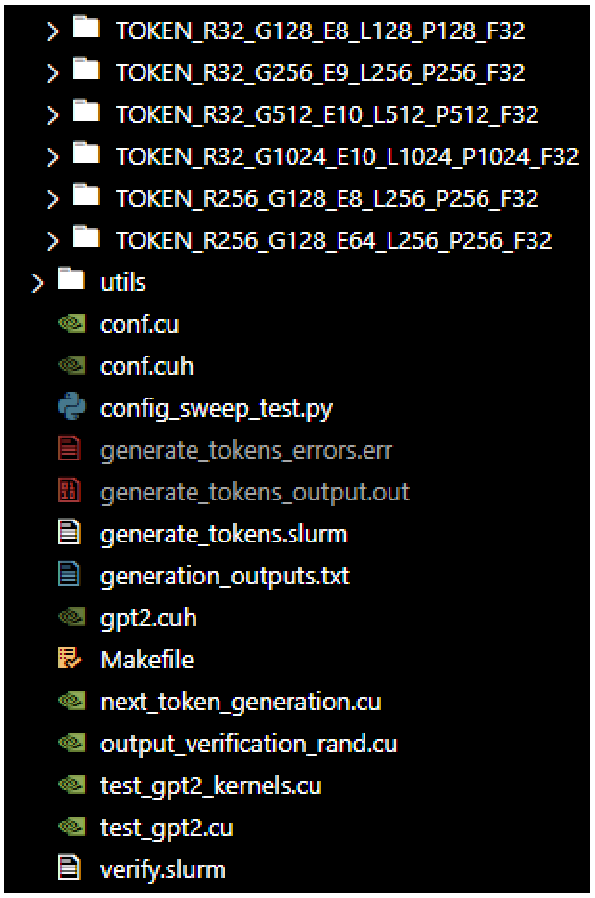
Step 1: Define sweep dimensions
Step 2: Uses input arguments to generate a slurm file
Step 3: Run the slurm job

For Step 1: A list of sweep dimensions is located at the top of the file. Each sweep dimension which is represented as a tuple is used to generate various files. Currently, the dimension ordering is set to specific variables, but in the future an enum can be used for easy mapping between indexes and variable names. Currently, there are 6 changing dimensions in the tuple in each sweep iteration RESIDUAL_BLK_DIM, GELU_BLOCK_SIZE, ENCODER_BLOCK_DIM, LAYERNORM_BLOCK_SIZE, PREPARE_BLOCK_SIZE, FP_BLOCK_DIM. The variables respectively for residual, gelu, encoder, layernorm, prepare out for adding bias for matmul, and floating point conversion kernels.

Step 2: To actually compile each kernel with new dimensions, each kernel uses a generated config file. After the user sets each sweep iteration, the script will automatically loop through the sweep and create a config file for the inputted dimensions. This script creates the "conf.cuh" and "conf.cu" files and saves them. Then the script will create a directory for the current sweep to run in. After that it will generate a Makefile with all compiled and assembled end locations set to the newly generated directory. This allows for each job to run independently and simultaneously. After generating the Makefile, the script will run "make clean && make all" to compile all required files. It will finally create the verify.slurm file and also set the new file paths to the generated directory, so it uses the currently compiled files and saves all results to the correct location. It will copy all generated files to the new working directory. An example of the created directory is shown below.



```
req_5
  kernels
    matmul_tensor_core_m2.cuh
    matmul.cuh
    output_verification_rand.cu
    prepare_qkv.cuh
    residual.cuh
    softmax.cuh
  test_sweep
    R4_G4_E3_L4_P4_F4
      conf.o
      test_gpt2_kernels.o
      test_gpt2_nsys_profile_.out
      test_gpt2_profile.nsys-rep
      test_gpt2_profile.sqlite
      test_gpt2.o
      verify_errors_R4_G4_E3_L4_P4_F4.err
      verify_output_R4_G4_E3_L4_P4_F4.out
```



```
    TOKEN_R32_G128_E8_L128_P128_F32
    TOKEN_R32_G256_E9_L256_P256_F32
    TOKEN_R32_G512_E10_L512_P512_F32
    TOKEN_R32_G1024_E10_L1024_P1024_F32
    TOKEN_R256_G128_E8_L256_P256_F32
    TOKEN_R256_G128_E64_L256_P256_F32
  utils
  conf.cu
  conf.cuh
  config_sweep_test.py
  generate_tokens_errors.err
  generate_tokens_output.out
  generate_tokens.slurm
  generation_outputs.txt
  gpt2.cuh
  Makefile
  next_token_generation.cu
  output_verification_rand.cu
  test_gpt2_kernels.cu
  test_gpt2.cu
  verify.slurm
```

Step 3: Once the script generates all required files and sets up the new folder, it will run "sbatch ./test_sweep/{file_path}/verify.slurm." This should submit the job and then within a couple minutes the profile outputs should show up in the generated working directory, The script also waits a couple seconds in between each job submission to allow for any latencies with copying or deletion.

Analysis

As mentioned above, the main dimensions that were tested were the block dimensions and thread counts. The variables that were changed were residual thread count, gelu thread count, encoder block dimension, layernorm thread count, and prepare out for matmul thread count. Changing the thread counts will also change the grid dimensions. Below is a table of all dimensions that were tested:

Residual thread count	Gelu thread count	Encoder block dimension	Layernorm thread count	Prepare out thread count
4	4	3	4	4
8	8	4	8	8
16	16	5	16	16
32	32	6	32	32
64	64	7	64	64
128	128	8	128	128
256	256	9	256	256
512	512	10	512	512

1024	1024	■	1024	1024

For the thread counts we started at 4 because we already know that having less would only lead to more inefficiency in both coalescing and increased launch times. The kernels that went up to 1024 had a 1d indexing scheme in the kernel, thus a max of 1024 threads which is the max threads in an SM for an A40 compute capability. Encoder was indexed three dimensionally so it was only tested from 3-10 as 10x10x10 would give 1000 threads. After figuring out these parameters, we guessed that the maximum threads closest to 1024 would be the most optimal as it would allow for maximum streaming multiprocessor occupancy, however, in almost every scenario this was not the case. We also ran the sweep on our most optimized fused kernels. Residualand gelu are both fused with the prepare out kernel which adds the bias values after the cublas matmul kernel. This is also tested without the KV cache. The sweep output from Nsight systems is shown below.

