GPT-2 Final Report

LLM Inference Scaling

Parallel Attentionists

Luke You Minseob Shin Kevin Peercy Luke Kwiatkowski

Introduction 1
Proposed Optimizations Locations 2
Baseline Profiling Report 2
Constant Memory 3
Configuration Sweep/Optimization 5
Restrict 14
Kernel Fusion 15
Flash Attention 18
Local/Windowed Attention 21
NVIDIA CUTLASS 23
KV Cache 27
Split-K 31
CUDA Streams 33
Conclusion and Future Directions 35

Introduction

LLMs have skyrocketed in mainstream utility and focus from the entire world due to exponential growth of model sizes and training applicability. The focus on these systems has extended to pretty much every field in hardware, but none are more central to the core of the system as GPU architecture. Abusing parallelization that GPU designs provide, through CUDA programming for us, lets you simultaneously evaluate countless tokens and inputs to generate next token output through GPT models.

Our final project focuses on breaking down the GPT-2 architecture and scrutinizing points of optimization for the best performance achievable. The main kernels we develop for our decoder-only transformer to run are Attention, Matrix Multiply, Layernorm, and Softmax.

In our last report, we proposed the following optimizations as our self-driven speedups: Kernel Fusion, Cuda Streams, CutLASS, and KV Cache. The report covers our experience implementing those and the results on our runs due to that work, and also how to run those advanced kernels for the graders. We also cover 6 required optimizations, Flash Attention, Configuration Sweep, Local Attention, Split-K, Restrict, and Constant Memory. Detailed below are summaries of the optimization, theoretically what we are looking to achieve from each of them, and the results of profiling alongside some analysis of what we actually observed,

Proposed Optimizations Locations

Kernel Fusion is in the branch “Kevin_branch”, and the gpt2.cuh file has been modified to include and utilize the correct kernel, with instructions on how to run it in the section discussing the optimization

KV Cache is in the “KV_Cache” branch, and instructions to run it are below

CUDA Streams is in the “luke11” branch and instructions listed below

CUTLASS is in the “lukeyou” branch and instructions below

Baseline Profiling Report

We utilized NVIDIA A40

```
Name:                          NVIDIA A40
Total global memory:           47608692736
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    1740000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     84
```

Figure 1. NVIDIA A40 Resources

CUDA API Summary

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
59.0	110,377,868	115	959,807.5	210,692.0	1,002	25,960,966	2,481,393.7	cudaDeviceSynchronize
34.2	63,935,586	3	21,311,862.0	8,298,968.0	19,827	55,616,791	29,995,952.4	cudaMemcpy
2.7	5,096,709	12	424,725.8	5,816.5	201	1,800,141	616,148.6	cudaFree
1.3	2,472,993	1	2,472,993.0	2,472,993.0	2,472,993	2,472,993	0.0	cudaFreeHost
1.0	1,868,579	1	1,868,579.0	1,868,579.0	1,868,579	1,868,579	0.0	cudaGetDeviceProperties_v2_v12000
0.6	1,214,832	1	1,214,832.0	1,214,832.0	1,214,832	1,214,832	0.0	cudaMallocHost
0.6	1,053,698	171	6,162.0	3,136.0	2,754	336,126	25,740.6	cudaLaunchKernel
0.4	742,995	6	123,832.5	114,834.0	3,457	315,878	117,419.2	cudaMalloc
0.1	190,957	810	235.7	220.0	100	741	97.6	cuGetProcAddress_v2
0.0	42,038	1	42,038.0	42,038.0	42,038	42,038	0.0	cudaMemset
0.0	24,548	18	1,363.8	386.0	310	11,531	2,740.5	cudaEventCreateWithFlags
0.0	8,596	18	477.6	366.0	250	2,084	423.4	cudaEventDestroy
0.0	3,918	3	1,306.0	1,353.0	1,142	1,423	146.3	cuInit
0.0	2,676	3	892.0	311.0	181	2,184	1,120.8	cuModuleGetLoadingMode
0.0	861	2	430.5	430.5	290	571	198.7	cudaGetDriverEntryPoint_v11030

Figure 2. Baseline API Times

CUDA GPU Memory Time Summary

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
86.6	55,529,666	2	27,764,833.0	27,764,833.0	864	55,528,802	39,264,181.5	[CUDA memcpy Host-to-Device]
12.8	8,203,836	1	8,203,836.0	8,203,836.0	8,203,836	8,203,836	0.0	[CUDA memcpy Device-to-Host]
0.6	406,144	1	406,144.0	406,144.0	406,144	406,144	0.0	[CUDA memset]

Figure 3. Baseline Memory Times

CUDA GPU Kernel Summary

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
78.3	86,203,508	49	1,759,255.3	1,610,399.0	428,864	25,956,529	3,565,786.9	matmul_forward_kernel
7.6	8,386,205	12	698,850.4	698,799.5	697,984	700,543	768.7	att_kernel
7.6	8,373,783	12	697,815.3	697,599.0	696,063	700,991	1,602.4	preatt_kernel
4.7	5,131,130	25	205,245.2	205,696.0	190,944	211,776	3,742.3	layernorm_forward_kernel
1.0	1,095,840	12	91,320.0	91,536.0	89,792	92,256	798.1	permute_kernel
0.3	367,935	12	30,661.3	30,656.0	30,399	30,944	144.5	softmax_forward_kernel
0.3	340,736	12	28,394.7	28,352.0	28,224	28,544	88.8	unpermute_kernel
0.1	96,641	24	4,026.7	4,064.5	3,521	4,544	396.6	residual_forward_kernel
0.1	95,489	12	7,957.4	7,936.0	7,808	8,128	111.2	gelu_forward_kernel
0.0	6,688	1	6,688.0	6,688.0	6,688	6,688	0.0	encoder_forward_kernel

Figure 4. Baseline Kernel Times

The most time-consuming kernel in our baseline implementation is the matmul_forward_kernel. This is due to the overhead of the memory operations caused by the bottleneck in the memory access pattern of directly accessing the global memory.

Since the matmul_forward_kernel takes a significant amount of the total time, we will go more in depth to analyze why that is.

Constant Memory

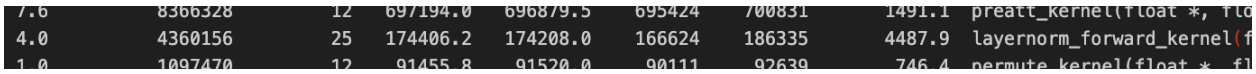
Constant memory is an extremely valuable, and extremely limited resource we could utilize for any static device input. Our maximum capacity in this memory format is only 64 KB, so we cannot just initialize every token and every input into constant mem as we would in a perfect world. The real upside of constant memory is the ability to access it quickly after it is loaded from global memory, and we will see the most performance gain when the loaded memory is reused a lot. The candidates for this optimization

kernel-wise therefore eliminates GELU, since we only use the input from launch once; Encoder, as weights are only used once per batch; and residual, as we only utilized the inputs once again. Attention also benefits minimally from constant memory as all inputs are in multi-dimension arrays that exceed our capacity in any useful case.

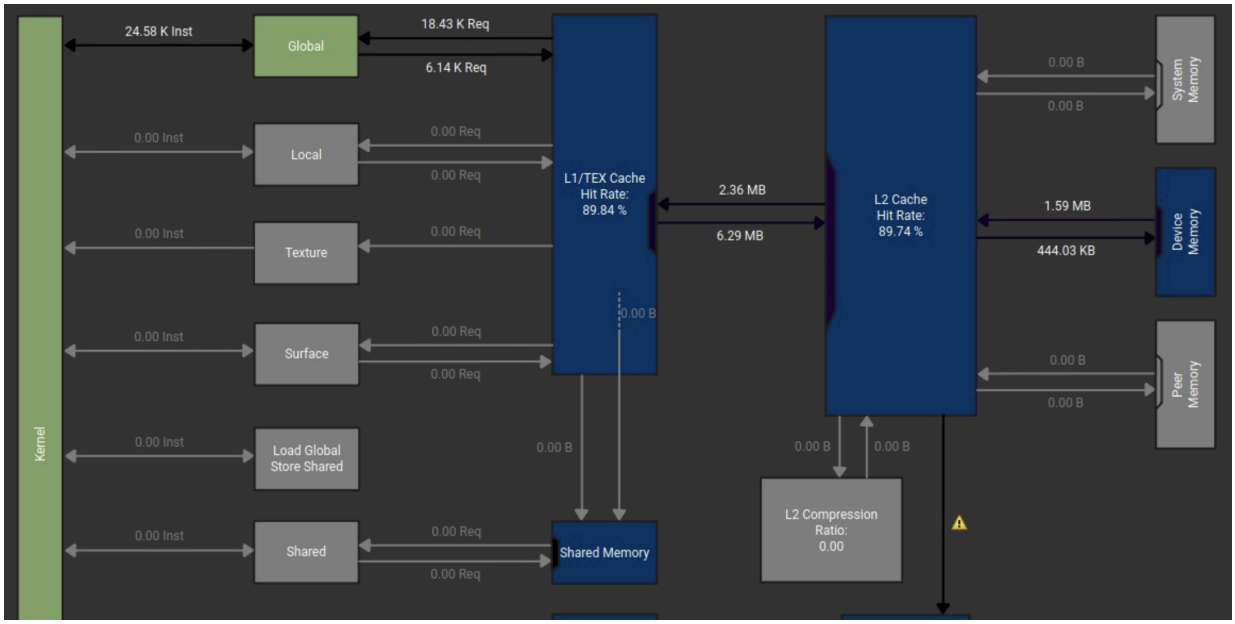
Layernorm is the best kernel to apply constant memory on, as our bias and weight input arrays are both static, and one dimension of length C. Matmul reuses the 1-D OC long array bias consistently which also allows for a constant memory load on the host side, any other input is a 2-D array like weights being 768 x 3*768 which is way over the constant memory limit..

We implemented this memory optimization on top of our milestone one kernels by loading the before-mentioned inputs in the host code from global memory into a hard-coded maximum capacity constant memory array, and pulling the values from that array whenever our indexes in the kernels are below the memory limit of 16K floats broken up by the number of arrays we loaded.

We predict that the overhead host memory load will be worth it when our reuse is as high as it is in the GPT script, and our memory bottlenecks from constant global loads will go down as they are replaced by the speedy constant memory loads.



Above is our run time of layernorm_forward_kernel, which was the heaviest user of constant memory. The speedup is around 20% compared to the ~5200000 time to run the 25 launches of layernorm from our baseline, which is solid. This is the exact same kernel, just with the two inputs mentioned previously loaded into constant memory first. This result reveals the capacity that quicker memory access time has on full kernel runtime, as the fast constant memory access after the first overhead sped up the entire process over just an additional few lines of code. The high layernorm reuse of the C long bias/weight arrays we loaded into constant memory was clearly sufficient enough to warrant this constant memory load, and benefit from it.



In the above table which represents memory transfer in our layernorm kernel, we observe a significant hit rate of around 90% in the L1/TEX Cache, which is where constant memory would be located after loading. The transfer sizes are small which is a perfect candidate for constant memory and shows why it was so effective in running this kernel. Normally our L2 cache takes the brunt of the load disparity, as our L1 cache is minimal and replaced easily, but placing our loads into the allocated constant memory signalled to our hardware to keep that data on hand and easily accessible.

Analyzing the other profiling results, our constant memory had minimal effect in our matmul runs, as the bias addition is done only once per input vector, while the bulk of the computation and data transfer is global loads for the input and tokens which are done K times per output value (each thread in our case). That timing disparity virtually nullifies the gains we receive from loading bias into constant memory and describes why our run time was no different for matmul. We interestingly found it to actually be slightly worse in our use case, likely because the overhead of constant memory load was completely worthless to us as our bias vector is loaded per-thread in a coalesced manner right at the start of the kernel. The max we could have saved from constant memory is T reuses of each bias[OC-dimension], and this is both minimal with a small T, and solving a very minor issue when our loads are naturally coalesced in the milestone 1 case as mentioned.

Constant memory has proven to be very limited in its use cases, but the few times you can find applications such as convolution masks or, in our case, layer normalizing, it is majorly effective and worth the slight bit more attention to detail. Be careful with its use, as some cases even see a performance loss if the reuse efficiency of constant memory can’t justify its additional overhead.

Configuration Sweep/Optimization

Although we have implemented and profiled a variety of optimizations for our kernels in milestone 2, there are many different factors that can affect the performance of each optimization which can make it harder to gauge their true effectiveness. Things like threadblock sizes, loop unrolling, and thread

coarsening can all affect the runtimes of our kernels, oftentimes in different ways across different datasets. This is why it's important to parameterize and sweep these factors in order to determine the best configurations for the datasets we wish to target. In our case, we swept configurations using test_gpt2 as a benchmark, but the sweep script we developed is applicable to any dataset which makes it a versatile and vital tool to help improve performance.

For our configuration sweep, we developed a few template kernels using the best fit optimizations from the previous milestone, and then wrote a bash script in order to apply our test parameters and profile them. To be more specific, we believed the best optimizations to sweep were the ones that were among the best-performing and most parameterizable. Our reduction version of the layernorm and softmax kernels were the only optimized versions of those kernels, so we included those. But for attention and matmul, we had three different optimizations we could choose from: joint register and shared memory tiling, tensor cores, and cuBLAS. Although cuBLAS was by far the best performing, it is also the most difficult to sweep because the library abstracts many of the finer details away from us, making it a no-go. Tensor cores is a valid candidate, but it can still be somewhat tricky to parameterize due to hardware-imposed restrictions on things like matrix/tile sizes. And finally, joint register and shared memory tiling wasn't far behind tensor cores in terms of performance, and it's also the most easily parameterizable. Thus, we settled on using the joint register and shared memory tiling versions of our matmul and attention kernels.

