GPT-2 Optimization

1. Introduction

Hello and welcome to GPA-2's introduction and guide to optimizing the GPT-2 Model!

GPT-2 (Generative Pre-trained Transformer 2) was fully released by OpenAI on November 5th, 2019. Without going too deep into the specifics of how it works, one critical factor to note is that the scale and resources provided to GPT-2 was a critical factor in performance and speed of program execution. This also means that it is critical that GPT-2 uses its resources in an optimized way in order to get every drop of performance allowed. Also a quick note, GPT-2 uses a LOT of GEMM (General Matrix Multiplication), which is something that GPUs are incredibly well-prepared for. As a result, we decided to take the initial CPU implementation provided for GPT-2 and created a GPU parallel implementation that aims to be more efficient and performant than its serial counterpart.

Taking a look of what's to come, we'll go over:

- What the Transformer Architecture is
- How we implemented these parts for the forward pass in the architecture
- What optimizations we considered and how we optimized parts of the Matrix Multiplication code
- Advanced Optimizations to consider
- Discussion

2. Understanding the Transformer Architecture

So, what are we going to be looking at for the GPT-2 forward pass? Well, the main layers in the transformer that we'll be working on are:

1. Attention
2. Encoder
3. GELU
4. Layernorm
5. MatMul
6. Residual
7. Softmax

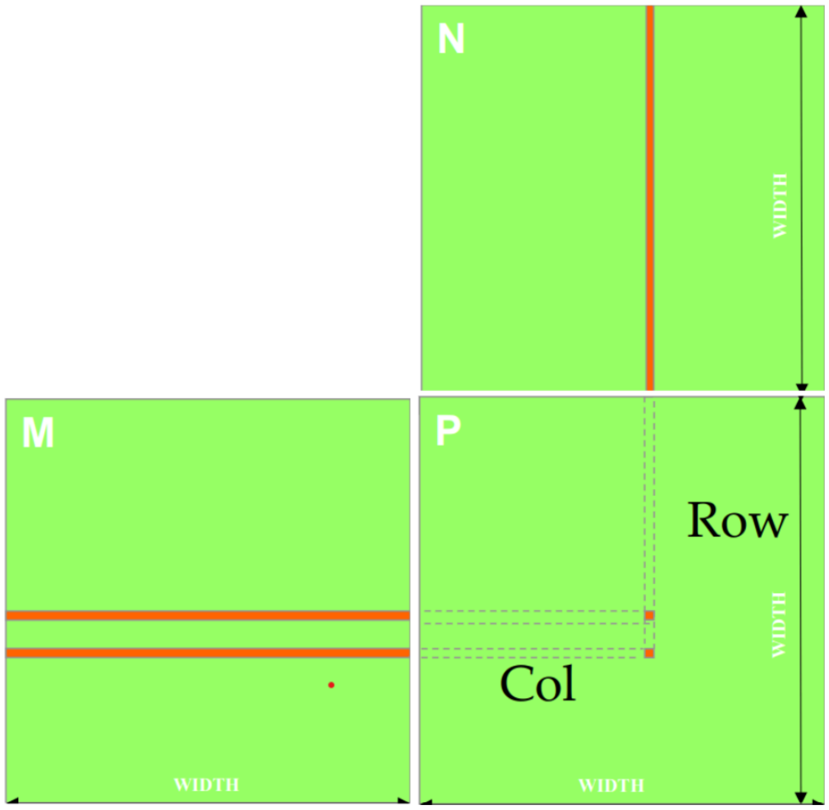
Each layer runs the following kernels in the following order:

- layernorm_forward(inputs);
- matmul_forward(inputs);
- attention_forward(inputs);
- matmul_forward(inputs);
- residual_forward(inputs);
- layernorm_forward(inputs);
- matmul_forward(inputs);
- gelu_forward(inputs);
- matmul_forward(inputs);
- residual_forward(inputs);

And for each layer, we want to optimize the performance such that our improvements are able to ensure the forward pass performs with high levels of efficiency. After looking at each of the kernels, it became clear that the matmul_forward kernel has a heavy workload and that optimizing it would reduce our runtime by a large amount.

3. Baseline Implementation: The Starting Point

So, let's first understand what the initial matrix multiplication implementation is. We have a matrix M and a matrix N, and we need to create an output matrix P that results from the multiplication of M and N. To achieve this, we take a row of M, and a column of N, and take the dot product of the two in order to get a single element of P. We iterate through the rows with a specific column of N and we get a column of P, we iterate through the columns of N with a specific row of M and we get a row of P, and if we continue this process we get a matrix P. The diagram below should show how much of each matrix M and N we need in order to get a single element of P.