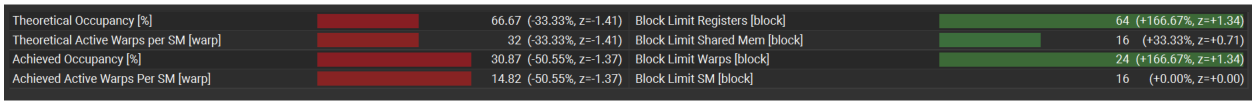
Thread count	Residual total kernel time (ns)	Gelu total kernel time (ns)	Layernorm total kernel time (ns)	Prepare out total kernel time (ns)
4	882,555	1,662,430	1,400,798	3,462,653
8	478,433	851,648	1,204,481	1,748,225
16	275,871	446,526	653,725	891774
32	177,281	244,127	383,038	279606
64	137,634	142,239	254,399	141189
128	132,672	96,833	195,682	168575
256	133,918	97,600	180,449	96042
512	139,041	103,811	256,734	165439

1024	177,024	154,751	533,955	183550

Block.x dimension	Encoder total kernel time (ns)
3	12,032
4	7,488
5	7,136
6	7,296
7	7,617
8	7,904
9	8,288
10	9,504

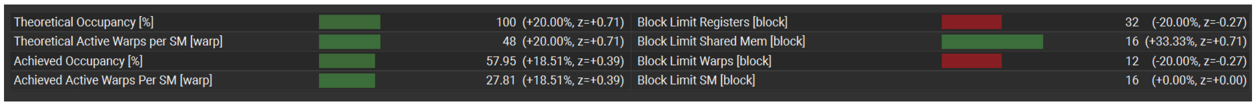
The most optimal time is highlighted on the table for every tested kernel. For both the residual and Gelu kernels 128 threads is best. For both the layernorm and the prepareout kernel 256 threads is best. For the encoder kernel having a 5x5x5 block dimension was also the best. As shown in all cases, the most optimal kernel dimension was never actually the highest possible thread count for the tested kernels. This went against our previous thinking that more threads equals more parallelism on the GPU, but this is not the case and more of a misconception. In the data, increasing threads past the optimal limit does the opposite of increasing speed, it decreases speed by a significant margin.

Since we changed the block dimensions between each kernel launch, we used Nsight compute to look at the occupancy to see why increasing threads does not equal more speed. We also specifically looked at the residual kernel for 64, 128, and 256 threads as this would show us when it peaked and started decreasing in efficiency again. In the occupancy portion of the report, we see that the theoretical occupancy of the 64 threads is at 66.67% of the maximum streaming multiprocessors per cycle and the achieved occupancy is even lower at 30.87%. Since we are not compute or memory bound, increasing this theoretical limit should allow us to increase the speed of the kernel. Now looking at that 128 thread report, the theoretical occupancy became 100% and the achieved occupancy was 57.95%. Lastly, for the 256 thread launch, it also had a theoretical occupancy of 100% and with an achieved occupancy of 66.92.



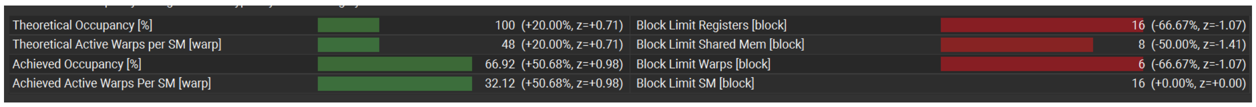
Theoretical Occupancy [%]	66.67 (-33.33%, z=-1.41)	Block Limit Registers [block]	64 (+166.67%, z=+1.34)
Theoretical Active Warps per SM [warp]	32 (-33.33%, z=-1.41)	Block Limit Shared Mem [block]	16 (+33.33%, z=+0.71)
Achieved Occupancy [%]	30.87 (-50.55%, z=-1.37)	Block Limit Warps [block]	24 (+166.67%, z=+1.34)
Achieved Active Warps Per SM [warp]	14.82 (-50.55%, z=-1.37)	Block Limit SM [block]	16 (+0.00%, z=+0.00)

Occupancy Metrics for 64 Threads per Block



Theoretical Occupancy [%]	100 (+20.00%, z=+0.71)	Block Limit Registers [block]	32 (-20.00%, z=-0.27)
Theoretical Active Warps per SM [warp]	48 (+20.00%, z=+0.71)	Block Limit Shared Mem [block]	16 (+33.33%, z=+0.71)
Achieved Occupancy [%]	57.95 (+18.51%, z=+0.39)	Block Limit Warps [block]	12 (-20.00%, z=-0.27)
Achieved Active Warps Per SM [warp]	27.81 (+18.51%, z=+0.39)	Block Limit SM [block]	16 (+0.00%, z=+0.00)