We swept a total of 15 different configurations across 9 iterations of profiling. This was done by sweeping the block sizes for all 3 dimensions across 3 iterations, then the unroll factors across 3 iterations, then thread coarsening factors across 3 iterations. We made sure to use default values when a parameter was not being swept in order to help isolate the changes. Additionally, we didn't apply parameters if it would break the kernel (e.g., block size for the reduction version of softmax has to be kept the same as a warp size). The exact values we used are shown below:

```
block_dims_1d=(128 512 1024)
block_dims_2d=(4 8 32)
block_dims_3d=(2 8 10)
unroll_factors=(4 16 64)
coarse_factors=(1 8 16)
```

We'll start with the 3 block configuration sweeps. Going into this, we expected to see a general improvement in performance as block sizes increased. We believe the kernels we selected did not use an excessive amount of shared memory or registers, so there shouldn't be a large detriment to using larger block sizes in terms of things like register spillage. As we profiled the kernels, we found the following:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
98.8	320858891	49	6548140.6	6100389.0	1563618	99347379	13661868.7	matmul_forward_kernel
0.5	1604066	12	133672.2	133696.0	133472	133857	121.5	att_kernel
0.4	1453024	12	121085.3	121072.0	120960	121216	81.3	preatt_kernel
0.1	201216	12	16768.0	16768.0	16608	16896	70.9	permute_kernel
0.0	153024	25	6121.0	5952.0	5472	7488	498.6	layernorm_forward_kernel
0.0	118784	12	9898.7	9904.0	9856	9952	41.7	unpermute_kernel
0.0	100032	24	4168.0	4176.0	3488	4896	606.1	residual_forward_kernel
0.0	92385	12	7698.8	7680.0	7584	7809	80.2	gelu_forward_kernel
0.0	58400	12	4866.7	4848.0	4800	5024	63.2	softmax_forward_kernel
0.0	19648	1	19648.0	19648.0	19648	19648	0.0	encoder_forward_kernel

1d block dim: 128, 2d block dim: 4x4, 3d block dim: 2x2x2, unroll factor: compiler, coarse_factor: 8

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
98.4	102876472	49	2099519.8	1924122.0	540479	30912295	4243571.2	matmul_forward_kernel
0.5	572831	12	47735.9	47648.0	46944	48640	474.5	att_kernel
0.5	488094	12	40674.5	40623.5	40096	41536	377.0	preatt_kernel
0.2	161183	25	6447.3	6400.0	5759	7872	458.3	layernorm_forward_kernel
0.1	101568	24	4232.0	4352.0	3584	4864	542.3	residual_forward_kernel
0.1	96736	12	8061.3	8064.0	7936	8224	84.6	gelu_forward_kernel
0.1	93727	12	7810.6	7808.0	7680	7968	94.0	permute_kernel
0.1	58368	12	4864.0	4832.0	4800	4992	62.5	softmax_forward_kernel
0.0	50944	12	4245.3	4256.0	4224	4288	20.8	unpermute_kernel
0.0	6560	1	6560.0	6560.0	6560	6560	0.0	encoder_forward_kernel

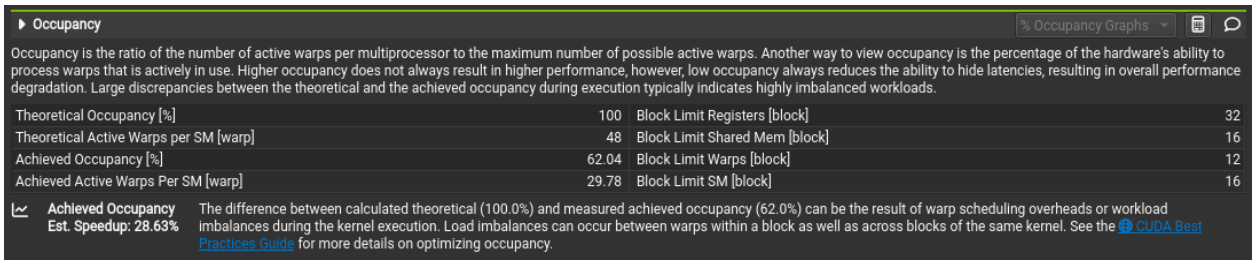
1d block dim: 512, 2d block dim: 8x8, 3d block dim: 8x8x8, unroll factor: compiler, coarse_factor: 8

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
95.6	23898810	49	487730.8	443007.0	140640	6445142	882190.5	matmul_forward_kernel
0.9	237440	25	9497.6	9344.0	8832	10784	471.8	layernorm_forward_kernel
0.8	201984	12	16832.0	16816.0	16736	16960	74.7	att_kernel
0.8	196607	12	16383.9	16384.0	16320	16448	36.1	preatt_kernel
0.5	127616	12	10634.7	10624.0	10336	10848	140.0	gelu_forward_kernel
0.4	108480	24	4520.0	4576.0	3968	4992	440.9	residual_forward_kernel
0.4	100928	12	8410.7	8448.0	8192	8576	127.1	permute_kernel
0.2	60832	12	5069.3	5056.0	5056	5088	16.5	unpermute_kernel
0.2	58592	12	4882.7	4864.0	4832	4992	51.9	softmax_forward_kernel
0.0	8544	1	8544.0	8544.0	8544	8544	0.0	encoder_forward_kernel

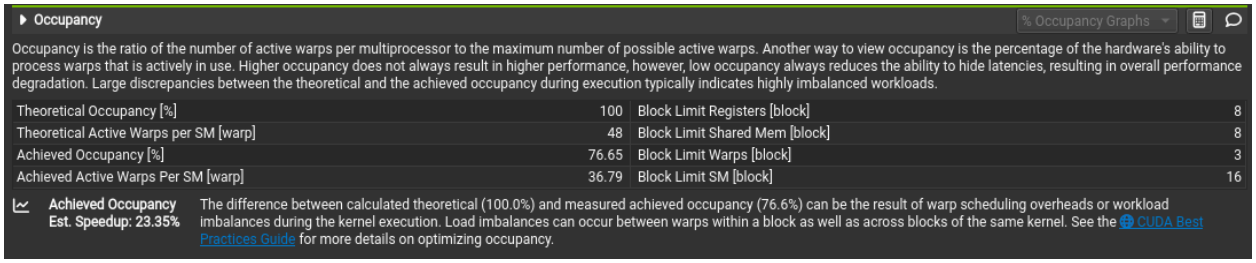
1d block dim: 1024, 2d block dim: 32x32, 3d block dim: 10x10x10, unroll factor: compiler, coarse_factor: 8

Looking through the different block configuration sweeps, we can see a general trend: the runtime of our joint register and shared memory tiling kernels decreased as the block sizes increased. This was especially true for our matmul_forward_kernel which saw a nearly 13.5x speedup when going from a 4x4 block to 32x32 block. We also saw speedups for the other kernels like att or preatt, suggesting that larger block sizes are better for this optimization. We believe this is because the larger block sizes allow for more shared memory reuse without going over the total shared memory capacity per SM. The smaller block sizes (like 4x4) also bottleneck the total number of threads in each SM due to the additional restriction on the maximum number of thread blocks per SM. When we looked at the other kernels, however, we saw a different story as some kernels like layernorm or gelu saw a gradual increase in runtime as block sizes

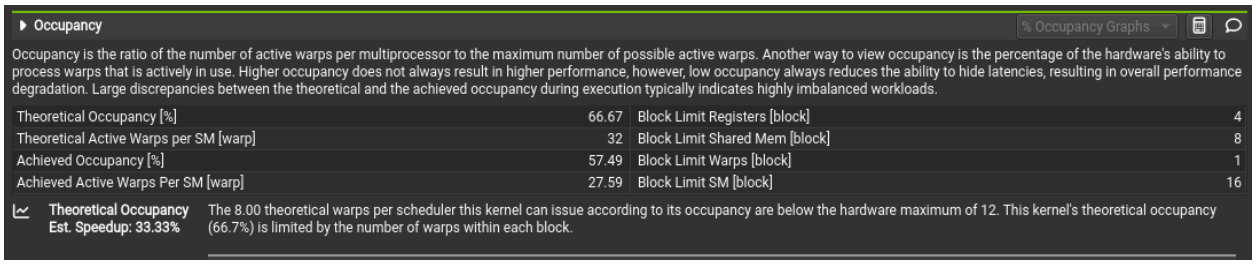
increased from 128 to 1024. This is likely due to the fact that larger block sizes can lead to various issues such as reduced occupancy or slower memory access patterns. There is also no tangible benefit to increasing block size for these kernels as there is no cooperation between the threads in a block. In order to better understand the reasons for the performance decreases, we looked at NSight compute for our gelu_forward_kernel:



gelu_forward_kernel with 1d block dim: 128

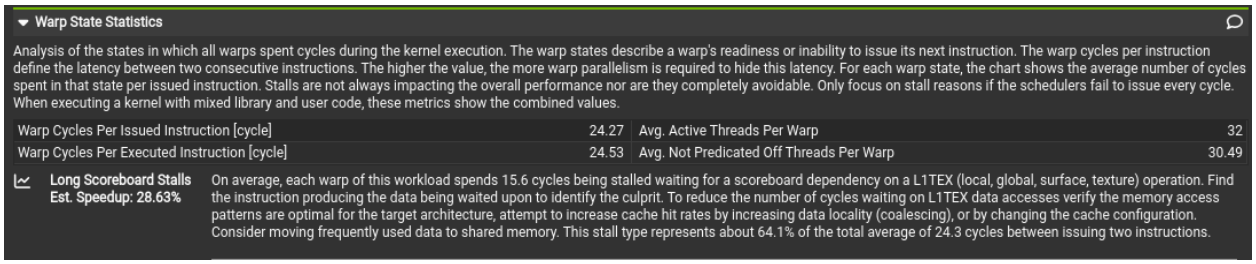


gelu_forward_kernel with 1d block dim: 512

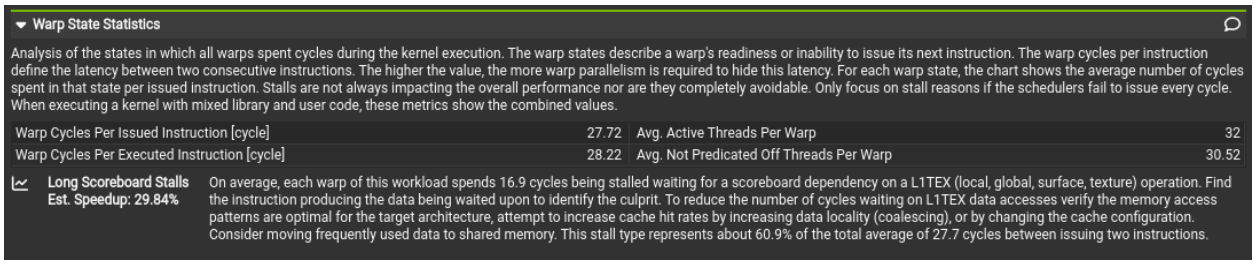


gelu_forward_kernel with 1d block dim: 1024

We can see that the achieved occupancy reduced from 62.04% to just 57.49% when going from 128 to 1024 threads per block, and that helps explain some of the decrease in performance. However, the achieved occupancy when going from a 128 to 512 block size actually increased, suggesting that there are other factors that led to the decrease in performance in the larger block size. We found similar results for our layernorm kernel as well. When we investigated other potential causes for the reduction in performance, we noticed the warp cycles per instruction differed between the two block sizes:



gelu_forward_kernel with 1d block dim: 128



gelu_forward_kernel with 1d block dim: 512

Although the version with 512 threads per block saw an increase in occupancy, it also saw an increase in the number of warp cycles it takes to execute each instruction from 24.53 to 28.22. This seems like a rather large increase which, along with other potential factors, offsets the improved occupancy, leading to a net decrease in performance. The note at the bottom clarifies that the version with 512 threads per block spent more cycles waiting for L1TEX memory operations to complete, suggesting that larger block sizes may lead to less ideal memory access patterns.

From our profiling results, we believe that larger block sizes are generally better for kernels that involve cooperation between threads in a block (e.g. via shared memory) so long as it does not exceed SM resource limits, but otherwise smaller block sizes are generally better. However, the block sizes should still be large enough so as to reach the max threads per SM while staying under the max blocks per SM because this helps keep occupancy high which is important for things like latency hiding.

Next, we profiled the three different pragma unroll factors. We weren't sure what to expect with this optimization as we had no prior comparison point such as what values the compiler automatically decides on, but we still expected to see a general improvement as the unroll factors increased because that allows a reduction in the number of instructions executed, but maybe with a certain cutoff at which point larger factors lead to worse performance.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
97.8	57536316	49	1174210.5	846976.0	176032	21480971	2987055.5	matmul_forward_kernel
0.6	326464	12	27205.3	27216.0	26400	27744	461.7	preatt_kernel
0.6	324288	12	27024.0	27232.0	26272	27584	483.1	att_kernel
0.2	145696	12	12141.3	12208.0	11744	12416	192.2	gelu_forward_kernel
0.2	138849	25	5554.0	5440.0	5152	6112	313.3	layernorm_forward_kernel
0.2	130081	12	10840.1	10864.0	10561	11072	148.8	permute_kernel
0.2	107296	24	4470.7	4480.0	4288	4736	137.2	residual_forward_kernel
0.1	64544	12	5378.7	5376.0	5280	5440	46.2	softmax_forward_kernel
0.1	60928	12	5077.3	5056.0	5024	5280	88.8	unpermute_kernel
0.0	6496	1	6496.0	6496.0	6496	6496	0.0	encoder_forward_kernel

1d block dim: 256, 2d block dim: 32, 3d block dim: 8, unroll factor: 1, coarse factor: 8

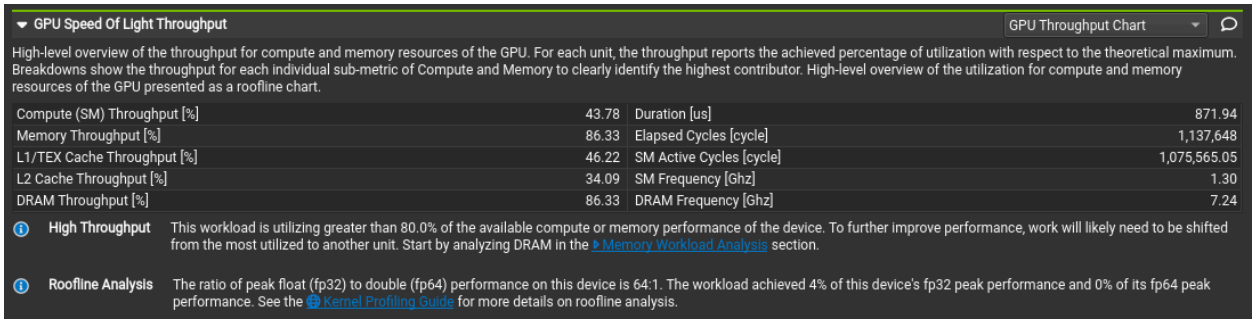
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
97.8	55602345	49	1134741.7	814976.0	167200	21002155	2922313.4	matmul_forward_kernel
0.5	306527	12	25543.9	25616.0	24864	26240	455.2	att_kernel
0.5	303423	12	25285.3	25328.0	24128	26208	595.6	preatt_kernel
0.3	142528	12	11877.3	11856.0	11648	12192	184.5	gelu_forward_kernel
0.2	135872	25	5434.9	5280.0	5056	5952	310.1	layernorm_forward_kernel
0.2	125888	12	10490.7	10512.0	10240	10816	161.9	permute_kernel
0.2	105184	24	4382.7	4384.0	3968	4704	178.7	residual_forward_kernel
0.1	60864	12	5072.0	5072.0	5056	5088	16.7	unpermute_kernel
0.1	60288	12	5024.0	5024.0	4960	5120	43.1	softmax_forward_kernel
0.0	6080	1	6080.0	6080.0	6080	6080	0.0	encoder_forward_kernel

1d block dim: 256, 2d block dim: 32, 3d block dim: 8, unroll factor: 16, coarse factor: 8

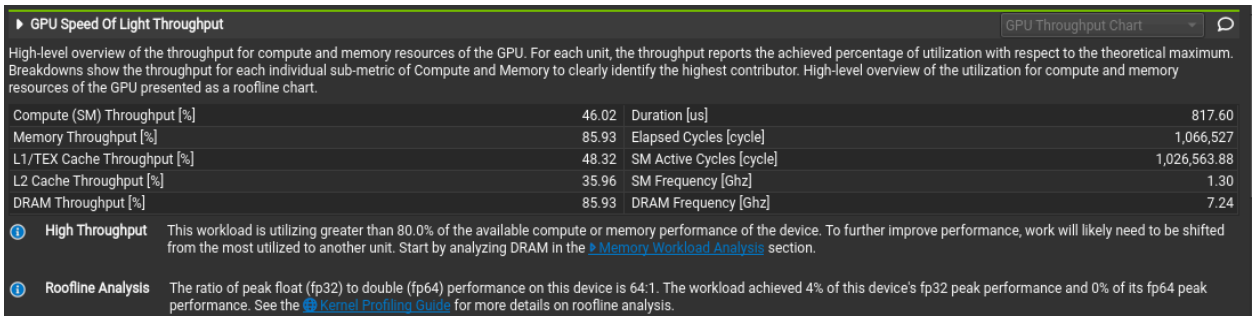
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
96.2	23853824	49	486812.7	442654.0	140480	6435526	880857.4	matmul_forward_kernel
0.8	201983	12	16831.9	16816.0	16640	17023	105.5	att_kernel
0.8	195646	12	16303.8	16288.0	16256	16383	41.9	preatt_kernel
0.5	131615	25	5264.6	5056.0	4832	5984	421.2	layernorm_forward_kernel
0.4	101055	12	8421.3	8415.5	8256	8672	109.9	permute_kernel
0.4	99104	12	8258.7	8208.0	8000	8608	165.1	gelu_forward_kernel
0.4	97503	24	4062.6	4096.0	3424	4672	477.9	residual_forward_kernel
0.2	60544	12	5045.3	5056.0	5024	5088	20.8	unpermute_kernel
0.2	55456	12	4621.3	4576.0	4544	5088	152.8	softmax_forward_kernel
0.0	6624	1	6624.0	6624.0	6624	6624	0.0	encoder_forward_kernel