So, to code this for the GPT-2 forward pass, we can get the following naive CPU implementation.

```
void matmul_forward_cpu(float* out,
                        const float* inp, const float* weight, const float*
bias,
                        int B, int T, int C, int OC) {
    // inp is (B,T,C), weight is (OC, C), bias is (OC)
    // out will be (B,T,OC)
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            float* out_bt = out + b * T * OC + t * OC;
            const float* inp_bt = inp + b * T * C + t * C;
            for (int o = 0; o < OC; o++) {
                float val = (bias != NULL) ? bias[o] : 0.0f;
                const float* wrow = weight + o*C;
                for (int i = 0; i < C; i++) {
                    val += inp_bt[i] * wrow[i];
                }
                out_bt[o] = val;
            }
        }
    }
}
```

However, we can translate this exact CPU code into GPU code as:

```
__global__ void matmul_forward_kernel(float *out, const float *inp, const
float *weight,
                                                    const float *bias, int C, int OC){
    // Implement this
    int myB = blockIdx.x;
    int myT = blockIdx.y;

    int T = gridDim.y;
    int B = gridDim.x;
    int o = blockIdx.z * blockDim.x + threadIdx.x;

    float *out_bt = out + myB * T * OC + myT * OC;
    const float *inp_bt = inp + myB * T * C + myT * C;

    if ((o < OC) && (myB < B) && (myT < T))
    {
        float val = (bias != NULL) ? bias[o] : 0.0f;
        const float *wrow = weight + o * C;
        for (int i = 0; i < C; i++)
        {
            val += inp_bt[i] * wrow[i];
        }
        out_bt[o] = val;
    }
}
```

The above matmul_forward_kernel does the exact same thing as the CPU implementation however it does so in a parallelized way and achieves faster performance.

However! There’s still some room for improvement. We ran NVIDIA’s Nsight Compute profiling on our parallel implementation of GPT-2, and from the high-level overview section “GPU Speed Of Light Throughput”, we noticed that our Compute Throughput and Memory Throughput are not optimal and can further be optimized. From the numbers below, we can see some bottlenecks that we can try and focus on.

Compute (SM) Throughput [%]	10.27
Memory Throughput [%]	84.74

These bottlenecks include:

- We've been using low computational throughput, this could mean that we're not doing enough calculations with the given data.

Some other results from our profiling that weren't included above also hinted that:

- We have high warp stall cycles
- Excessive global memory access

These are all issues that come with our very simple implementation of matmul, however once improved we should be able to see huge levels of improvement.

4 Tensor Core Optimization

What are Tensor Cores?

At the heart of the computational efficiency in modern NVIDIA GPUs lies a game-changing innovation: Tensor Cores. You can tell from the "core" in the name that they are processing units located in GPUs that are specialized for **fused multiply-add** (FMA) operations. Note, each Streaming Multiprocessor (SM) contains several tensor cores for FMA like: $C = A * B + C$

Take the Ampere 40 GPU with computation capacity 8.6 as an example, the related statistics are:

GPU Architecture	NVIDIA Ampere
SMs	84
Tensor Cores/SM	4 (3rd Gen)
Tensor Cores/GPU	336 (3rd Gen)

How do Tensor Cores work?

For this following operation: $C = A * B + C$

1. Instead of computing one output element with each thread, tensor cores divide the input matrices A and B into tiles (tile structure depending on the precision you are using, typically $16 * 16 = 256$).

2. This looks like the shared memory tiling matrix multiplication, but the tiles are loaded into Warp Matrix Multiply-Accumulate (WMMA) Fragments instead of shared memories. The WMMA Fragments are special data structures that **each warp can work with**.

3. The multiplication of these tiles is performed with FMA operations, and the results are accumulated directly into the destination matrix C.

In summary, Tensor Cores accelerate the FMA with following features:

- Warp-level parallelism: Tensor cores operate at the warp level with 32 threads working together on a $16x16$ tile of matrix multiplication in a coordinated manner using tensor core's high-throughput WMMA instructions.
- Special precision: there are several supported precisions for tensor cores. Precisions like FP16 and TF32 take less memory bandwidth and can be computed faster.

Tensor Core API calls

With all of the advantages mentioned above, let's get to know the tensor core APIs and how to use them!