Occupancy Metrics for 128 Threads per Block

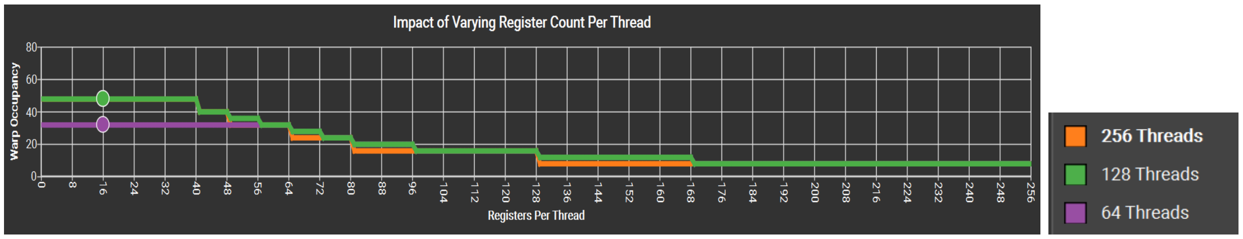


Theoretical Occupancy [%]	100 (+20.00%, z=+0.71)	Block Limit Registers [block]	16 (-66.67%, z=-1.07)
Theoretical Active Warps per SM [warp]	48 (+20.00%, z=+0.71)	Block Limit Shared Mem [block]	8 (-50.00%, z=-1.41)
Achieved Occupancy [%]	66.92 (+50.68%, z=+0.98)	Block Limit Warps [block]	6 (-66.67%, z=-1.07)
Achieved Active Warps Per SM [warp]	32.12 (+50.68%, z=+0.98)	Block Limit SM [block]	16 (+0.00%, z=+0.00)

Occupancy Metrics for 256 Threads per Block

Even though the 256 thread launch has more achieved occupancy compared to the 128 thread launch, due to block limits and register utilization, there is no actual speed up with the kernel. We can see that for 64 thread launches there are block limit registers at 64 registers per thread, for 128 threads there is a 32 block limit register, and for 256 there are 16 block limit registers. The register count per thread is a limiting factor.

In the figure below, we see that the 128 thread launch has always higher warp occupancy compared to the rest of the launches. With fewer threads, more blocks can fit on each SM as less registers are used for each SM and thus more calculations are happening concurrently. There will also be more concurrency as there will be less warp stalling as less latency hiding is required to run more threads without issues. This led to the 128 thread launch having the highest compute and memory throughput allowing it to be the fastest launch.



Impact of Varying Register Count Per Thread for 256, 128, and 64 Threads

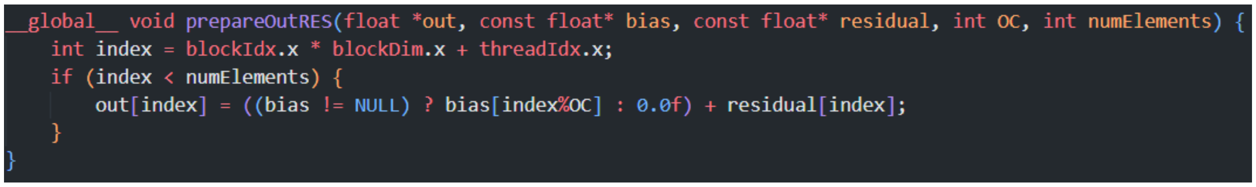
Fused Kernels (Prop 3)

Background

When analyzing the kernels, we noticed two of the kernels that were computationally lightweight, which were gelu and residual. In addition, when looking at the forward pass of the gpt network in the gpt2.cuh file, we noticed that these two kernels came after a matmul kernel. For such a simple kernel, the additional overhead of launching the kernel wasn't necessary, and wanted to fuse the kernels to remove some of the kernel launch overhead.

Implementation

Since the matmul kernel had an additional kernel, prepareOut, to add in the bias for the output matrix, we decided to fuse gelu and residual into that kernel. To fuse the kernels, we created two separate fused matmul kernels, one for each fusion. In the prepareOut kernel, we just added the gelu and residual operations into the kernel. We had to change the argument headers in the matmul, prepareOut, and when we called the matmul in gpt2.cuh to account for the fused kernels.

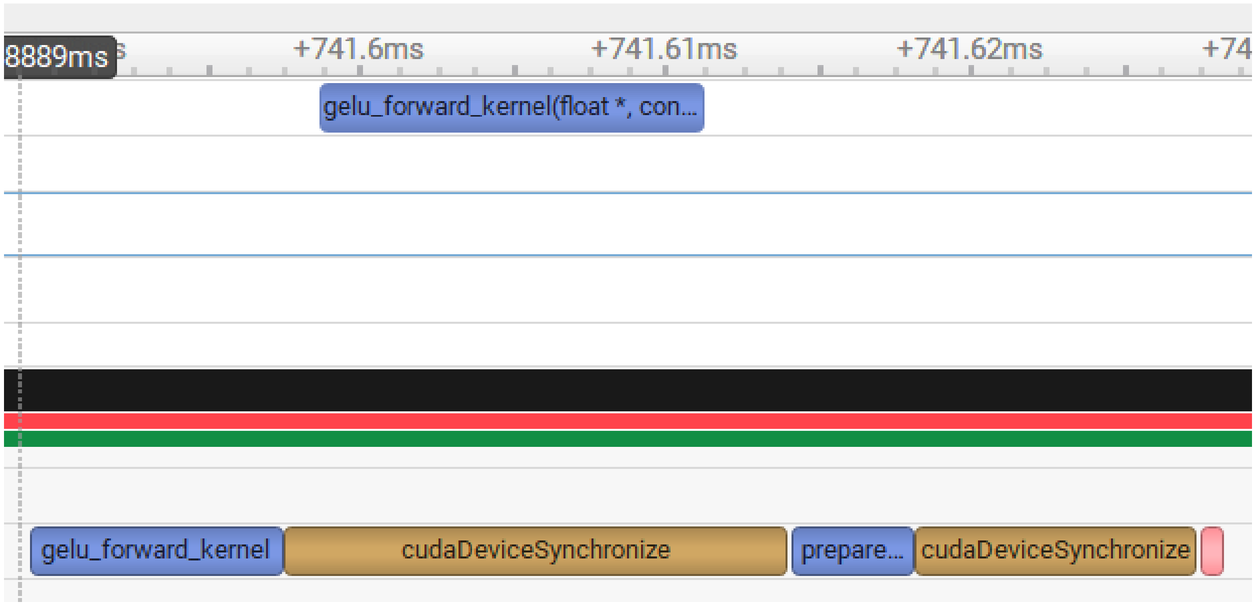


```
__global__ void prepareOutRES(float *out, const float* bias, const float* residual, int OC, int numElements) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < numElements) {
        out[index] = ((bias != NULL) ? bias[index%OC] : 0.0f) + residual[index];
    }
}
```