1d block dim: 256, 2d block dim: 32, 3d block dim: 8, unroll factor: 64, coarse factor: 8

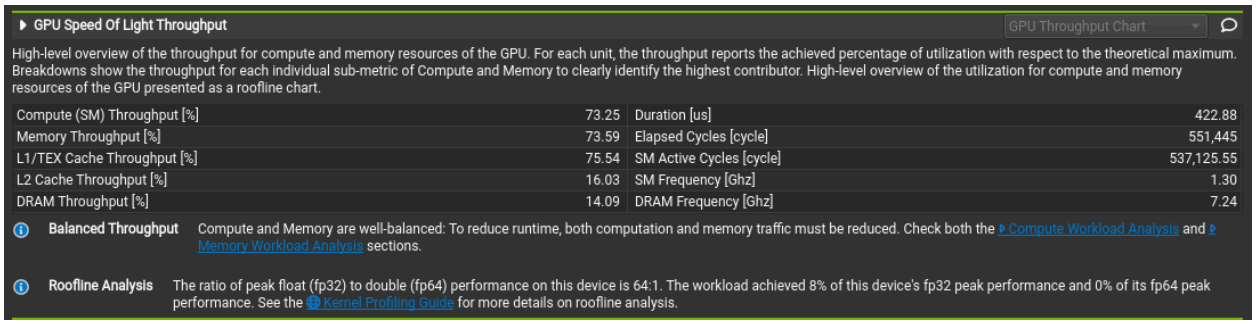
Looking through our profiling results, we can see that most kernels saw an improvement with larger unroll factors. Interestingly, the jump from an unroll factor of 1 to a factor of 16 (a 16x increase) resulted in a relatively small performance increase in our kernels, but the jump from a factor of 16 to a factor of 64 (a 4x increase) resulted in a drastic improvement. The only kernel that didn't see a consistent performance improvement with larger unroll sizes was the encoder kernel, but this kernel didn't even have an pragma unroll in it meaning the variations in runtime was likely due to other factors.



matmul_forward_kernel with unroll factor: 1

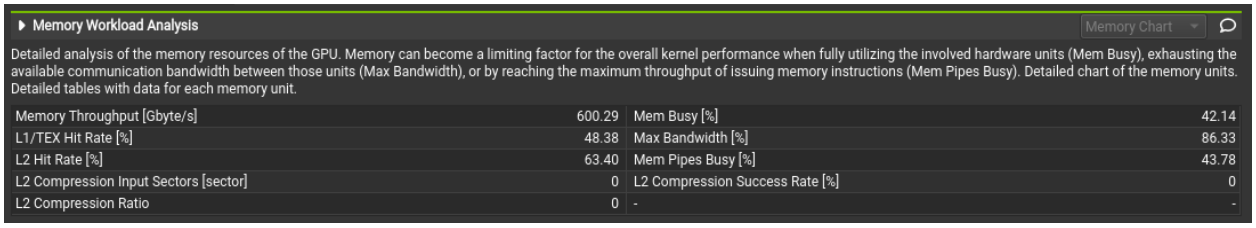


matmul_forward_kernel with unroll factor: 16

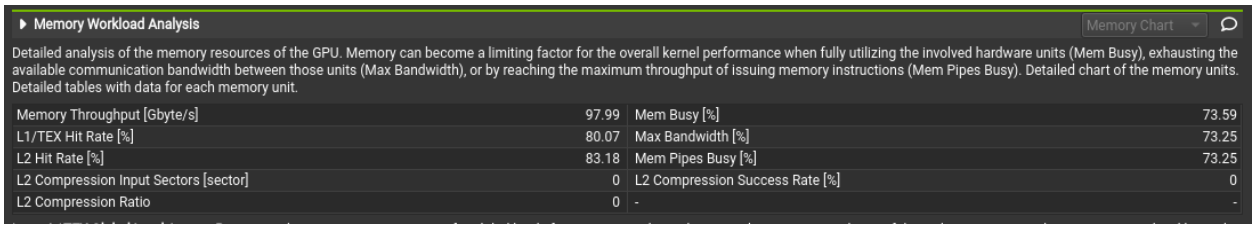


matmul_forward_kernel with unroll factor: 64

Looking at the compute throughputs for the matmul_forward_kernel, we can see that it's significantly higher with an unroll factor of 64 compared to with an unroll factor of 1, going from 43.78% to 73.25%. This is also accompanied by a lower overall memory throughput. Most notably, the DRAM throughput decreased the most, going from 86.33% to just 14.09%, while the L1/TEX cache throughput increased. This suggests that higher unrolling factors allow for a much higher rate of cache hits which is supported by the below statistics:



matmul_forward_kernel with unroll factor: 1



matmul_forward_kernel with unroll factor: 64

We can see the L1/TEX hit rate nearly doubled from 48.38% to 80.07%, and a higher cache hit rate is often in part due to better coalesced memory accesses. We can also see the mem pipes busy percentage

went from 43.78% to 73.25% which seems to suggest that the larger unroll factors allow the GPU to request more memory transactions in a single loop.

Although my first instincts were that loop unrolling would allow for more compute throughput because it reduces the number of instructions needed to increment the loop counter, it seems that most of the performance benefits actually come from memory-related factors. Things like better cache hit rates and more memory pipe utilization saw the most improvement with larger unroll factors. It's also worth noting that this version of the matmul_forward_kernel with an unroll factor of 64 is the best performing version we achieved throughout this configuration sweep. It's possible that we could achieve even better performance with even larger unroll factors.

Finally, we swept through different coarsening factors. The only kernels this applied to are the joint register and shared memory tiling kernels (matmul and attention). We had already tried out different values when optimizing the matmul kernel during the previous milestone, and so we felt fairly confident the values we chose were in a good range for that specific kernel. However, we didn't test as thoroughly with the attention kernels, so we were hoping to see some performance improvements with one of the other coarsening factors in this sweep. The results from that are shown below:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
96.4	24420537	49	498378.3	451998.0	142527	6537627	895134.0	matmul_forward_kernel
0.7	184288	12	15357.3	15344.0	15296	15520	67.5	preatt_kernel
0.7	174333	12	14527.8	14527.5	14463	14720	72.2	att_kernel
0.5	128960	25	5158.4	5024.0	4672	5888	447.2	layernorm_forward_kernel
0.4	100829	12	8402.4	8384.0	8287	8512	75.2	permute_kernel
0.4	97918	24	4079.9	4112.0	3328	4736	589.8	residual_forward_kernel
0.4	97662	12	8138.5	8144.0	7935	8351	135.3	gelu_forward_kernel
0.2	60544	12	5045.3	5056.0	5024	5088	20.8	unpermute_kernel
0.2	58560	12	4880.0	4864.0	4800	5024	55.4	softmax_forward_kernel
0.0	6816	1	6816.0	6816.0	6816	6816	0.0	encoder_forward_kernel

1d block dim: 256, 2d block dim: 32, 3d block dim: 8, unroll factor: compiler, coarse factor: 4

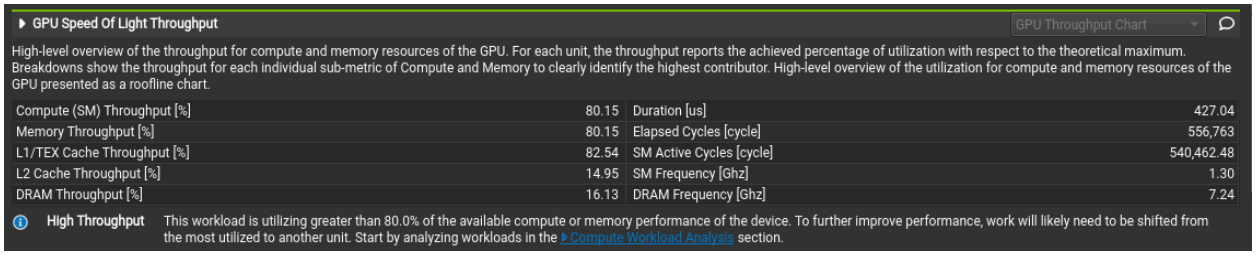
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
96.2	23878117	49	487308.5	442749.0	140735	6436480	880961.1	matmul_forward_kernel
0.8	202143	12	16845.3	16832.0	16768	16992	77.6	att_kernel
0.8	196831	12	16402.6	16400.0	16320	16512	51.9	preatt_kernel
0.5	130272	25	5210.9	5024.0	4672	6016	429.5	layernorm_forward_kernel
0.4	100959	12	8413.3	8447.5	8256	8544	92.0	permute_kernel
0.4	99616	12	8301.3	8288.0	8000	8512	147.9	gelu_forward_kernel
0.4	97152	24	4048.0	4048.0	3488	4832	488.9	residual_forward_kernel
0.2	60576	12	5048.0	5056.0	5024	5088	19.9	unpermute_kernel
0.2	58592	12	4882.7	4864.0	4832	5024	68.8	softmax_forward_kernel
0.0	6368	1	6368.0	6368.0	6368	6368	0.0	encoder_forward_kernel

1d block dim: 256, 2d block dim: 32, 3d block dim: 8, unroll factor: compiler, coarse factor: 8

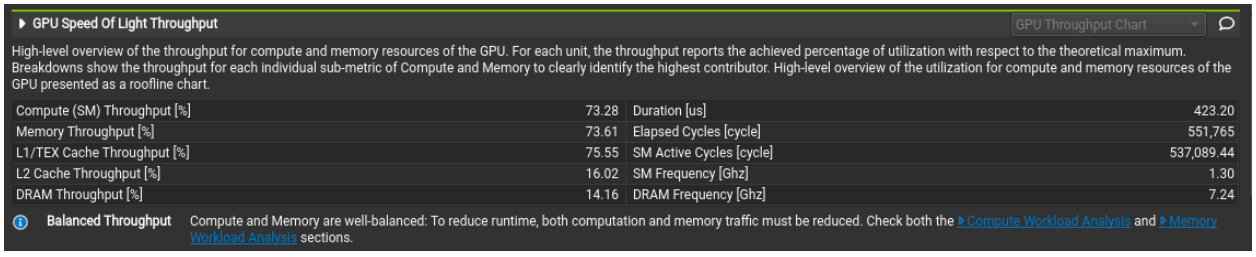
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
97.2	36979471	49	754683.1	581215.0	187839	13919189	1930668.4	matmul_forward_kernel
0.6	241375	12	20114.6	20112.0	19872	20448	174.6	att_kernel
0.6	228096	12	19008.0	18992.0	18912	19232	88.4	preatt_kernel
0.3	132928	25	5317.1	5184.0	4736	5984	391.6	layernorm_forward_kernel
0.3	132448	12	11037.3	11008.0	10784	11232	127.6	gelu_forward_kernel
0.3	117792	12	9816.0	9808.0	9632	10080	128.1	permute_kernel
0.2	93664	24	3902.7	3904.0	3776	4096	75.8	residual_forward_kernel
0.2	60576	12	5048.0	5056.0	5024	5088	19.9	unpermute_kernel
0.2	58624	12	4885.3	4864.0	4832	4992	45.9	softmax_forward_kernel
0.0	6592	1	6592.0	6592.0	6592	6592	0.0	encoder_forward_kernel

1d block dim: 256, 2d block dim: 32, 3d block dim: 8, unroll factor: compiler, coarse factor: 16

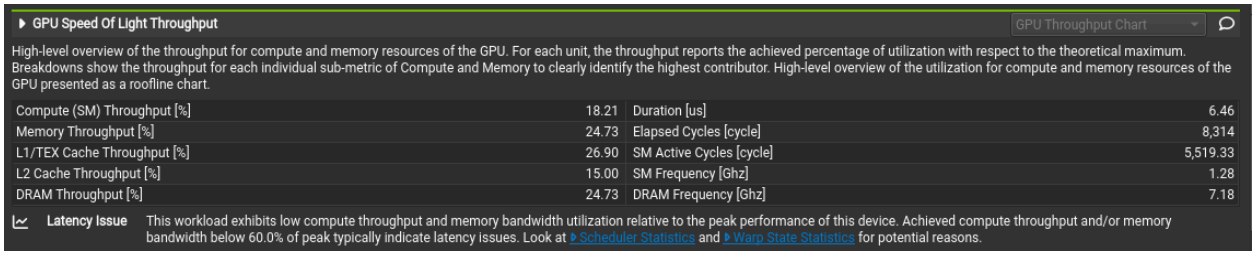
We can see our original coarsening factor of 8 was indeed the best performing version for our matmul kernel which was more or less expected. But we also saw that the attention kernels (att and preatt) actually performed better with smaller coarsening factors. We first decided to try to figure out why the matmul responded differently to different factors:



matmul_forward_kernel with coarse factor: 4



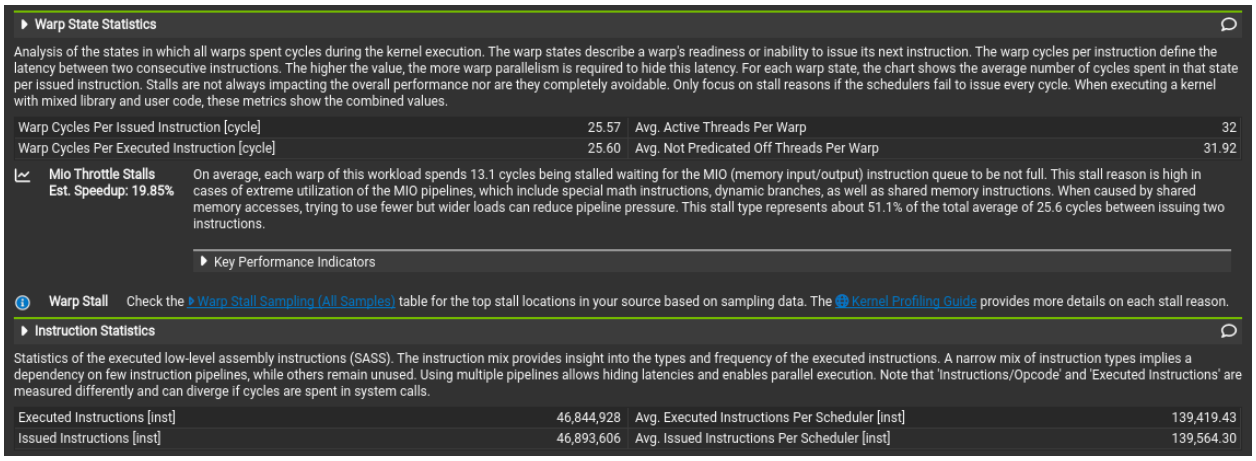
matmul_forward_kernel with coarse factor: 8