1. **wmma::fragment** - We use fragment to store tiles A, B, and C. There are several parameters to determine:
    a. you should specify `wmma::matrix_a` for representing the left hand side of matrix multiplication, `wmma::matrix_b` for the right-hand side, and `wmma::accumulator` for the output.
    b. `WMMA_M`, `WMMA_N`, `WMMA_K` are the three dimensions for tile matrix multiplication with dimension $(M * K) * (K * N) = (M * N)$.**The specific dimension you should use depends on your choice of precision** [link for precision and dimension].
    c. `wmma::row_major` means that the data in the fragment are stored in row major, meaning that elements in the same row are stored consecutively, and `wmma::col_major` vice versa. This parameter can determine how

you are loading and accessing the data in the fragment.

2. `wmma::load_matrix_sync`: After declaring the fragments, we want to load in data to the tiles and perform FMA in a tiled pattern. This method is responsible for data loading.
    a. `wmma::fragment`: the first parameter needed is the fragment that we are loading data into.
    b. `src`: the second parameter is the starting pointer to the first element to be loaded.
    c. `ldm`: the leading dimension, or the stride for between the loading in each iteration.

Why does data layout matter? Let's say your entire matrix B is transposed, meaning that B is essentially stored in **column major**. Thus when loading into the declared column major fragment `matrix_b` you need to be very careful about the starting point and the leading dimension. Generally speaking, if a matrix is stored in row_major then the leading dimension should be the number of columns of the matrix, otherwise the number of rows.

3. `wmma::sync`: This is the function that actually performs the warp matrix multiply-accumulate synchronization. At this step all 32 threads in a warp work together to perform the matrix multiplication on a $16 * 16$ tile.

4. `wmma::store_matrix_sync`: you can use this function to store the computed result back to the output matrix.

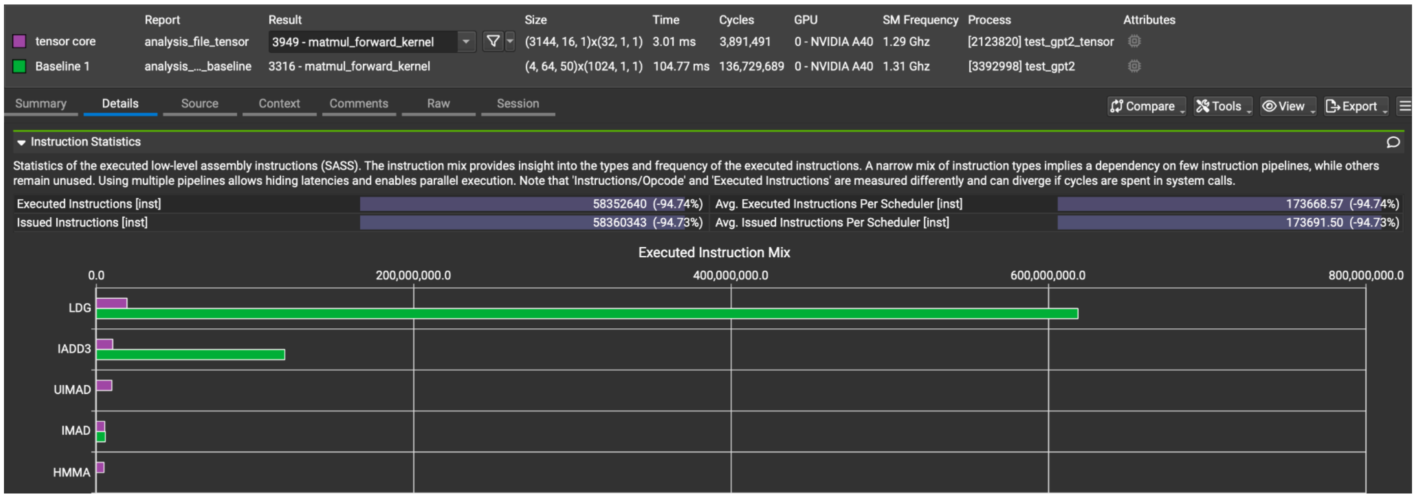
By combining all of the API calls, an example of tensor core implemented matrix multiplication kernel might look like this:

```
#define WMMA_M 16
#define WMMA_N 16
```

```
#define WMMA_K 16

__global__ void matmul_forward_kernel(const half* A, const half* B, float*
C,
                                        int M, int N, int K) {
    // A: half precision with dimension M x K, row_major stored
    // B: half precision with dimension N x K, it is the transposition the
of original B, so we use col_major
    // C: FP32 precision with dimension M x N, row_major stored
    int warpN = blockIdx.x;
    int warpM = blockIdx.y;

    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half,
wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half,
wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float>
c_frag;

    wmma::fill_fragment(c_frag, 0.0f);

    for (int i = 0; i < K; i += WMMA_K) {
        int aRow = warpM * WMMA_M;
        int aCol = i;
        int bCol = i;               // Shared dim (K)
        int bRow = warpN * WMMA_N;      // Columns of original B → Rows of
transposed B

        wmma::load_matrix_sync(a_frag, &A[aRow * K + aCol], K);
        wmma::load_matrix_sync(b_frag, &B[bRow * K + bCol], K);
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
    }
    int cRow = warpM * WMMA_M;
    int cCol = warpN * WMMA_N;

    wmma::store_matrix_sync(&C[cRow * WMMA_N + cCol], c_frag, WMMA_N,
wmma::mem_row_major);
}
```