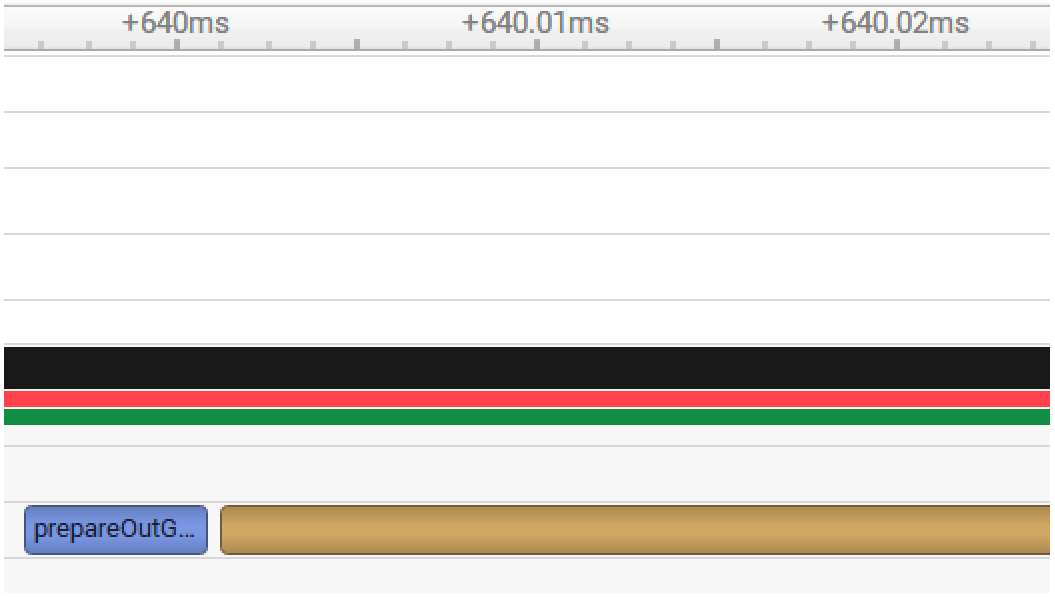
The one thing we had to consider was when the prepareOut kernel was called in the matmul method for the gelu fusion. Because the gelu function should not apply to the inputs of the matrix, the prepareOut kernel was instead called after the matrix multiplication so the gelu function was called on the right value.

Analysis

Looking at the Nsight Systems Report, we noticed a decrease in execution and launch time by fusing the residual and gelu kernels with prepareOut. When looking at the timeline of kernel calls, we can see that by having separate calls between gelu or residual and prepareOut, the system needs to launch each kernel separately. For example, one kernel launch of the gelu function takes around 8.195 µs, and the subsequent prepareOut kernel takes 3.867 µs to launch. When the two kernels are fused together, the kernel launch time takes 4.91 µs, which is a 2.45x speed up of kernel launch times. In addition, fusing kernels also decreased the total execution time. Looking at the gelu example again, the gelu kernel takes 12576 µs to execute and the prepareOut kernel takes 5.12 µs, while the fused prepareOutGelu takes 12.416 µs to execute. By fusing together kernels, the launch overhead can be reduced, which can reduce overall execution since matmul-gelu and matmul-residual is present multiple times in our gpt network, and can reduce execution time by performing the tasks of two kernels at the same time.



Without Kernel Fusion



With Kernel Fusion

KV Cache (Prop 1)

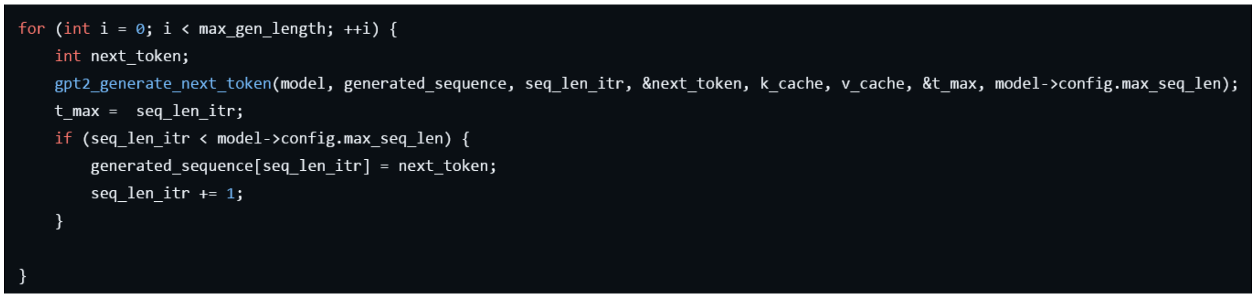
Background

To generate tokens, GPT 2 needs to run through its forward pass for each new token. This is due to the forward pass of GPT 2 outputting vocab scores for the next token, based on all the previous tokens. Essentially, the first forward pass will just run the forward pass on all the prompt tokens and produce the first token of the response. The next forward pass would use the prompt tokens and first output token as inputs. The input size would keep successively increasing by one until the terminating token is generated. There is a fundamental redundancy with this, since we are repeating computations on the earlier tokens. A key redundancy lies with how attention is computed. Attention takes a query, key, and value matrix for its computations. These three matrices are created by a linear transformation (matmul) on the input embeddings with a attention weights matrix. The query matrix is specifically based on the current token, so there is no redundancy in its creation. However, the key and value matrices represent relationships between the current token and past tokens relationships. Since we already know previous tokens relationships from the previous forward passes, it is redundant to compute the entire K and V matrices again, since we have most of that information and just need to find the new tokens contribution. This is where the idea of a KV cache comes from. By caching the K and V matrices for a forward pass, the subsequent forward pass can load in these previous computed values, only needing to compute the new row contributed by the current token. This drastically reduces the number of computations needed for each forward pass after the first, which increases the number of tokens generated per second. A clear downside is the additional memory storage needed for the cache, and the overhead from loads/stores to the cache. However, due to the speedup gained from the large amount of redundant computations skipped, some form of a KV cache is utilized in almost every modern LLM.