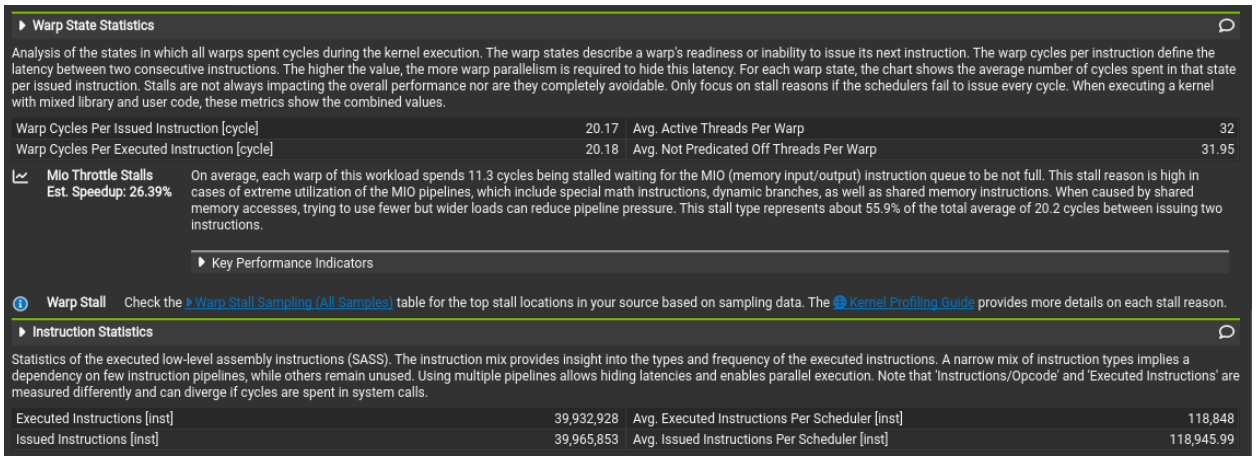
matmul_forward_kernel with coarse factor: 16

We can see that the matmul kernel had lower usage of a variety of resources when we increased the coarse factor from 4 to 8. Things like compute throughput, memory throughput, L1/TEX cache throughput, and DRAM throughput all decreased. These seem like signs of lower occupancy which was verified within

NSight compute, going from an achieved occupancy of 31.00% to just 15.69% (screenshot not shown). The reduction in throughput between a coarse factor of 4 and 8 is actually lower than expected, but this could be explained by the role that matmul plays in the overall gpt2 architecture as its input sizes are large and require a large number of thread blocks which can exceed the amount of hardware resources. Instead of letting the hardware serialize the execution of blocks, we sort of apply serialization manually on a thread-level via coarsening. When we increased the coarse factor even further from 8 to 16, we saw a massive reduction in throughput, meaning this may be the point at which further coarsening will not make full use of the hardware. But this doesn't completely explain why a coarsening factor of 8 improves performance compared to a factor of 4. Why is that the case?



matmul_forward_kernel with coarse factor: 4



matmul_forward_kernel with coarse factor: 8

It turns out that increasing the coarsening factor leads to fewer instructions being executed. This may be because each thread has a bit of "boilerplate" code that doesn't scale with the coarse factor such as the initialization of loop counters or some conditional statements, and applying thread coarsening allows us to remove some of those instructions from the grid. Additionally, we noticed that the number of warp cycles per instruction decreased as well which could be explained by an increase in the number of matrix elements stored in registers when using larger coarsening factors, allowing for better reuse.

As for why the preatt kernel saw a performance reduction when going from a coarse factor of 4 to 8, we believe the inputs for this kernel may be small enough such that the lowered compute/memory throughput would already cause a net detriment even at these smaller factors. In the end, this sweep provided a valuable insight: a coarse factor of 4 seems to work best for the attention kernels, and a factor of 8 works best for the matmul kernel.

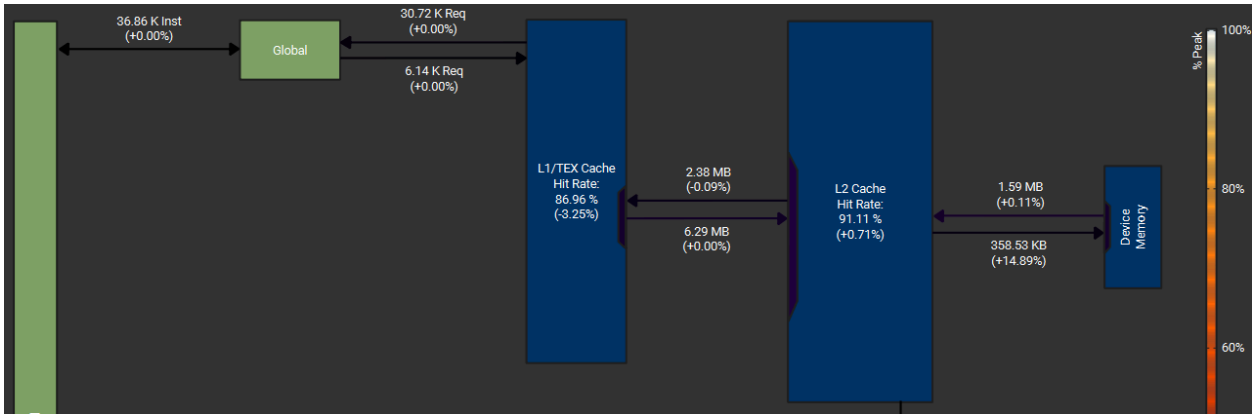
It's important to keep in mind that many of these profiling results are specific to the inputs and parameters that the test_gpt2 file uses. Different inputs will lead to different results, and it's crucial to test and test and test in order to achieve the best results.

Restrict

The restrict keyword is a sign to the compiler that any pointer with that keyword will never be accessing the same address space as another restrict pointer. This enables aggressive compiler optimizations including instruction reordering, redundant load elimination, and improved register allocation. When the compiler knows two pointers don't overlap, it can safely cache values in registers across multiple uses, eliminate redundant memory accesses, and reorder operations for better parallelism without worrying about read-after-write or write-after-read hazards.

Observing our kernel implementations, we identified that most of our kernels have clearly separated input and output buffers with no aliasing. For example, in matmul, the input, weight, bias, and out pointers all reference distinct memory regions—the input and weight matrices are read-only, bias is read-only, and output is write-only with no overlap. Similarly, layernorm reads from input and writes to separate out, mean, and rstd buffers. The GELU, encoder, and residual kernels all follow this pattern of non-overlapping memory regions. The key challenge was identifying which kernels could safely use __restrict__ without introducing correctness issues. We had to verify that our kernel launch patterns never perform operations where input equals output like layernorm_forward, as this could violate the aliasing guarantee and produce undefined behavior.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	
79.0	86207075	49	1759328.1	1609113.0	428766	25980023	3569130.9	matmul_forward_kernel(f
7.7	8390783	12	699231.9	699389.0	697981	700701	798.4	vaccum_kernel(float *,
7.7	8372414	12	697701.2	697293.5	696190	701245	1741.0	preatt_kernel(float *,
3.9	4237742	25	169509.7	170560.0	164000	180479	3995.1	layernorm_forward_kerne
1.0	1039901	12	86658.4	86688.0	86080	87072	320.0	permute_kernel(float *,
0.3	368926	12	30743.8	30719.5	30496	31168	192.6	softmax_forward_kernel(
0.3	347677	12	28973.1	28975.5	28896	29056	44.1	unpermute_kernel(float
0.1	95359	24	3973.3	3936.0	3424	4544	405.4	residual_forward_kernel
0.1	93792	12	7816.0	7776.0	7712	7968	99.5	gelu_forward_kernel(flo
0.0	6528	1	6528.0	6528.0	6528	6528	0.0	encoder_forward_kernel(



The majority of differences between the baseline and restrict versions can be explained by the variation between tests. Restrict likely fails to do anything meaningful here because the bottle necks are actually memory access times and not order of instructions. Restrict could prove beneficial in kernels that are compute bound rather than memory bound, but in these memory bound kernels we tested on there isn't much that store or load reordering would do.

Performance results showed minimal overall improvement with essentially no measurable speedup to total execution time. The per-kernel breakdown revealed that matmul remained at about the same time for baseline and restrict versions. The layernorm kernel showed the only notable improvement at around a 17% speedup. This makes sense given layernorm's multi-pass structure with sequential mean and variance calculations and normalization passes where restrict allows the compiler to cache input values in registers across passes and eliminate redundant loads. Since layernorm represents less than 5% of the total execution time this improvement translates to a less than 1% speedup overall. The total execution time remained pretty much the same, indicating that restrict provided minimal benefit to our implementation despite the theoretical advantages and clear non-aliasing properties of our code.

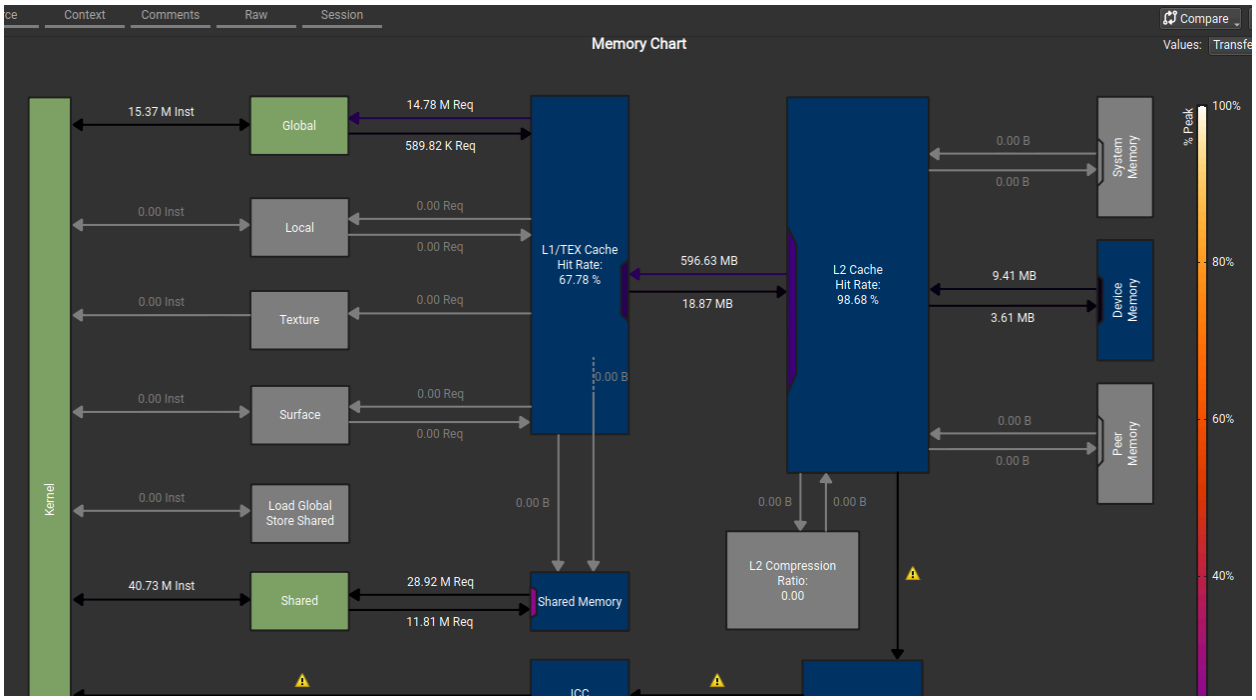
Kernel Fusion

Kernel fusion serves to combine multiple calls of sequential kernel launches into one launch, and in turn mainly focusing on removing the overhead of outputting from the first kernel and inputting those same values immediately into the next kernel. We utilize a lot of shared memory in a standard kernel fusion implementation to replace the 'input' to the second kernel, and that is what we used as well.

Observing the kernel launches from gpt2.cuh, we see layernorm outputting into memory and being loaded straight back to matmul immediately following. This is a prime opportunity for a kernel fusion, and we worked to combine these two calls into one layernorm_matmul call. The most limiting factor regarding redesign of the algorithms is the fact that our launch dimensions must be the same between both parts of the kernel. This required us to take a fairly inefficient route with the actual computation, but we are hoping this is compensated by our removal of the intermediate memory operations.

```
        // now do the forward pass
        // layernorm_forward(l_ln1, l_ln1_mean, l_ln1_rstd, residual, l_ln1w, l_ln1b, B, T, C);
        // matmul_forward(scratch, l_ln1, l_qkvw, l_qkvb, B, T, C, 3*C);
        layernorm_matmul_forward(scratch, l_ln1_mean, l_ln1_rstd, residual, l_ln1w, l_ln1b, l_qkvw, l_qkvb, B, T, C, 3*C
        attention_forward(l_atty, l_qkvr, l_att, scratch, B, T, C, NH);
        matmul_forward(l_attproj, l_atty, l_attprojw, l_attprojb, B, T, C, C);
        residual_forward(l_residual2, residual, l_attproj, B*T*C);
        // layernorm_forward(l_ln2, l_ln2_mean, l_ln2_rstd, l_residual2, l_ln2w, l_ln2b, B, T, C);
        // matmul_forward(l_fch, l_ln2, l_fcw, l_fcb, B, T, C, 4*C);
        layernorm_matmul_forward(l_fch, l_ln2_mean, l_ln2_rstd, l_residual2, l_ln2w, l_ln2b, l_fcw, l_fcb, B, T, C, 4*C)
        gelu_forward(l_fch_gelu, l_fch, B*T*4*C);
        matmul_forward(l_fcproj, l_fch_gelu, l_fcprojw, l_fcprojb, B, T, 4*C, C);
        residual_forward(l_residual3, l_residual2, l_fcproj, B*T*C);
    }

    residual = acts.residual3 + (L-1) * B * T * C; // last residual is in residual3
```

FOR TESTING: Above is the fused layernorm_matmul_forward call replacing the separate layernorm and matmul launches that are commented out right above. This is how you would run the custom kernel we designed for this optimization. Also #include kernels_fusion/layernorm_matmul.cuh in the gpt script. This optimization was demo'd to be working correctly on Monday, and the commit that we used at that time was hash "388a145e3860a26f6d8833c1d481f0cf99e038a7" on "Kevin_branch" which you can see was committed the weekend before the demo.



The above diagrams from our profiling results highlight the reduced global memory loads, and also the bottleneck being shifted away from memory-limits towards a more even split of workload along computation and memory.

Analyzing these results, we concluded that the kernel fusion did sufficiently circumvent the loading and storing to global memory between the immediate kernels we replaced, and instead offloaded the output of layering to the shared memory that we can see was heavily utilized by the following matmul. We consider this a success, however, our overall runtime was not great, which we attribute to a worse algorithm for reduction in our layer norm, and more forced serial computation of matmul outputs. The change in grid launch dimensions was hard to work around algorithm wise while maintaining similarity to milestone 1, but still the main focus of minimizing the redundant memory offloading to global memory was achieved. With more focus on runtime and efficient algorithms, we think kernel fusion is a solid optimization that can be extended to minimize runtime in many use cases.