Profiling results

For our largest matmul kernel performing matrix multiplication between A: (4 x 64) x 768, B: 2304 x 768, the tensor core kernel above significantly reduced the runtime from 104.77ms to 3.01ms. Let's dive deeper into the profiling result to see where the improvement comes from.



	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
🟪 tensor core	analysis_file_tensor	3949 - matmul_forward_kernel	(3144, 16, 1)x(32, 1, 1)	3.01 ms	3,891,491	0 - NVIDIA A40	1.29 Ghz	[2123820] test_gpt2_tensor	
🟩 Baseline 1	analysis_..._baseline	3316 - matmul_forward_kernel	(4, 64, 50)x(1024, 1, 1)	104.77 ms	136,729,689	0 - NVIDIA A40	1.31 Ghz	[3392998] test_gpt2	

Instruction Statistics

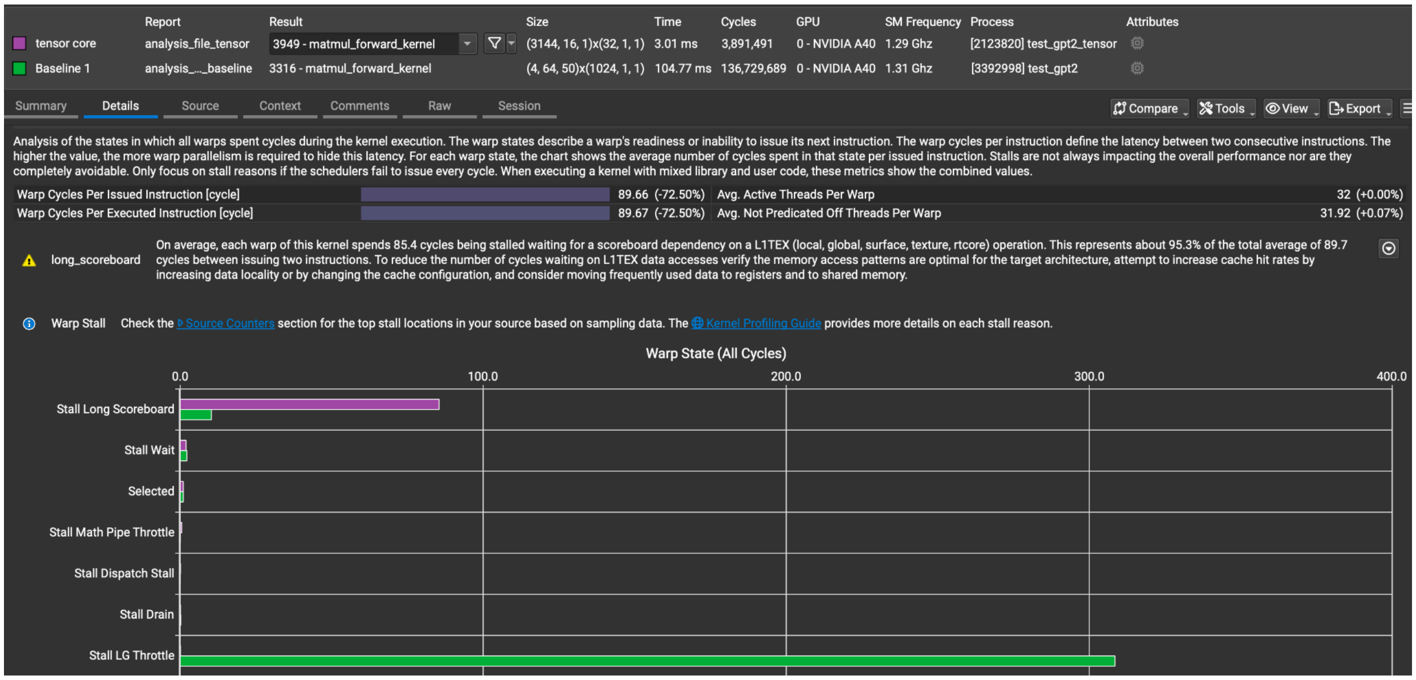
Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines 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]	58352640 (-94.74%)	Avg. Executed Instructions Per Scheduler [inst]	173668.57 (-94.74%)
Issued Instructions [inst]	58360343 (-94.73%)	Avg. Issued Instructions Per Scheduler [inst]	173691.50 (-94.73%)