Kernel Implementation

For our key value cache implementation, we edited two main files the "next_token_generation.cu" and the "gpt2.cuh" files. One of the most important parts of the KV cache is the dimensions. Each key and value matrix is a separate matrix of size model->config.num_layers * SEQUENCE_GENERATION_BATCH_SIZE * model->config.max_seq_len * model->config.channels * sizeof(float). The number of layers is the number of layers in the mode. That is important because each layer must have a different key value matrix. The sequence generation batch size is simply the B size mentioned previously. The max_seq_len is the maximum context length of the output of the GPT2 model. Then we have the channels which are the embedding size of each token. After defining both the key and value of the specified size, we must also account for one more thing. We need to save the t_max value which tells which time step we are in the forward pass of GPT2. This is an important variable as it tells whether or not we need to calculate more of the key value matrix.

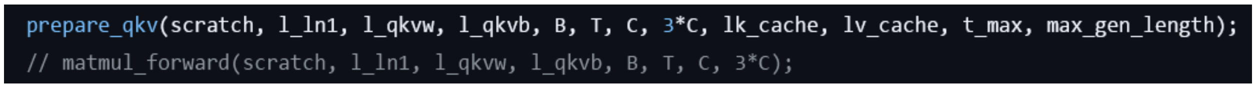
Now to actually start the pass we go through a for loop of the maximum possible generation length:



```
for (int i = 0; i < max_gen_length; ++i) {
    int next_token;
    gpt2_generate_next_token(model, generated_sequence, seq_len_itr, &next_token, k_cache, v_cache, &t_max, model->config.max_seq_len);
    t_max =  seq_len_itr;
    if (seq_len_itr < model->config.max_seq_len) {
        generated_sequence[seq_len_itr] = next_token;
        seq_len_itr += 1;
    }

}
```

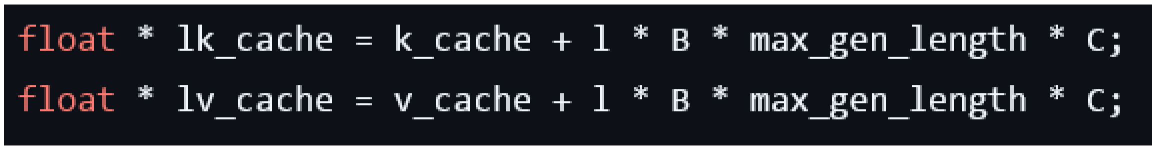
In this code we essentially generate each next token, and set the new t_max value so that we do not recalculate already calculated KV matrices.

After a series of calls we eventually get to the actual model in the gpt2_forward method. Instead of redoing the matmul we simply do a prepare_qkv call which will prepare the QK matrix for attention.



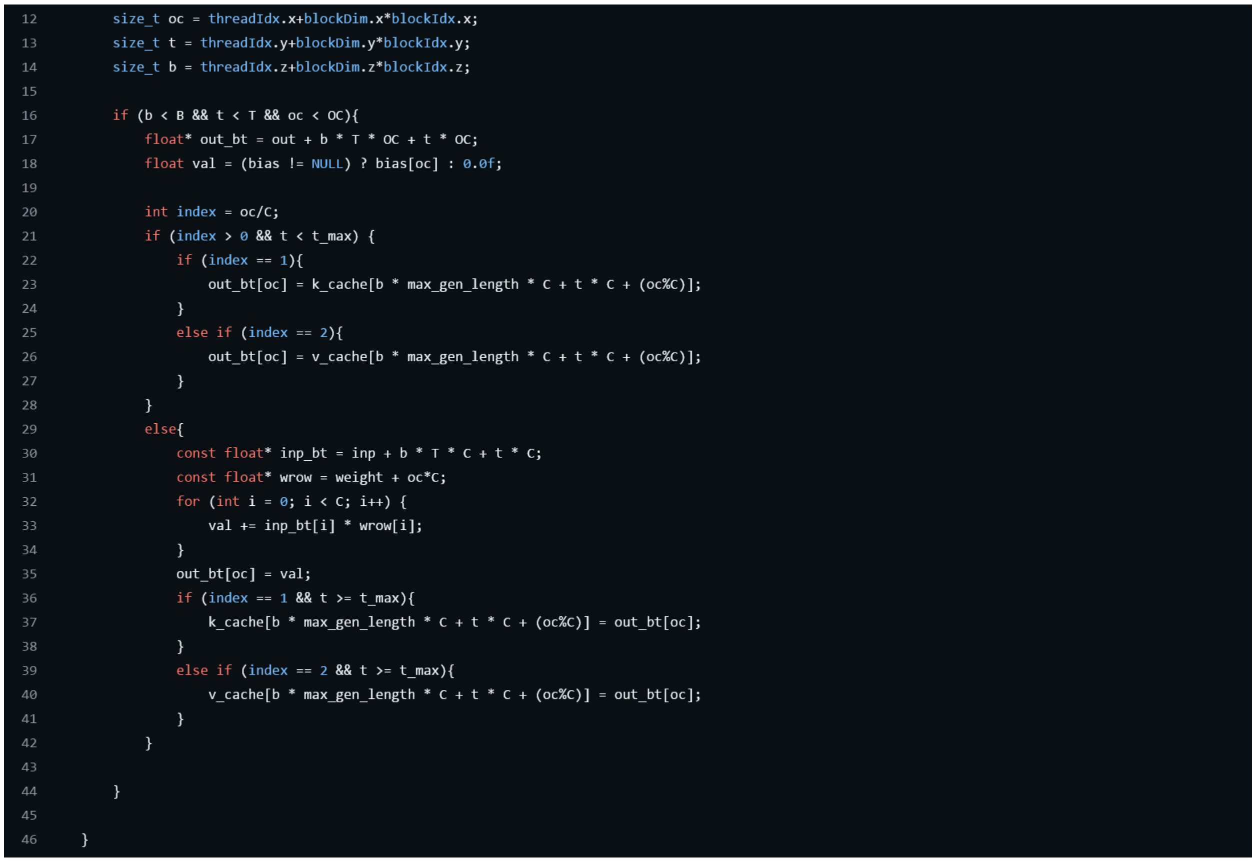
```
prepare_qkv(scratch, l_ln1, l_qkvw, l_qkvb, B, T, C, 3*C, lk_cache, lv_cache, t_max, max_gen_length);
// matmul_forward(scratch, l_ln1, l_qkvw, l_qkvb, B, T, C, 3*C);
```