Flash Attention

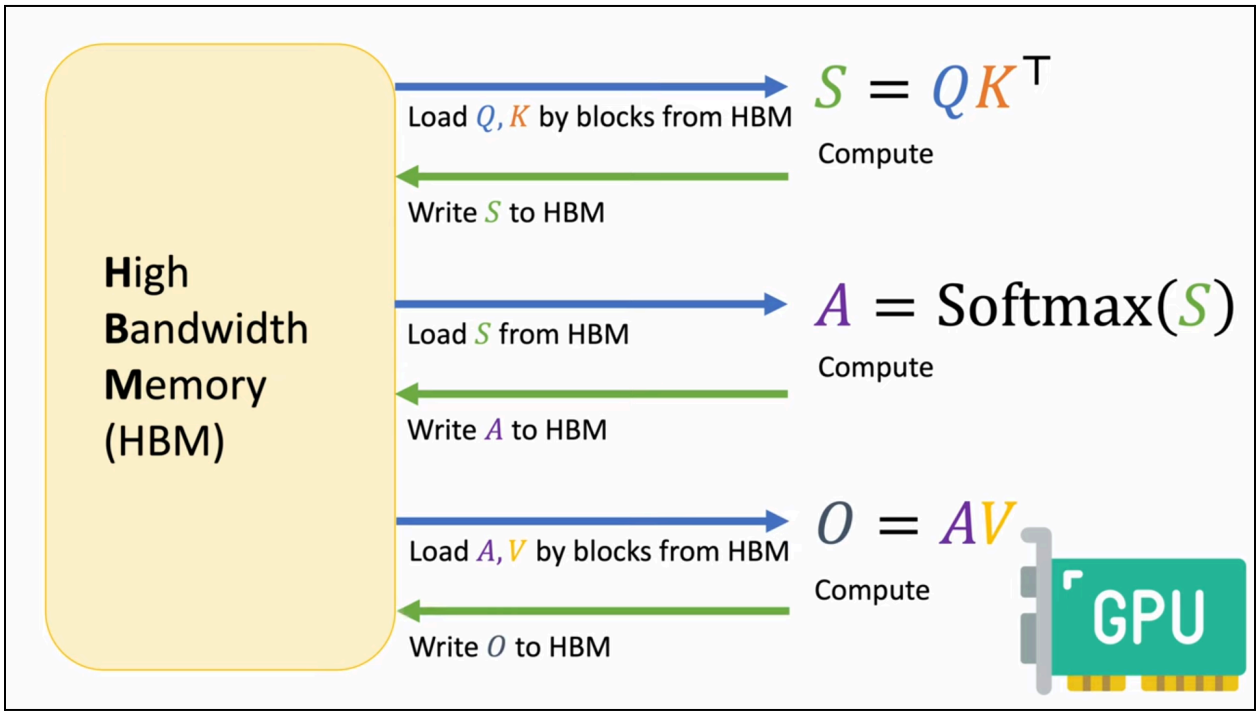
Flash Attention is an IO-aware exact attention algorithm that accelerates Transformer training and inference by optimizing GPU memory accesses. It addresses the memory bandwidth bottleneck in standard attention mechanisms by reducing the number of read/write operations between the GPU's slow High Bandwidth Memory (HBM) and fast on-chip SRAM.

Standard attention consists of three kernels that compute the following expressions. This algorithm forces large data transfer between host and device, and it also disables the use of tiling because computing softmax requires the entire row of tokens.

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Standard Attention Algorithm

We have a batch size of 4, token size of 64, and token length of 768, transferring such large data back and forth from host to device will inherently slow the program down.



Standard Attention Memory Transfer Diagram (Huang)

To solve this problem, flash attention proposes the utilization of Online Softmax.

$$
\begin{aligned}
&1: \ m_0 \leftarrow -\infty \\
&2: \ d_0 \leftarrow 0 \\
&3: \ \textbf{for } j \leftarrow 1, V \textbf{ do} \\
&4: \quad m_j \leftarrow \max\left(m_{j-1}, x_j\right) \\
&5: \quad d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j} \\
&6: \ \textbf{end for} \\
&7: \ \textbf{for } i \leftarrow 1, V \textbf{ do} \\
&8: \quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V} \\
&9: \ \textbf{end for}
\end{aligned}
$$

Safe softmax with online normalizer calculation

Online Softmax algorithm keeps a running max value m and normalization term d as it iterates over the input, continuously rescaling earlier values when a new max appears. This eliminates the need to compute

the global maximum upfront, allowing Softmax to be computed on an iterative basis.

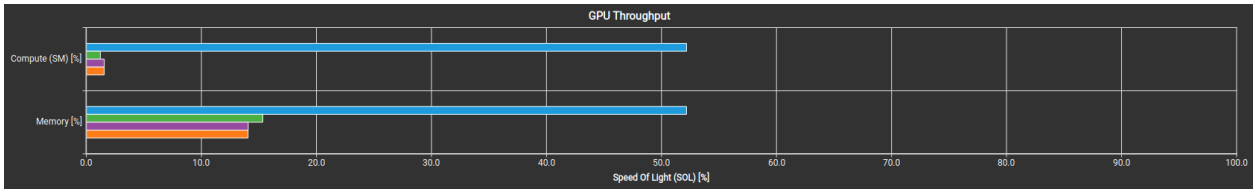
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

Flash Attention Algorithm

Online Softmax algorithm allows the computation of softmax without having to know the maximum value of vector S. This removes the dependency and allows us to perform kernel fusion on the attention kernels. Using Online Softmax, Flash Attention fuses all three attention steps (QK matmul, Softmax, and the PV matmul) into one kernel. With removed dependency and data movement between the kernels, this allows us to do tiling on attention.

Let's take a look at the improved performance.



Speed of Light Comparison

With the significantly reduced memory overhead achieved by kernel fusion and tiling, we now observe that our flash attention kernel has balanced compute and memory latency. Previously, all three attention kernels were memory-bound and did not utilize shared memory. Another thing that we could look at to compare the memory bottleneck is the warp states.

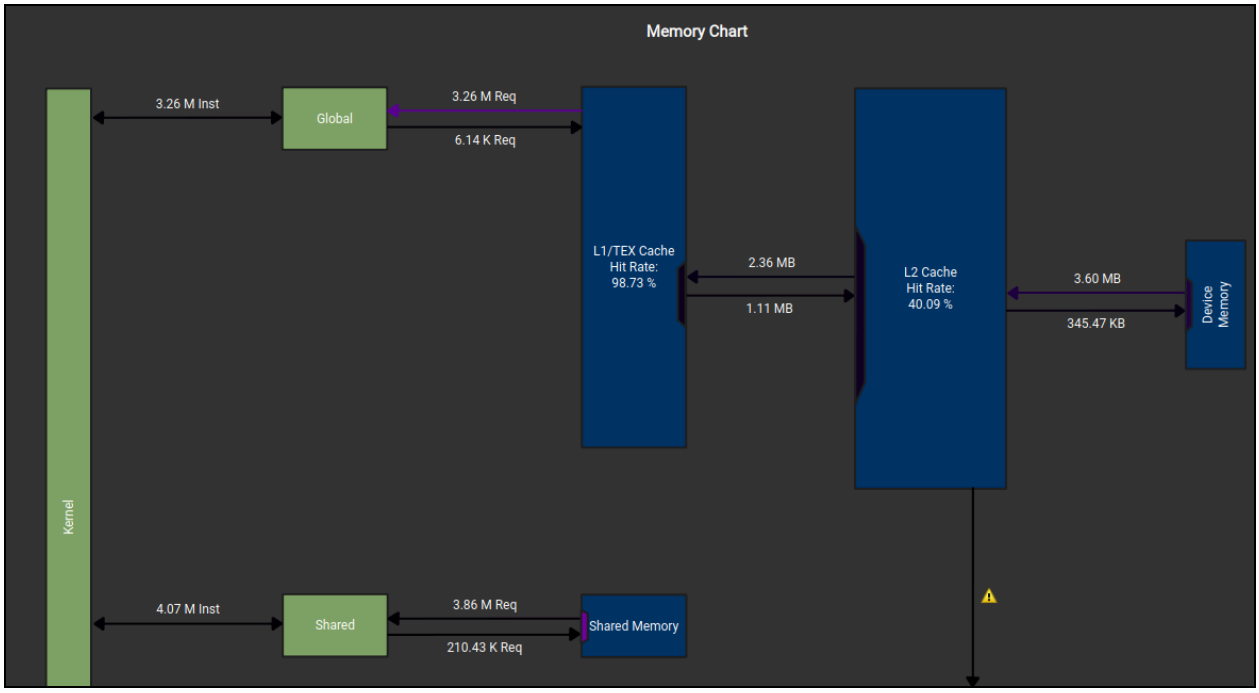
Stall Long Scoreboard

Result	Value
Flash Attention	0.46
Softmax kernel	1,18.61 (-97.52%)
Preatt kernel	2,17.81 (-97.41%)
Vaccum Kernel	3,30.91 (-98.51%)

Stall LG Throttle

Result	Value
Flash Attention	0.13
Softmax kernel	1,6.22 (-97.91%)
Preatt kernel	2,38.19 (-99.66%)
Vaccum Kernel	3,2.83 (-95.40%)

The warp state of the Stall Long Scoreboard suggests that the kernel is requesting data from Global Memory (DRAM) or L2, and the arithmetic units are waiting for that data to arrive. And we could see that flash attention with kernel fusion and tiling was able to reduce this stall significantly. The Stall LG Throttle also suggests a similar memory bottleneck but in terms of load/store unit being full and memory instructions being unable to be issued.



Flash Attention Memory Chart

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
91.7	86,235,183	49	1759901.7	1610589.0	428800	25968532	3567453.1	matmul_forward_kernel
5.5	5,132,983	25	205319.3	205408.0	191712	211039	3610.3	layernorm_forward_kernel
2.5	2,306,618	12	192218.2	192207.5	192000	192512	145.5	flashatt_kernel
0.1	97,824	12	8152.0	8144.0	8032	8416	96.6	permute_kernel
0.1	96,736	24	4030.7	4048.0	3616	4448	254.3	residual_forward_kernel
0.1	92,800	12	7733.3	7776.0	7520	7936	113.6	gelu_forward_kernel
0.1	60,352	12	5029.3	5024.0	4992	5088	26.7	unpermute_kernel
0.0	6,464	1	6464.0	6464.0	6464	6464	0.0	encoder_forward_kernel

Our flash attention kernel had a runtime of 2,306,618ns compared to standard attention kernels which totaled up to 17,127,923ns, resulting in 7.4x speedup. This is a massive improvement, and a testament to the algorithm developed and so widely utilized, and how significant each of the individual components of optimization mentioned before are when combined.

Local/Windowed Attention

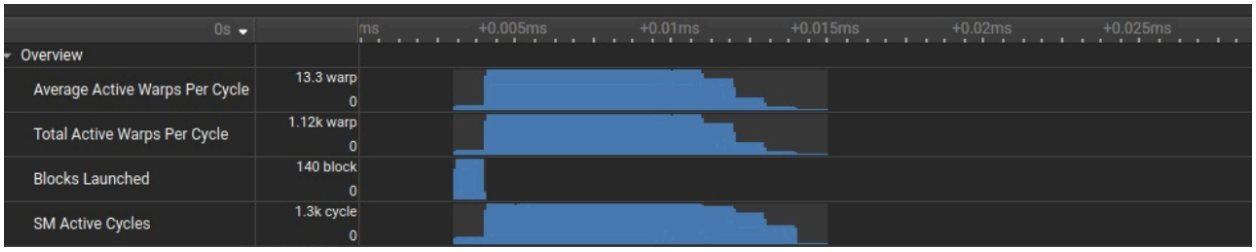
Self-attention algorithms typically calculate attention scores for the current vector relative to all tokens that had been generated preceding the current one. This is obviously a huge toll on memory load and compute power, and we try to subsidize the process with this optimization for windowed attention. Given a hyper-parameter window size, we only apply the attention algorithm on the tokens within a window before the current vector, that way we do not have to read an entire matrix to calculate our vector's attention.

The motivation behind this is mainly compute complexity dropping from previously O(T^2) to now O(T * Window), which keeps it more constant when you get deep into the token space.

An obvious drawback of this would be a loss in accuracy of our output, as we are not accounting for every prior token, just the ones within the window. The way GPT models operate, however, does not mean our output is completely wrong, just that it has a more limited view of data to generate with. We will look into errors of our results as well as the affect on performance this optimization comes with.

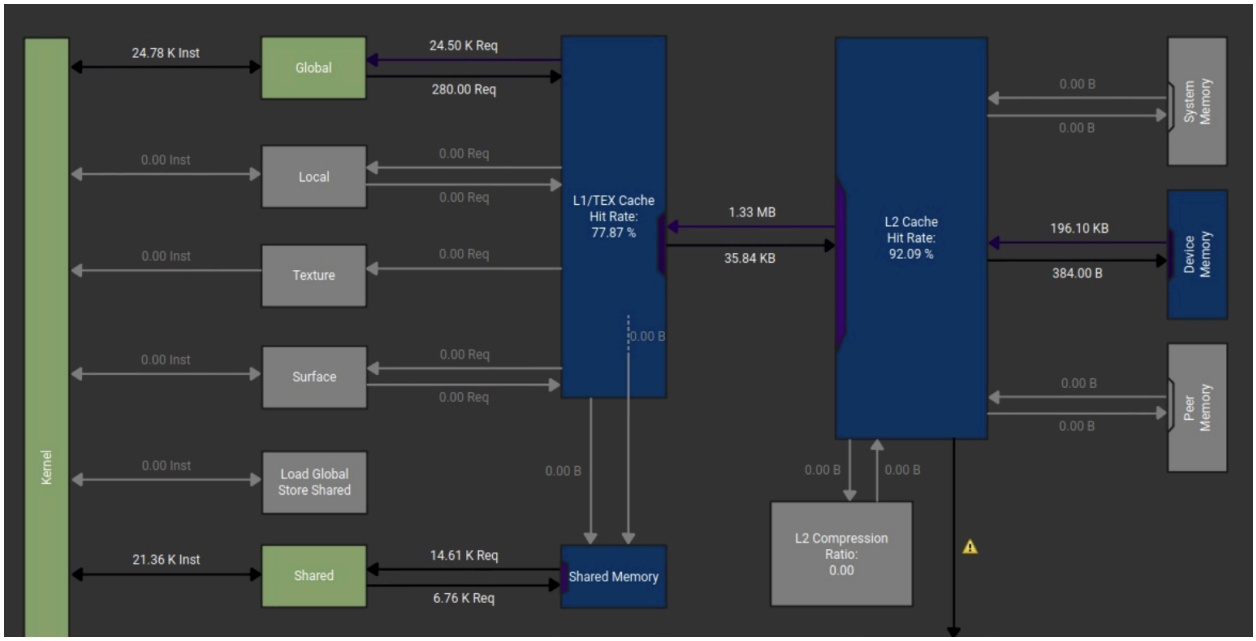
```
GPT2 - Reallocating activations for B=1 T=400
GPT2 - Allocated 420 MiB for activations
Final sequence length: 400 tokens
Local attention context: last 128 tokens (32.0% of sequence)

Logit statistics at final position:
  - Min logit:  -131.8952
  - Max logit:  -64.5230
  - Mean logit: -92.7385
  - Range:      67.3722

Top-5 predicted tokens at final position:
  1. Token ID 198 (logit: -64.5230)
  2. Token ID 50256 (logit: -70.1190)
  3. Token ID 628 (logit: -71.9090)
  4. Token ID 1222 (logit: -72.2746)
  5. Token ID 11 (logit: -72.7165)
```

Above is an output that was generated by ChatGPT using our GPT test models with a little more insight into the actual accuracy relative to the full context than we could get with our own tests in isolation. Varying the inputs and using our set 384 Token generation window, by the time we hit our last generation, you can see the variation in predicted tokens and the range of possible outputs we are predicting. We see a pretty high deviation between the generated tokens, and this is likely due to our model trying to guess the context that precedes the window that it has absolute insight into. The loss in accuracy is hard to quantify, but in theory 128 tokens is a solid amount of context to continue generation, just with the possibility of some prompts completely derailing mid-response.