From the comparison we can see that tensor core reduces runtime in a most straightforward way: **reducing the total number of instructions by 94.7%**.

So, what are the instructions reduced? The most significant ones are the `LDG` instruction and the `IADD3` instruction. `LDG` is instruction loading form global memory, and `IADD3` is responsible for the integer addition used in for loops. This shows that our tensor core implementation significantly reduces the instructions needed for data loading and for loop with the tiled pattern matrix multiplication.

However, this doesn't mean that tensor core is the perfect answer. There are several room for improvement.



	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
🟪 tensor core	analysis_file_tensor	3949 - matmul_forward_kernel	(3144, 16, 1)x(32, 1, 1)	3.01 ms	3,891,491	0 - NVIDIA A40	1.29 Ghz	[2123820] test_gpt2_tensor	
🟩 Baseline 1	analysis_..._baseline	3316 - matmul_forward_kernel	(4, 64, 50)x(1024, 1, 1)	104.77 ms	136,729,689	0 - NVIDIA A40	1.31 Ghz	[3392998] test_gpt2	

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 warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp 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 avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	89.66 (-72.50%)	Avg. Active Threads Per Warp	32 (+0.00%)
Warp Cycles Per Executed Instruction [cycle]	89.67 (-72.50%)	Avg. Not Predicated Off Threads Per Warp	31.92 (+0.07%)

long_scoreboard On average, each warp of this kernel spends 85.4 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture, rtcore) operation. This represents about 95.3% of the total average of 89.7 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to registers and to shared memory.

Warp Stall Check the Source Counters section for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

The warp stalling statistics show that although tensor core has freed our kernel from heavy global memory access instructions (Stall LG Throttle), instructions like HMMA used by `wmma::mma_sync` still has to **wait for the global memory data to be loaded**, leading to **high Stall Long Scoreboard**.

We can also see that since we were only using 32 threads per block, the occupancy is very low.

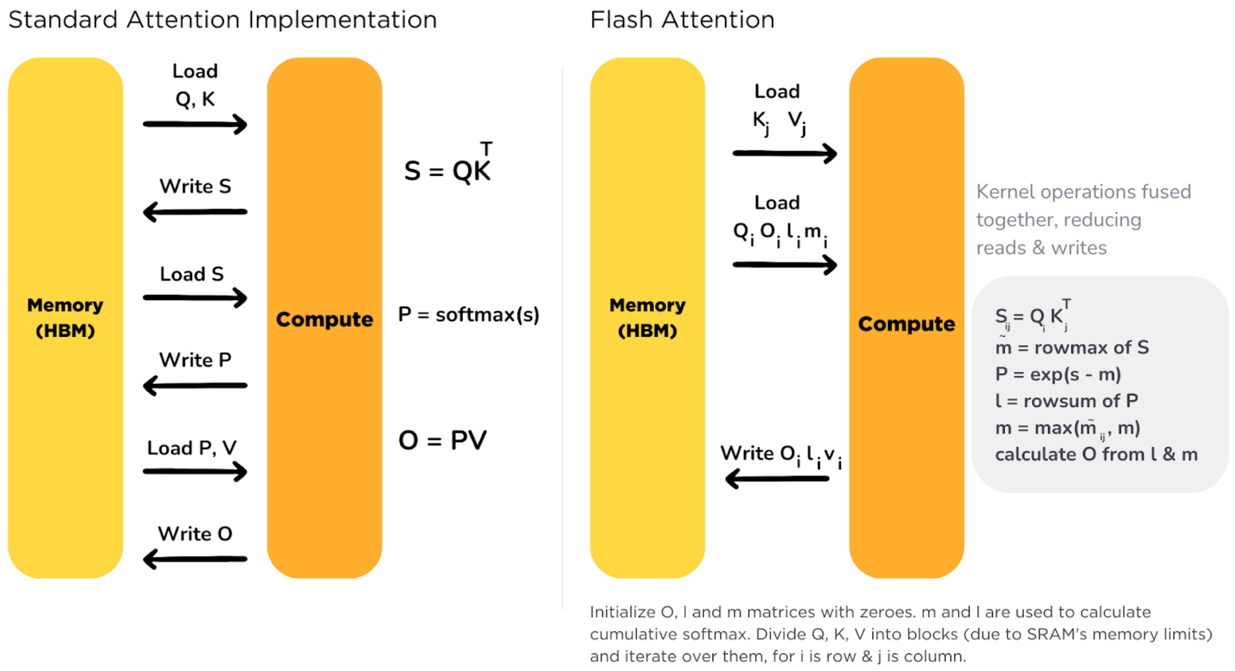
1. Can we decrease global memory loading by loading in shared memory?
    a. For the fragments, since we are not reusing the shared memory it may not be effectively
2. Can we simply increase the block size?
    a. Not quite, the tensor core works on a warp so even through you may launch let's say 128 threads per block, still only 32 of them actually do the tiled matrix multiplication if you are only launching 1 tile instead of 4 tiles.