We also define the proper K and V cache indexes. For each call we need to account for the current model layer * batch * maximum token length * embedded token size as each subunit in the cache is separated by a full layer allowing for coallessing.



```
float * lk_cache = k_cache + l * B * max_gen_length * C;
float * lv_cache = v_cache + l * B * max_gen_length * C;
```

Finally, we get to the actual prepare_qkv kernel where the KV cache is prepared.

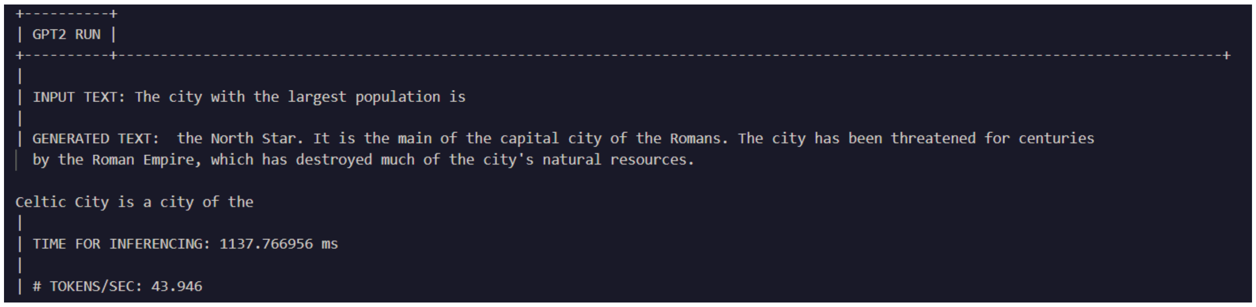


```
        size_t oc = threadIdx.x+blockDim.x*blockIdx.x;
        size_t t = threadIdx.y+blockDim.y*blockIdx.y;
        size_t b = threadIdx.z+blockDim.z*blockIdx.z;

        if (b < B && t < T && oc < OC){
            float* out_bt = out + b * T * OC + t * OC;
            float val = (bias != NULL) ? bias[oc] : 0.0f;

            int index = oc/C;
            if (index > 0 && t < t_max) {
                if (index == 1){
                    out_bt[oc] = k_cache[b * max_gen_length * C + t * C + (oc%C)];
                }
                else if (index == 2){
                    out_bt[oc] = v_cache[b * max_gen_length * C + t * C + (oc%C)];
                }
            }
            else{
                const float* inp_bt = inp + b * T * C + t * C;
                const float* wrow = weight + oc*C;
                for (int i = 0; i < C; i++) {
                    val += inp_bt[i] * wrow[i];
                }
                out_bt[oc] = val;
                if (index == 1 && t >= t_max){
                    k_cache[b * max_gen_length * C + t * C + (oc%C)] = out_bt[oc];
                }
                else if (index == 2 && t >= t_max){
                    v_cache[b * max_gen_length * C + t * C + (oc%C)] = out_bt[oc];
                }
            }

        }

    }
```

The KV cache utilizes a simple matrix multiplication using 3d block dimensions. The main changes we made were how we indexed bases on the output channel to map towards either the key or value cache. If the output channel / channel gives one and the t value is less than the current maximum t values stored in the cache, then it should map towards the key cache, if it maps towards two and t < t_max, then it should map towards the value cache. An oc/c value of zero and values of t > t_max will be manually calculated using matrix multiplication. If oc/c is zero, then that will be directly used for the query matrix. If oc/c is not zero, then the calculated value will be stored in the K or V cache, depending on if oc/c is equal to 1 or 0.

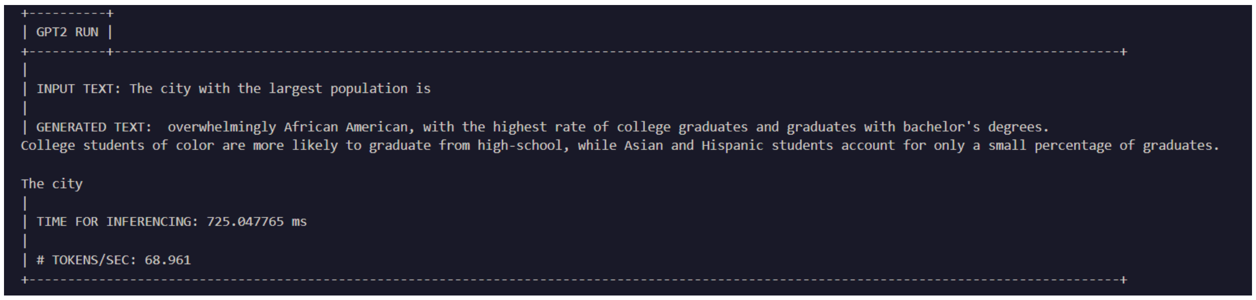
Analysis

Empirically, we saw faster token generation speeds with the KV cache. Below is a comparison between a GPT-2 run with KV cache off and on:



```
+----------+
| GPT2 RUN |
+----------+-----------------------------------------------------------------------------------------------------------------+
|
| INPUT TEXT: The city with the largest population is
|
| GENERATED TEXT:  the North Star. It is the main of the capital city of the Romans. The city has been threatened for centuries
| by the Roman Empire, which has destroyed much of the city's natural resources.

Celtic City is a city of the
|
| TIME FOR INFERENCING: 1137.766956 ms
|
| # TOKENS/SEC: 43.946
```