Profiling the actual efficiency of our window limitation was not the most effective for us, as we did not have a baseline model to compare to, but we can still analyze with the intended consequences of our optimization in mind. Above we see the active warps per cycle, and we see they are pretty uniform

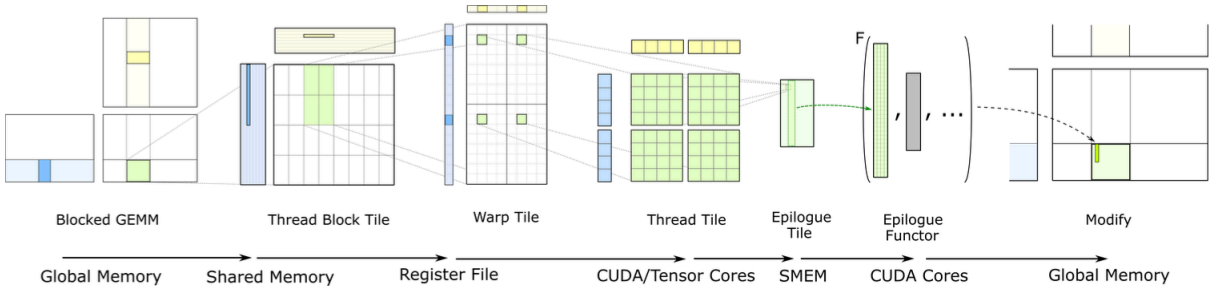
without dropping out for most the run. This uniformity can be attributed to our window maintaining a set ’limit’ of sorts on how long our loops will run for with our window sizes.



We also saw a strong L2 Cache hit rate, as the eviction process of previously loaded weights in the cache has been mitigated by the limited size of our window. Now when we are reading weights for attention calculations at a given index, it is more likely that we would have seen the input in a different thread just previously compared to if our context was unlimited and other threads already would have dropped out by the time later token attention was calculated.

NVIDIA CUTLASS

NVIDIA cuBLAS is a highly optimized library that abstracts away many of the finer details necessary for high-performance matrix multiplication, but it is that very abstraction that makes it difficult to fine-tune for specific use cases. CUTLASS on the other hand provides a more manual approach that allows you to adjust various parameters such as operator class, architecture, threadblock size, and epilogue functions, giving way for potential performance improvements over cuBLAS.



The Hierarchical GEMM Computation Embodied by CUTLASS

For this optimization, one of the things we focused on was the use of epilogue functions to add the bias vector as that was something that wasn't possible with the cublas<t>gemmStridedBatched() function used in the previous milestone. Doing so allows us to perform our bias addition without the overhead of a separate kernel launch, essentially allowing us to perform kernel fusion. The reason this is possible is because CUTLASS allows us to perform the matrix multiplication operation defined by $D = alpha * (A@B) + beta * C$ where we can insert the bias vector for input C. cuBLAS did not provide this ability to use separate matrices for inputs C and D which was why we had to handle the bias separately.

By default, the CUTLASS functions we used performed matrix multiplication using CUDA cores. We wanted to instead utilize tensor cores and test out different tiling sizes, but we could not get this to work. We managed to get a kernel using the tensor core operator class to compile and pass the output_verification_rand tests, but it always failed with a misaligned memory error at test_gpt2.cu:75. We tried debugging this until the deadline was near but ultimately could not manage to get it working. However, the use of CUDA cores allows us to see what kinds of optimizations we could make to our other non-tensor core optimizations, making it still valuable.

In order to test this optimization, the following steps should be taken:

1. Clone the project repository and switch to the `lukeyou` branch.
2. Copy the Makefile under kernels_req_lukeyou/ into the top-level directory of the project.
3. Copy attention.cuh and matmul.cuh under kernels_req_lukeyou/ into the kernels/ folder.
4. Make the project using `make all`.
5. Edit the verify.slurm file to run output_verification_rand and test_gpt2.
6. Run `sbatch verify.slurm` and check test results once done. All tests should pass.

We first decided to focus on the effect of using an epilogue function to combine the bias addition kernel. Our profiling results are shown below:

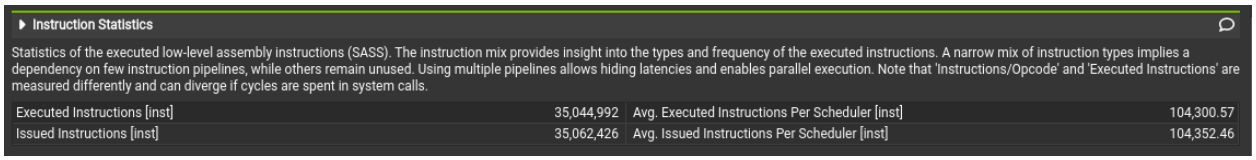
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
61.6	9796623	61	160600.4	88063.0	11456	1620893	218623.8	void cutlass::Kernel (matmul)
32.2	5127064	25	205082.6	205216.0	191552	210303	3643.2	layernorm_forward_kernel
2.2	350208	48	7296.0	6720.0	4704	11808	2404.7	matmul_forward_kernel
1.3	205760	12	17146.7	17120.0	16992	17408	120.4	softmax_forward_kernel
0.9	142368	12	11864.0	11824.0	11744	12288	150.2	void cutlass::Kernel (attention)
0.5	82176	24	3424.0	3408.0	3104	3840	225.9	residual_forward_kernel
0.4	70623	12	5885.3	5856.0	5791	6304	136.8	permute_kernel
0.4	67200	12	5600.0	5472.0	5344	6272	326.9	gelu_forward_kernel
0.4	60672	12	5056.0	5056.0	5024	5088	13.6	unpermute_kernel
0.0	6464	1	6464.0	6464.0	6464	6464	0.0	encoder_forward_kernel

NVIDIA CUTLASS (separate bias) Execution Times

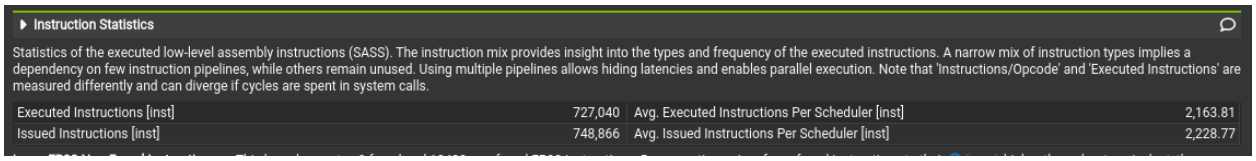
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
62.9	9844619	61	161387.2	88544.0	11648	1631485	219771.6	void cutlass::Kernel (matmul)
32.8	5131580	25	205263.2	205152.0	191263	214400	3947.9	layernorm_forward_kernel
1.3	205696	12	17141.3	17088.0	16992	17408	115.2	softmax_forward_kernel
0.9	141792	12	11816.0	11760.0	11712	12224	147.7	void cutlass::Kernel (attention)
0.7	104384	12	8698.7	8672.0	8544	8864	109.9	gelu_forward_kernel
0.6	88544	24	3689.3	3728.0	3360	4064	252.4	residual_forward_kernel
0.5	72448	12	6037.3	5984.0	5920	6304	124.6	permute_kernel
0.4	60640	12	5053.3	5056.0	4992	5088	25.4	unpermute_kernel
0.0	6464	1	6464.0	6464.0	6464	6464	0.0	encoder_forward_kernel

NVIDIA CUTLASS (combined bias) Execution Times

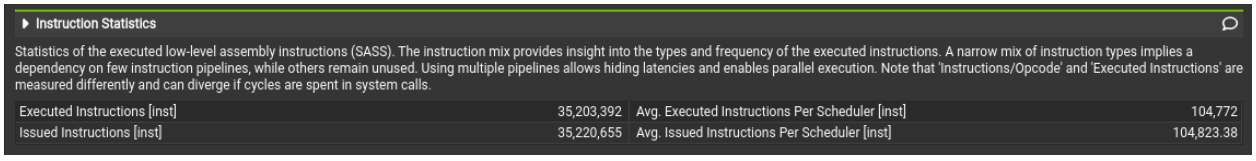
Using CUTLASS with a separate bias kernel results in a total runtime of 9796623 + 350208 = 10,146,831 ns for the matmul kernel. Combining the bias addition via the use of epilogue functions brings that down to just 9844619 ns. This may not be a particularly major speedup, but it's still a welcome improvement. The most obvious reason for this speedup is due to removing the overhead of a separate kernel launch, but there are likely more factors to it than just that. It's worth noting that both configurations (separate bias or combined bias) took more than double the time cuBLAS took which was only 4,266,175 ns. This reduction in performance is likely in large part due to the lack of tensor core usage which was something cuBLAS automatically handled. Regardless, we decided to delve into NSight compute in order to figure out why exactly the use of epilogue functions provided an improvement over separate kernels:



NVIDIA CUTLASS (separate bias) Instruction Statistics

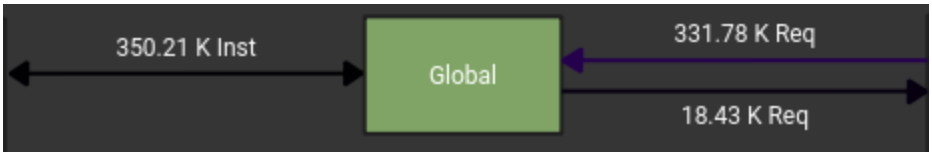


Separate Bias Addition Kernel Instruction Statistics

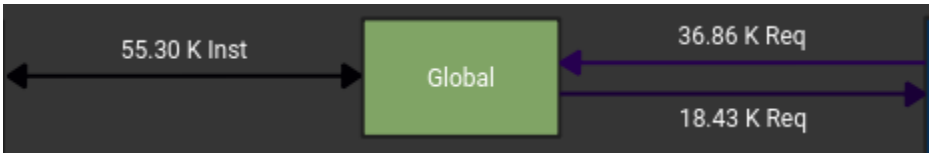


NVIDIA CUTLASS (combined bias) Instruction Statistics

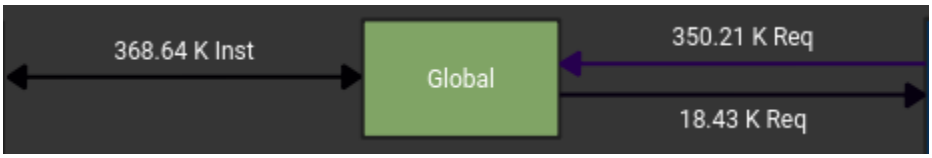
The total number of instructions executed with the two separate kernels is 35,044,992 + 727,040 = 35,772,032 which is about 500k more compared to the 35,203,392 executed by the fused version. This difference could stem from a variety of reasons including the recalculation of indices or additional writes/reads to/from device memory. The latter was especially evident when we looked at the memory charts:



NVIDIA CUTLASS (separate bias) Memory Chart



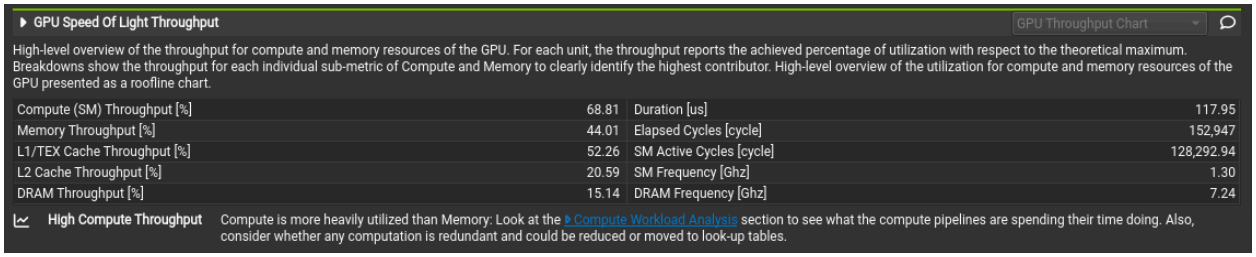
Separate Bias Addition Kernel Memory Chart



NVIDIA CUTLASS (combined bias) Memory Chart

We can see from the above that the two separate kernels required more total memory transfers in order to perform the same operations than the combined kernel did. This is because combining the kernels allows us to just add the bias to the values already stored in registers rather than having to write to global memory and read back again in a separate kernel. It's likely that this is one of the biggest reasons for the performance improvements of using the epilogue functions.

Next, we wanted to compare our kernel to other non-tensor core optimizations such as the joint register and shared memory tiling matmul kernel which took 23,853,824 ns in the best configuration we found during the sweep. This would allow us to see where it falls short compared to CUTLASS.



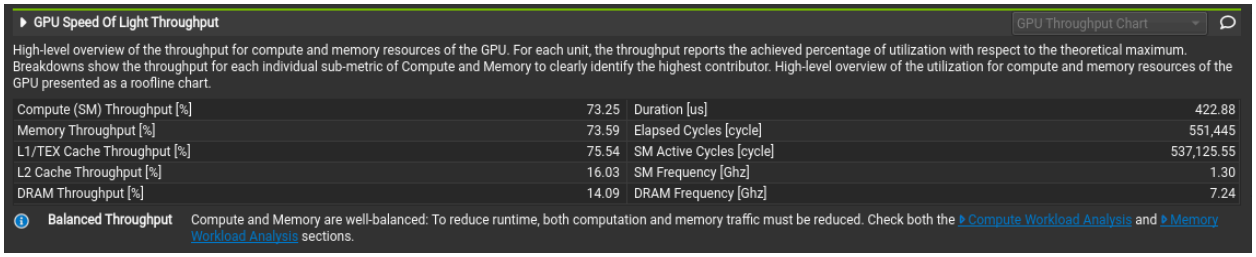
GPU Speed Of Light Throughput — GPU Throughput Chart

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 [%]	68.81	Duration [us]	117.95
Memory Throughput [%]	44.01	Elapsed Cycles [cycle]	152,947
L1/TEX Cache Throughput [%]	52.26	SM Active Cycles [cycle]	128,292.94
L2 Cache Throughput [%]	20.59	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	15.14	DRAM Frequency [Ghz]	7.24

High Compute Throughput Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

NVIDIA CUTLASS (combined bias) SoL Statistics



GPU Speed Of Light Throughput — GPU Throughput Chart

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.

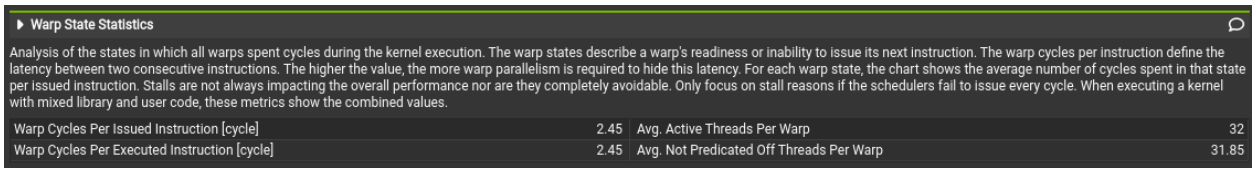
Compute (SM) Throughput [%]	73.25	Duration [us]	422.88
Memory Throughput [%]	73.59	Elapsed Cycles [cycle]	551,445
L1/TEX Cache Throughput [%]	75.54	SM Active Cycles [cycle]	537,125.55
L2 Cache Throughput [%]	16.03	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	14.09	DRAM Frequency [Ghz]	7.24

Balanced Throughput Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis sections.