5. Advanced Optimizations

5.1 Local/Windowed Attention

Motivation

In the baseline implementation, a full attention matrix of shape (T×T) is computed and materialized for each head, requiring O(T²) memory and computation. This quickly becomes a bottleneck as sequence length T increases. To alleviate this, we explore a more efficient variant: Flash Attention. It is a technique that eliminates the need to store the attention matrix explicitly and instead fuses computation of attention scores, softmax normalization, and weighted sum into a single kernel. This not only improves memory locality but also reduces global memory pressure, which is critical for GPU performance.



Standard Attention Implementation
Memory (HBM)
Compute
Load Q, K
Write S
Load S
Write P
Load P, V
Write O
$S = QK^T$
P = softmax(s)
O = PV
Flash Attention
Memory (HBM)
Compute
Load K_j V_j
Load Q_i O_i l_i m_i
Write O_i l_i v_i
Kernel operations fused together, reducing reads & writes
$S_{ij} = Q_i K_j^T$
$\tilde{m}$ = rowmax of S
P = exp(s - m)
l = rowsum of P
m = max($\tilde{m}_{ij}$, m)
calculate O from l & m
Initialize O, l and m matrices with zeroes. m and l are used to calculate cumulative softmax. Divide Q, K, V into blocks (due to SRAM's memory limits) and iterate over them, for i is row & j is column.

Implementation Details

- Input Format: We begin with a tensor of shape (B, T, 3*C), representing the concatenated queries, keys, and values. This tensor is split into Q, K, and V using a permute_kernel, similar to baseline attention, and stored in contiguous memory in (B, NH, T, HS) layout (where HS = C / NH).

- Kernel Logic: Inside the flash_attention_forward_kernel, each thread handles a specific (B, NH, T) triplet and computes the attention for a single query position (qt). The steps are:

    1. Compute Attention Scores: Iterate over all previous keys kt ≤ qt, calculating the dot product between Q[qt] and K[kt] using a loop over HS.

2. Softmax: Track maxval across the scores for stability, subtract before exponentiation, and normalize.

3. Weighted Sum: Use the normalized softmax scores to compute a weighted sum over the values V[kt].

- Storage Strategy: A per-thread local array weights[MAX_T] is used to store intermediate attention scores. Once computed, the results are written to a scratch buffer (inp), which is later unpermuted using the unpermute_kernel to match the expected output layout.

- Memory Optimization: Instead of allocating a separate buffer for intermediate results, the original input buffer inp is reused to minimize memory usage.

Performance Impact

Since the attention matrix is never fully constructed, this avoids a massive O(T^2) buffer for each head and batch. On smaller batch sizes, however, the performance gains are modest — and in some cases negligible — due to:

- The per-thread inner loop over T, which introduces warp divergence and reduces occupancy.

- Lack of shared memory reuse or tiling optimizations.

- Softmax, dot-product, and value-weighted sum all happening serially within one thread.

- Profiling results using Nsight Compute show that while global memory transactions decrease, register and local memory pressure increases due to the large per-thread weights array. Warp stalling is common, as each thread independently computes and accumulates over many timesteps.

Potential Improvements

Move T Iteration to Blocks

Currently, each thread handles a full token, iterating over all kt <= qt, resulting in nested loops, uneven workload, and warp divergence.