KV Cache off (43.946 tokens/second)



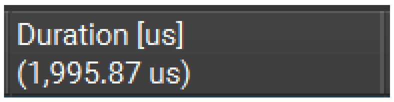
```
+----------+
| GPT2 RUN |
+----------+-----------------------------------------------------------------------------------------------------------------+
|
| INPUT TEXT: The city with the largest population is
|
| GENERATED TEXT:  overwhelmingly African American, with the highest rate of college graduates and graduates with bachelor's degrees.
College students of color are more likely to graduate from high-school, while Asian and Hispanic students account for only a small percentage of graduates.

The city
|
| TIME FOR INFERENCING: 725.047765 ms
|
| # TOKENS/SEC: 68.961
+--------------------------------------------------------------------------------------------------------------------------------+
```

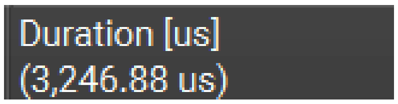
KV cache on (68.961 tokens/second)

Overall, our KV cache resulted in a 57% speedup for token generation.

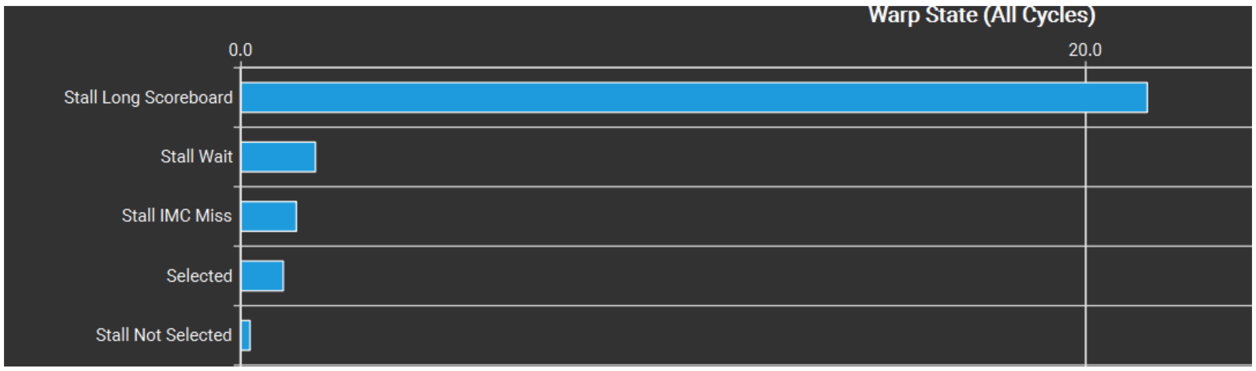
Profiling with the KV cache was a challenging task. A regular profiling job only yields information about the first forward pass. It would make sense that the first forward pass will actually be slower with KV cache. This was what we determined through comparing two forward passes with and without the KV cache.

The implementation without the KV cache had a total runtime of about 2 ms while using the KV cache, the total runtime was 60% greater, at 3.2 ms.

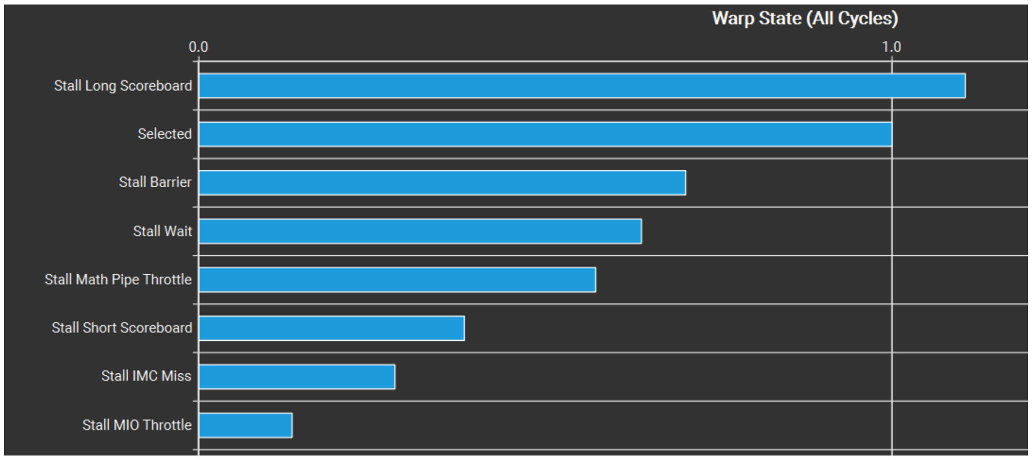

Duration [us]
(1,995.87 us)


Duration [us]
(3,246.88 us)

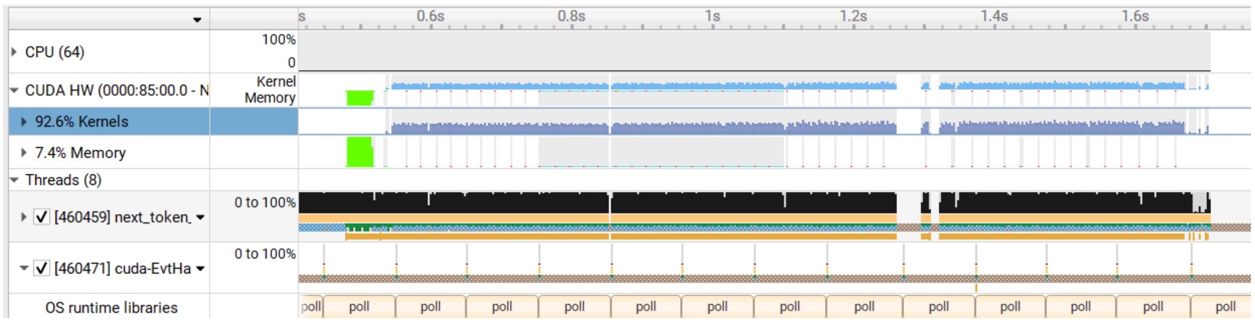
This makes sense, because in the first forward pass, the prepare QKV kernel will not only have to perform the matmul and store to the output, but also store to the global K and V caches. Our hypothesis was verified, because Nsight Compute detected that warps stalled an average of 21.5 cycles waiting on global memory reads in the prepareQKV kernel.