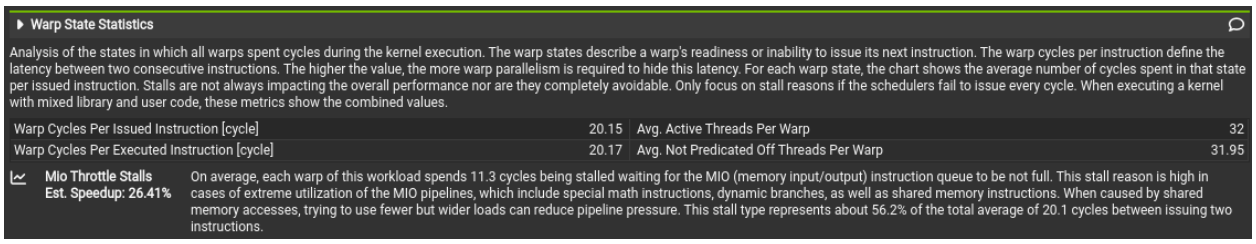
Joint Register and Shared Memory Tiling SoL Statistics

We can see that NVIDIA CUTLASS achieved a lower memory throughput which is expected and is indicative of better memory usage. Interestingly, NVIDIA CUTLASS also achieved lower compute

throughput than the joint register and shared memory tiling kernel which typically would lead to reduced performance. When we looked through the other sections, we noticed the achieved occupancy and L1/TEX hit rate was also significantly lower for CUTLASS (16.66% vs 37.13% for achieved occupancy and a mere 7.20% vs 80.07% for L1/TEX hit rate), so we were left wondering how exactly CUTLASS managed to achieve better performance. When we looked even further, we noticed that there was one key metric that saw a drastic improvement:



NVIDIA CUTLASS (combined bias) Warp State Statistics



Joint Register and Shared Memory Tiling Warp State Statistics

The number of warp cycles per instruction was around 8x fewer for CUTLASS than it was for joint register and shared memory tiling. This indicates that even despite the lower occupancy and cache hit rate, CUTLASS somehow manages to achieve better latency hiding. We're not sure exactly how it manages to do this, but it could be due to a variety of factors such as better instruction scheduling, pipelining, reduced bank conflicts, and more. This is an incredibly interesting result and shows that we still have a lot we can do to improve our own optimizations. If we were to do this project again, we could look through CUTLASS articles and its source code in order to get a better understanding of how and why it works, and then apply those same ideas to our hand-written optimizations.

KV Cache

We proposed utilizing a KV cache. A KV cache is a memory structure used in transformer models, particularly for large language models (LLMs), to store the key and value tensors from self-attention layers. Utilizing a KV cache will significantly reduce the time needed for self-attention.

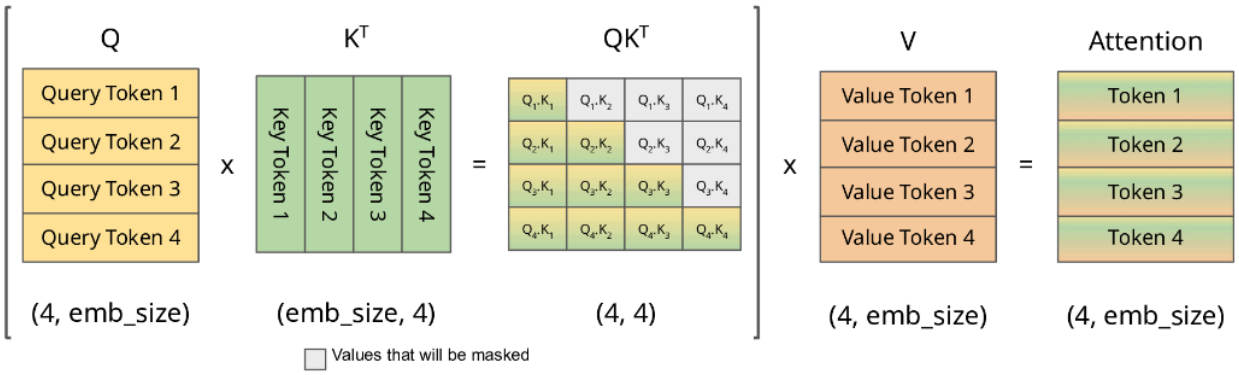


Figure 1. KV Cache

A KV Cache allows us to focus on only calculating the attention for the new token because a KV cache caches previous keys and values. This is a significantly less computation needed for attention. We will use the device's global memory for the caching purpose, and even though this increases our memory latency, we believe that reducing the computation redundancy is more important for the performance. This is because of the fact that we have to recompute key and value matrices every step. This results in $O(n^2)$ computation work. However, with a KV cache, we eliminate the need to recalculate the key and value matrices and reduce the computation work of $O(n)$. Given the parameter size, we believe this will result in significant speedup.

For the KV cache, we have modified the following files:

/kv_cache/attention.cuh
Attention forward now includes kv cache update kernel and takes in kv cache pointers and positional information

/kv_cache/encoder.cuh
Encoder now includes positional information

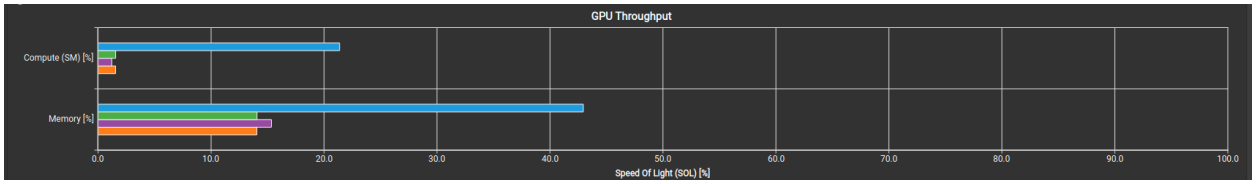
/gpt2.cuh
Gpt2 struct and forward were modified to include kv cache memory allocation and proper function call for encoder and attention forward

/next_token_generation.cu
Modified for the token generation with kv cache

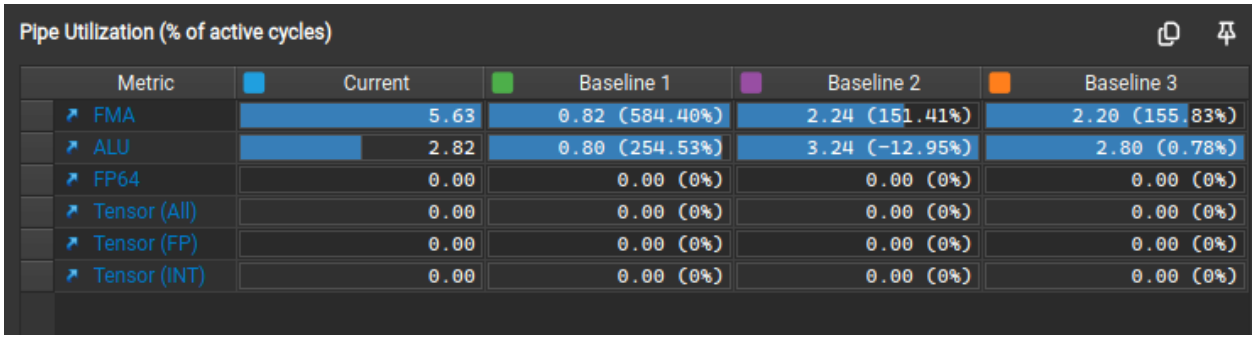
We were able to verify the kv_cache by running test_gpt2 and we verified it further by running token generation to observe proper speedup. For the grader to verify the results, the instructions are at the bottom of this section. Here we present gpt2 run both with kv cache and without kv cache. You could observe that the number of tokens/sec is more than double for kv cache compared to one without. This is because utilizing the kv cache was able to reduce the time for inferencing from 1403ms to 622ms,

resulting in 2.25x speedup. This demonstrates that the kv cache indeed reduces the computation work significantly.

```
+----------+
| GPT2 RUN |
+----------+-------------------------------------------------------------------------------------------------------------------------------------------------+
|
| INPUT TEXT: The quick brown fox jumps over the lazy dog. In a surprising turn of events, the fox decided to
|
| GENERATED TEXT:  sit down and eat the lazy dog. The lazy dog is very excited about this, and takes the lazy dog to the backyard. The lazy dog began laying on the ground, and started to run. The lazy dog stayed for a few minutes, and
|
| TIME FOR INFERENCING: 1403.621819 ms
|
| # TOKENS/SEC: 35.622
+-------------------------------------------------------------------------------------------------------------------------------------------------+

+----------+
| GPT2 RUN USING KV CACHE|
+----------+-------------------------------------------------------------------------------------------------------------------------------------------------+
|
| INPUT TEXT: The quick brown fox jumps over the lazy dog. In a surprising turn of events, the fox decided to
|
| GENERATED TEXT:  take the opportunity to pee.

"Hey, wait a moment! I had a little pee in my hair last night! I just can't believe I did that!"

To the surprise of all, the fox didn't immediately grab her
|
| TIME FOR INFERENCING: 622.080033 ms
|
| # TOKENS/SEC: 80.376
+-------------------------------------------------------------------------------------------------------------------------------------------------+
```



In the Nsight Compute, we were able to observe that kv cached attention overall is less memory bound because we were able to eliminate kernel launch overhead by using kernel fused attention. But more importantly, we were able to observe less computation as kv cache removes the overhead. This only demonstrates less computation in ALU because we used test_gpt2.cuh for this profiling due to time constraint.



Pipe Utilization (% of active cycles)

Metric	Current	Baseline 1	Baseline 2	Baseline 3
FMA	5.63	0.82 (584.40%)	2.24 (151.41%)	2.20 (155.83%)
ALU	2.82	0.80 (254.53%)	3.24 (-12.95%)	2.80 (0.78%)
FP64	0.00	0.00 (0%)	0.00 (0%)	0.00 (0%)
Tensor (All)	0.00	0.00 (0%)	0.00 (0%)	0.00 (0%)
Tensor (FP)	0.00	0.00 (0%)	0.00 (0%)	0.00 (0%)
Tensor (INT)	0.00	0.00 (0%)	0.00 (0%)	0.00 (0%)

Here we illustrate the speedup from kv cache for longer token sequences.

```
+=============================================================================
==+
|                  SPEEDUP COMPARISON TABLE                                  |
+=============================================================================
==+
| Tokens      | KV Cache (ms)   | Baseline (ms)   | Speedup    | Time Saved |
+-------------|-----------------|-----------------|------------|------------+
| 25          |        261.23 |        478.69 |     1.83x |    217.46 ms |
| 50          |        566.34 |       1259.45 |     2.22x |    693.11 ms |
| 100         |       1318.12 |       3639.31 |     2.76x |   2321.19 ms |
| 200         |       3384.00 |      13193.83 |     3.90x |   9809.83 ms |
+=============================================================================
==+
```

We tested kv cache speedup increases with various sequence lengths and we were able to observe that kv cache indeed becomes more efficient with the sequence length increasing. This is due to the fact that baseline has O(T^2) complexity per generation whereas kv cache has O(T) with the caching. We were able to observe an average of 2.87x speedup across all sequence lengths that we tested. Note that this was tested using an AI-generated testcode that modified test_gpt2.cuh to iterate multiple times.

In order to test the kv cache, we have added the compiler flag KV_CACHE for /gpt2.cuh and

/next_token_generation.cu. So just add -DKV_CACHE in the Makefile to enable the use of the kv cache.

E.g.
next_token_generation: next_token_generation.cu $(KERNELS)
    $(NVCC) $(CFLAGS) $(CUBLAS_INCLUDES) -c next_token_generation.cu -o next_token_generation.o **-DKV_CACHE**
    $(NVCC) $(CFLAGS) -o next_token_generation next_token_generation.o $(CUBLAS_LIBS)

The same could be done for test_gpt2.
To get the speedup comparison, run test_kv_gpt2.

* Note that this needs to be done in kv_cache branch

Split-K

Split-K is a parallelization technique that divides the contraction dimension (K, or C in our case) of a matrix multiplication across multiple kernel launches, allowing more concurrent execution and better GPU utilization. Instead of having each thread block compute a complete dot product over all C elements, we split the work so multiple kernels each compute partial dot products over non-overlapping ranges of C, which are then combined in a final reduction step. Split-K attempts to maximise SM utilization by launching multiple instances of the kernel simultaneously, each processing a slice of the concurrent work across the GPU's SMs. Our implementation modifies the baseline matmul kernel by adding c_start and c_end that specify which portion of the C dimension to process. The matmul_forward function launches 2 separate kernel instances where split 0 processes 0 to C/2 and split 1 processes the other half if K were to be 2 for example. Each split kernel writes its partial results to a temporary buffer. After the splits complete a reduction kernel combines the partial results.