We need to restructure the kernel so that each thread works on a specific (b, nh, qt, hs)

By reconstructing the kernel, we can achieve better wrapping uniformity and parallelism, significantly lower the time taken to grind the nested loops, eliminating the warp stalling problem that we encountered in matmul and weighted sum.

Need to take care about the softmax portion of the kernel: can probably use one thread per T to calculate softmax

Use Shared Memory for Q, K, V, and Weight Arrays

Since k, q, v are accessed multiple times within loops, it might be a good idea to preload them into shared memory, reducing needs to access global memory

Weight array is stored in global memory. Similarly, this per thread array is also read/wrote multiple times during the execution of the kernel, so storing it in shared memory can be beneficial. This is help if we're going to encompass T over block.

5.2 Multi-GPU Optimization

Motivation

As the input size grows, GPU kernels tend to accumulate more memory and computational needs, which scales proportionally with the input. To mitigate these issues, we need to distribute the workload onto multiple GPUs. One of the possible way to do it is by "slicing the batch". In GPT-2 inference, the batch dimension (B) is independent across samples—meaning each input in the batch can be processed in isolation. By partitioning the batch evenly across available GPUs, each device handles a smaller subset of the input while still performing the full forward pass for its assigned portion. This allows for parallel kernel launches, better GPU utilization, and reduced per-GPU memory footprint, all without requiring synchronization or communication between devices during inference.

Implementation Details

To implement multi-GPU support, we added a new wrapper around each kernel launcher (e.g., launch_attention_forward, launch_layernorm_forward, etc.). These wrappers first determine the number of available GPUs using cudaGetDeviceCount, and divide the total batch size B into disjoint slices for each GPU. This slicing is calculated as:

```
int B_local = (B + num_gpus - 1) / num_gpus;
```

Each GPU receives a segment of the input tensor, corresponding scratch buffers, and output storage. The kernel logic remains unchanged, but each GPU processes only its assigned batch slice.

Take attention as an example. After allocating and transferring input tensors (inp, qkvr, out) on each GPU using cudaMemcpyAsync, we launch the launch_attention_forward() kernel on each device stream in parallel. The kernel flow is the same:

- Inputs are permuted into Q, K, and V using a permute_kernel.
- flash_attention_forward_kernel computes the softmax-scaled attention scores and the weighted sum.
- Results are unpermuted back into the original output format using unpermute_kernel.

All parameter tensors are replicated across devices to avoid cross-GPU dependency. After execution, each output chunk is copied back to GPU 0 and stitched together into the final output tensor. Memory cleanup is performed via cudaFree, and streams are synchronized and destroyed to ensure correctness.

This approach was generalized to other modules like layernorm, matmul, residual, and GELU by applying the same pattern: divide the batch, copy slices to each GPU, execute kernels independently, and gather outputs.

Notable Metrics

- All benchmarks were collected on 4 GPU configurations.
- Many multi-GPU kernels individually run at approximately ⅓–¼ the time of their single-GPU counterparts.
- However, due to large overheads from cudaMalloc and cudaMemcpyAsync, the total runtime does not always improve. In some cases, it even worsens.

Here are a few kernel-specific examples:

- **Layernorm**: 660μs (naive) → 638μs (multi-GPU) — small gain due to small input and high parallelism.

- **Flash Attention**: 576μs → 1.267ms — runtime nearly doubles.
- **Residual Add**: 4.1μs → 463μs — memory dominates
- **cuBLAS Matmul**: 1.788ms → 21.233ms — same issue as above

What Went Wrong?

- The biggest limitation was memory overhead. Every kernel requires memory allocation and transfer of inputs/outputs (and sometimes weights and biases) to every GPU. For larger kernels like matmul, these tensors can reach megabytes in size, and copying them to four GPUs becomes a bottleneck. Repeated cudaMalloc and cudaMemcpyAsync calls increase latency and stall execution.
- In some cases like residual_add, the actual computation takes just a few microseconds, yet memory overhead adds hundreds of microseconds, effectively negating the benefits of parallelism.
- Additionally, not all kernels benefit equally from batch slicing. Flash attention, for example, didn't show significant improvement in runtime even when the input was split across four GPUs. This is because each thread already works over the entire sequence dimension T, and the runtime is dominated by loops over Tand HS— not batch. So cutting the batch size doesn't reduce per-thread workload or memory pressure in a meaningful way.