Compared to this, the standard matrix projection, which used CUBLAS, had almost zero warp stalling:



Just by looking at the compute report, it would be hard to see why the KV cache is causing a token speedup. To get a deeper insight into what was going on, our team ran a custom test with multiple forward passes using the QKV cache, profiled with Nsight Systems. We encountered challenges with this, because the profiling often ended after the second pass due to events getting lost after a certain time, which we suspected was because of too many events being profiled. However, even from the first two passes, we noticed an interesting observation.

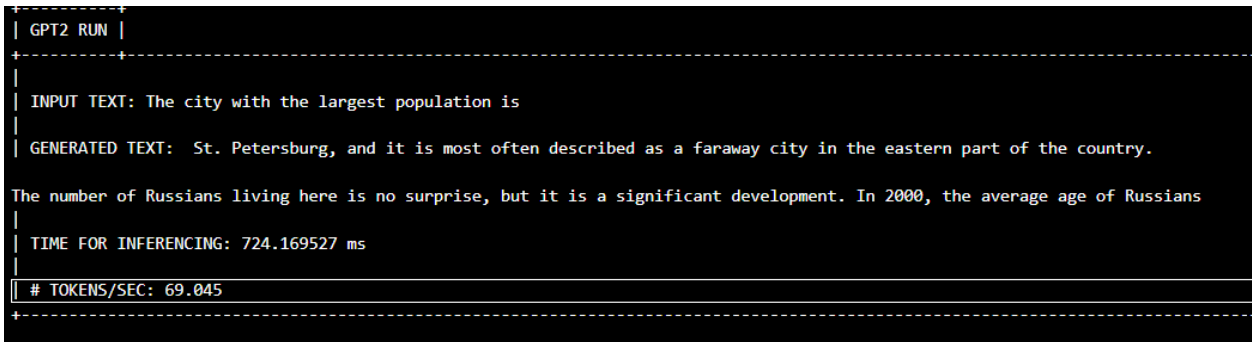


The first forward pass, started at about 0.53s and ended at about 1.27s. The second forward pass started at about 1.3s and ended at about 1.7s. From this, we can infer that the first forward pass took close to 0.74s while the second took about 0.4s. The second pass was about 46% faster. (Note: the discrepancy between this and the earlier runtime is mostly due to the fact that we used a much slower version of the matmul kernel here, for other, unrelated testing).

This is likely because for every pass after the first pass, since all the input tokens have already been processed, many multiply operations had to be done, reducing the additional number of stores to global memory. This means that even though on paper, the prepare QKV kernel is slower than a normal matrix multiplication, this is only through the first pass. This kernel will outpace the matmul kernels for future passes. Of course, since we had problems with profiling, it is difficult to get more definitive metrics to support this claim. For next steps, we will work around the system profiling problem by reducing the sample rate and determine how to reduce global loads to the cache to reduce overhead at the beginning.

Conclusion

Overall, our greatest improvements came from the flash attention, KV cache, tensor core, and kernel tuning with batch sweeping optimizations. All of our fastest kernels combined with each other led to a speed of almost 70 tokens/second! This is overall 14x faster performance than the original GPU baseline, which only ran at 5 tokens/second.



```
+----------+
| GPT2 RUN |
+----------+--------------------------------------------------------------------------------------------------------------
|
| INPUT TEXT: The city with the largest population is
|
| GENERATED TEXT:  St. Petersburg, and it is most often described as a faraway city in the eastern part of the country.

The number of Russians living here is no surprise, but it is a significant development. In 2000, the average age of Russians
|
| TIME FOR INFERENCING: 724.169527 ms
|
| # TOKENS/SEC: 69.045
+-------------------------------------------------------------------------------------------------------------------------
```

While we are proud of our achievement, there is still room for improvement. In the next steps of this project we may explore a number of exciting areas that promise potential speedups. Some of these include:

- Various techniques to improve matrix multiplication performance (mentioned above)
- More advanced KV caching techniques spanning across additional kernels like attention
- Speculative decoding techniques using a draft model or other estimation functions.
- Additional kernel fusion techniques (we didn’t mention the details here but we did do some initial fused kernel implementations for gelu+matmul and residual+matmul).
- Experiments with Flash Attention 3 and other attention optimization techniques that can be done using the Hopper Architecture and above.
- Deep dives into PTX
- Experiments with streams

References

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[2] https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/
[3] https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-reference
[4] https://arxiv.org/pdf/2004.05150v2
[5] https://huggingface.co/docs/transformers/en/model_doc/gpt2
[6] https://arxiv.org/pdf/2205.14135
[7] https://tridao.me/publications/flash2/flash2.pdf
[8] https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[9]

https://stackoverflow.com/questions/44704506/limiting-register-usage-in-cuda-launch-bounds-vs-maxrregcount

[10]

https://alexarmbr.github.io/2024/08/10/How-To-Write-A-Fast-Matrix-Multiplication-From-Scratch-With-Tensor-Cores.html

[11] https://siboehm.com/articles/22/CUDA-MMM

[12] https://medium.com/data-science/matrix-multiplication-on-the-gpu-e920e50207a8

[13] https://arxiv.org/abs/2408.05646

[14] https://arxiv.org/pdf/1811.09736

[15] https://arxiv.org/abs/1706.03762

[16] https://arxiv.org/pdf/2301.03598

[17] https://arxiv.org/pdf/2308.15152