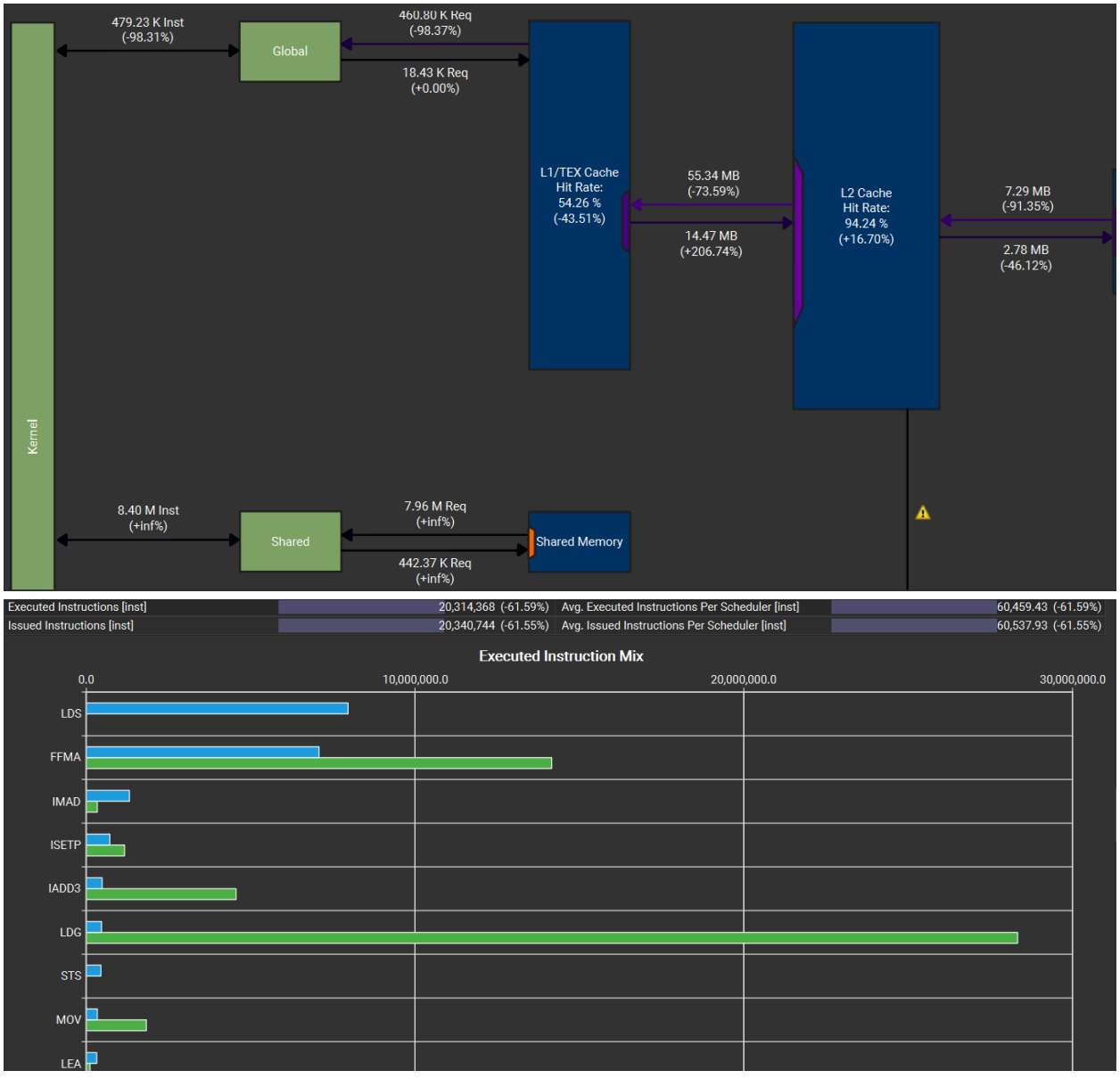
An alternative version of the split-k algorithm would involve atomicAdds at each of the output locations to circumvent the reduction at the end. We implemented both these cases, and decided upon the reduction version with a split of just 2 which will be justified with our results below.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	
50.6	24412851	49	498221.4	461311.0	127584	7281141	999386.3	matmul_forward_kernel(flo
17.4	8390676	12	699223.0	699087.0	698207	700991	835.3	vaccum_kernel(float *, fl
17.3	8369044	12	697420.3	696959.0	695711	700735	1483.2	preatt_kernel(float *, fl
10.6	5117792	25	204711.7	205120.0	191136	208960	3485.7	layernorm_forward_kernel(
2.3	1094878	12	91239.8	91248.0	90111	92192	735.7	permute_kernel(float *, f
0.8	368288	12	30690.7	30672.0	30496	31008	157.0	softmax_forward_kernel(fl
0.7	340704	12	28392.0	28368.0	28256	28544	85.3	unpermute_kernel(float *,
0.2	97664	24	4069.3	4000.0	3520	4672	462.2	residual_forward_kernel(f
0.2	90944	12	7578.7	7584.0	7392	7712	105.5	gelu_forward_kernel(float
0.0	6496	1	6496.0	6496.0	6496	6496	0.0	encoder_forward_kernel(fl

Above is the kernel execution time when K=2 from the reduction implementation, indicating 2 separate launches per "half" of the C-dimension which we can see from double the instances.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	
52.6	27157620	98	277118.6	251120.0	79328	3639326	495631.2	matmul_forward_kernel(flo
16.3	8388604	12	699050.3	698912.0	697983	700448	765.3	vaccum_kernel(float *, fl
16.2	8369309	12	697442.4	697152.0	695679	700160	1359.5	preatt_kernel(float *, fl
9.9	5122940	25	204917.6	205440.0	191232	209184	3518.5	layernorm_forward_kernel(
2.1	1070432	12	89202.7	89296.0	88224	90112	537.6	permute_kernel(float *, f
1.2	622848	49	12711.2	9120.0	2176	266304	37312.3	add_arrays(float *, const
0.7	368640	12	30720.0	30688.0	30464	31008	186.6	softmax_forward_kernel(fl
0.7	340832	12	28402.7	28384.0	28256	28704	117.7	unpermute_kernel(float *,
0.2	86080	24	3586.7	3632.0	2912	4160	409.0	residual_forward_kernel(f
0.2	83232	12	6936.0	6976.0	6144	7520	371.7	gelu_forward_kernel(float
0.0	6720	1	6720.0	6720.0	6720	6720	0.0	encoder_forward_kernel(fl

Above is the kernel execution times with split-k using atomicAdds, and one grid launch per matmul kernel call. K = 4 for this run.



You can see how the split-K implementation uses more shared memory loads instead of global and uses

half the number of arithmetic operations since its split in two.

Performance results showed dramatic improvement with the baseline matmul executing at approximately 80ms per forward pass across 49 matmul calls and Split-K with K_SPLITS equals 2 executing at approximately 25ms per forward pass, yielding a 3.2x speedup. This speedup comes from increased block-level parallelism rather than algorithmic improvements. Each split kernel does half the work by iterating over only C divided by 2 elements instead of the full C equals 128, but both kernels execute concurrently on different streaming multiprocessors. The GPU can now run twice as many blocks simultaneously, filling more SMs that were previously idle due to insufficient parallelism in the baseline grid configuration. Nsight Systems profiling confirmed this with the timeline showing both split kernels executing simultaneously rather than sequentially. The slight overhead from the reduction kernel, which performs a simple element-wise addition over B times T times OC elements, is negligible at approximately 1 to 2ms compared to the parallelism gains.

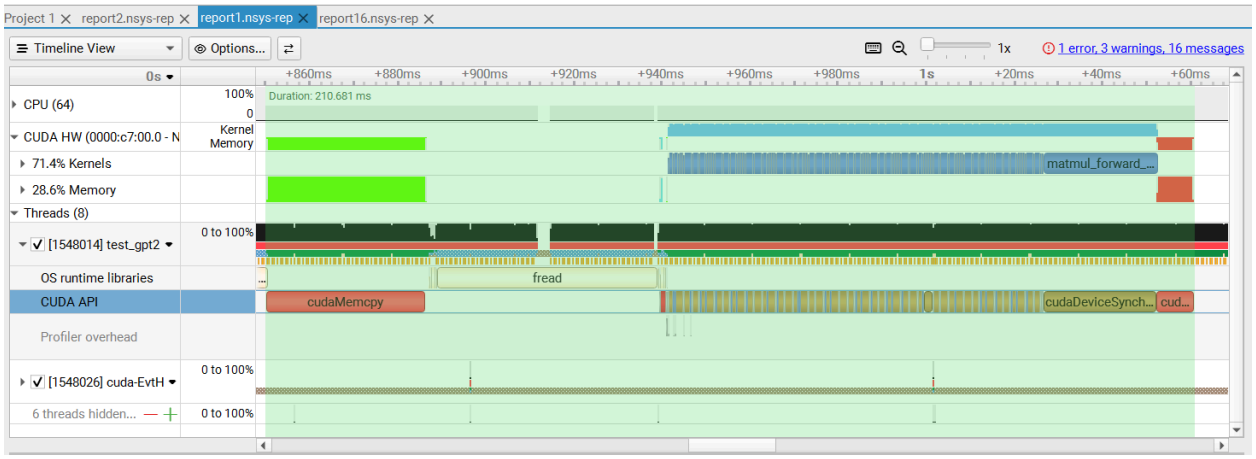
We experimented with different K_SPLIT values to find the optimal configuration, 2 provided the best speedup with minimal reduction overhead. With the temporary buffers, we needed to allocated an entire output size buffer using the reduction method, which is obviously harmful to our performance with higher K values. Higher values led to clearly diminishing returns as reduction overhead began to outweigh the parallelism benefits.

The implementation encountered significant challenges initially. Our first version called cudaMalloc and cudaFree for the partial sums buffer inside matmul_forward, which executes 49 times per forward pass. These memory allocation functions are synchronous operations that completely stall the GPU pipeline, resulting in catastrophic performance degradation to 547ms, a 6.8x slowdown instead of the expected speedup. We also initially included cudaDeviceSynchronize after each kernel launch, which serialized execution and prevented the split kernels from running concurrently, eliminating any parallelism benefits. After identifying these bottlenecks through Nsight Systems profiling, which showed massive gaps in GPU utilization and sequential kernel execution, we implemented the persistent buffer strategy and removed unnecessary synchronization calls. CUDA kernels launched on the default stream execute asynchronously and maintain launch order, so the reduction kernel naturally waits for both split kernels to complete without explicit host-side synchronization. This fix brought performance from 547ms down to the final 25ms.

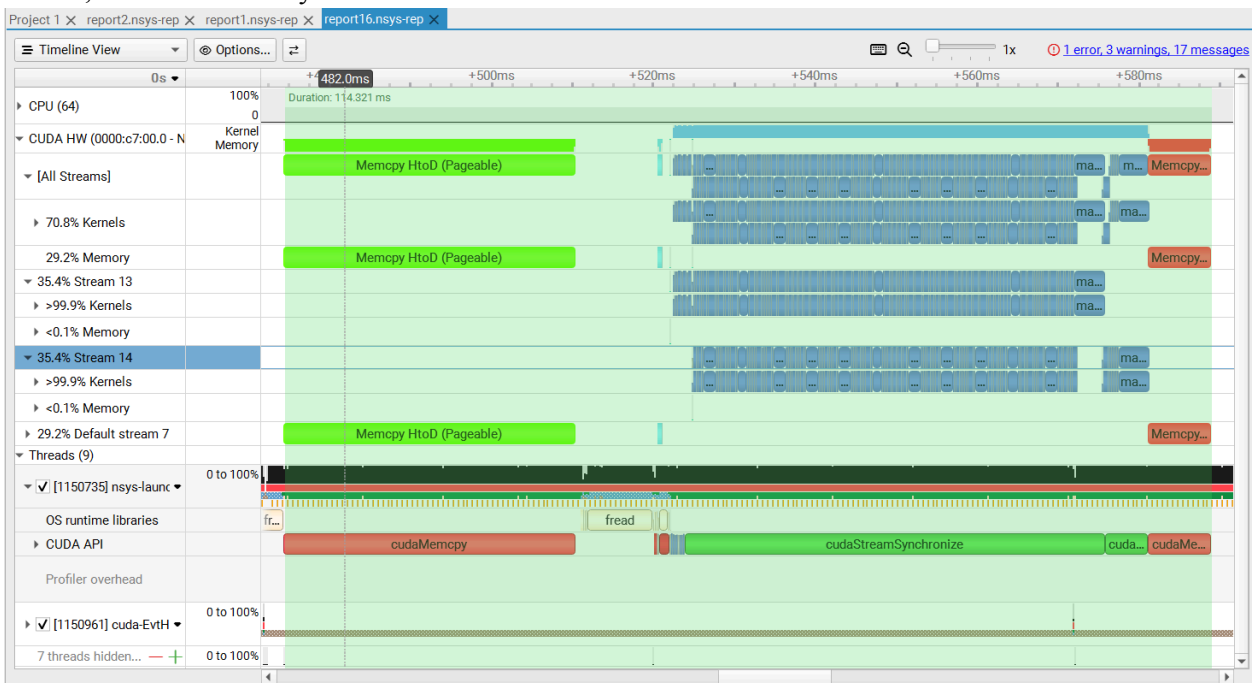
CUDA Streams

CUDA Streams optimization targets parallelism between batches by pipelining independent mini batches across multiple streams instead of feeding the whole batch through the model on the default stream. In the baseline implementation, all the work was done sequentially on a single stream which left gaps where the GPU was idle. To address this problem, we introduced an array of CUDA streams in the GPT2 struct, added pinned host memory for inputs to enable asynchronous host to device transfers, and restructured gpt2_forward so batch dimension B was split into mini batches. For each stream we compute a slice of the batch and only copy that slice’s tokens into memory, then we run the entire forward pass for that slice on the corresponding stream.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	
39.3	41757348	98	426095.4	338720.5	104576	3760388	510931.8	matmul_forward_kernel(float *, const float *, cor
24.6	26093367	24	1087223.6	937281.0	554752	1540385	362470.5	att_kernel(float *, float *, float *, int, int, i
15.4	16388431	24	682851.3	626273.0	549824	1129409	172486.5	preatt_kernel(float *, float *, float *, int, int
11.5	12200302	50	244006.0	185680.5	169088	489792	112978.1	layernorm_forward_kernel(float *, float *, float
3.5	3737251	24	155718.8	103872.0	68736	320352	83870.2	permute_kernel(float *, float *, float *, const f
3.5	3723811	24	155158.8	144256.0	123137	276481	38015.7	softmax_forward_kernel(float *, float, const floa
1.3	1432706	24	59696.1	53088.0	20480	133408	29379.0	unpermute_kernel(float *, float *, int, int, int,
0.5	536448	24	22352.0	7168.0	4576	56576	23251.6	gelu_forward_kernel(float *, const float *, int)
0.3	285216	48	5942.0	4224.0	2304	34432	5630.2	residual_forward_kernel(float *, float *, float *
0.0	10208	2	5104.0	5104.0	4576	5632	746.7	encoder_forward_kernel(float *, const int *, cons



Observing the Nsight Systems timeline for the baseline, we saw large gaps where the GPU was completely idle during the initial cudaMemcpy host-to-device transfer of input tokens, and additional idle periods between kernel launches due to synchronization overhead. The fundamental insight is that different batch elements are completely independent, meaning batch element 0 can be in the encoder while batch element 1 is being transferred from host memory, and batch element 2 is executing attention kernels, all simultaneously.



Nsight Systems timeline comparison

Nsight Systems timeline now displays multiple CUDA streams, each with its own sequence of encoder and transformer kernels that overlap in time instead of forming a single serialized chain. At the same time though, the report showed the clear limitations of this approach; our matmul and attention kernels remain largely memory bound, streams can only rearrange and overlap work, they can not exceed the hardware's peak DRAM throughput. In practice the speedup would be the most noticeable for larger batches. The time that the kernels were running went from 110ms to 59ms, which was the using 2 streams, this turned out to be the fastest by balancing overhead while trying to achieve DRAM saturation. Increasing the number of streams beyond two did not improve performance further, suggesting that global memory bandwidth becomes the dominant bottleneck once sufficient concurrency is introduced. Overall, CUDA streams proved to be a powerful optimization for our batch-based workload, significantly improving GPU utilization and execution time while maintaining correctness through careful attention to memory offsets and synchronization semantics. The technique composes well with other optimizations like Split-K and tiling, as streams provide grid-level concurrency while those optimizations improve per-kernel efficiency, and the combination of both achieves better overall performance than either alone.

To check for correctness just run gpt2.cuh on the stream kernels in kernels_stream in the luke11 branch. I changed gpt2.cuh to take in the stream kernels so all you have to do is run make and srun ./test_gpt2

```
// kernels
#include "kernels_stream/attention.cuh"
#include "kernels_stream/encoder.cuh"
#include "kernels_stream/gelu.cuh"
#include "kernels_stream/layernorm.cuh"
#include "kernels_stream/matmul.cuh"
#include "kernels_stream/residual.cuh"
```

Conclusion and Future Directions

Working on this project has been a great experience and well directed to let us explore the realms of what accelerates GPU architecture towards its SOL potential. Lectures and tests teach us a ton about the theory and computations behind how we could run a process at max efficiency, but developing and profiling these optimizations really got us thinking about why it all works, and what we could really push for the best result available.

The obvious next steps would be a careful analysis of which of these optimizations would produce the strongest results when paired together, and how we could make the entire process run the fastest it can. Also, more consideration towards differing input parameters such as number of tokens or size of our data analyzed is crucial in true all around functionality, as we may have unintentionally optimized for the constant data we are passed in these test cases.

This work really got us excited about the inner workings and application of every concept we learned in class, and our entire group is grateful to have been able to work in such a structured manner with this material.