Potential Improvements

Persistent Memory Allocation

Avoid cudaMalloc and cudaMemcpy inside every forward call by allocating reusable memory during model initialization (e.g., in gpt2.cuh). Reusing buffers would dramatically reduce launch overhead and improve throughput.

Target Only Compute-Heavy Kernels

Use multi-GPU parallelism only where it matters — on slow, compute-intensive kernels. Light kernels like GELU or layernorm are fast enough already, and parallelism only adds noise. For heavyweights like attention or matmul, this strategy shines.

Focus on Partition-Responsive Kernels
Some kernels scale poorly with batch partitioning due to tight inner loops or poor memory access patterns. Profiling should be used to identify these cases. Avoid complicating non-scaling kernels like flash attention with multi-GPU logic unless major rewrites (e.g., split T over blocks) are planned.

5.2 KV-Cache

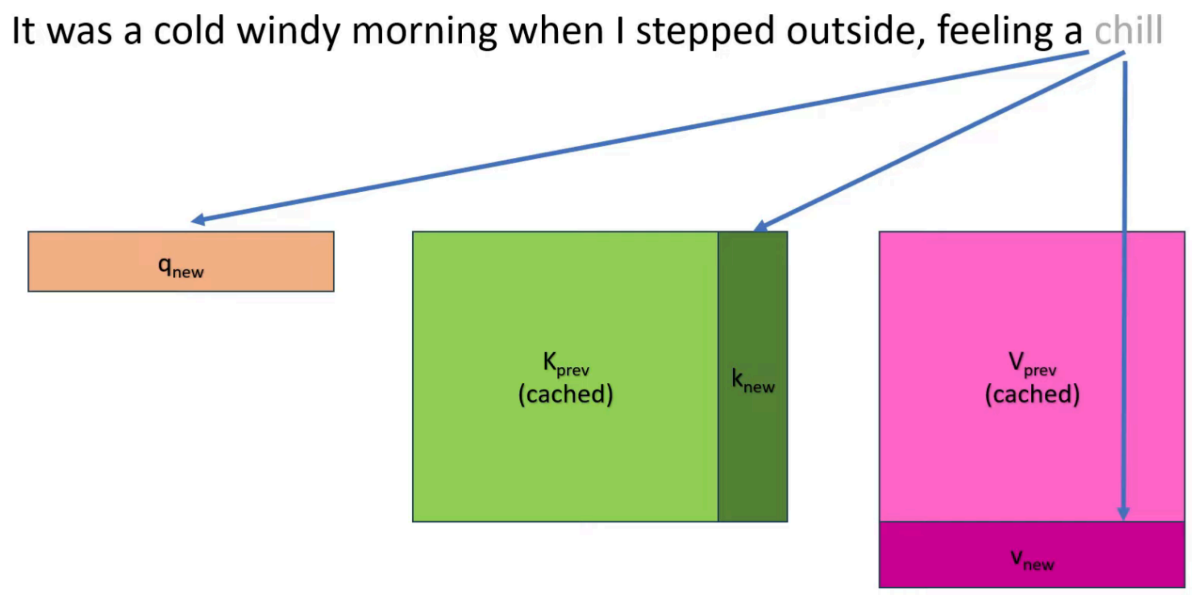
Motivation

This optimization sits at the core of the Transformer model, and is one of the most significant optimizations you can add for a generative LLM. As you'll see, naively reusing the forward pass of a transformer model for inference leaves a lot of performance improvements on the table. KV caching is targeted towards optimizing the inference pass by reducing unnecessary calculations and reusing data with a managed cache.

In order to compute the Attention matrix, the model computes three matrices: Q, K, and V. These are calculated by multiplying the input vector with three weight matrices for Q, K and V. The K-V Cache focuses on optimizing the computation of the K and V matrices by caching and reusing the K and V matrices.

Background

Generative decoder-only models like GPT-2 take a sequence of tokens as input to generate one output token. Then, it repeats the computation while adding the previously generated token into its input sequence. This is called autoregressive decoding. For every new token that's generated, it must recompute the K and V matrices entirely. However, the K and V matrices don't change that much over time. In fact, every token only adds one column and one row to the K and V matrices respectively.


It was a cold windy morning when I stepped outside, feeling a chill
q_{new}
K_{prev} (cached)
k_{new}
V_{prev} (cached)
v_{new}

The figure above shows the idea behind KV caching. For each token ("chill" in this case), you compute a Q, K, and V matrix. With KV caching however, notice that we don't need a query for every single token–we only need the query for the new token, "chill". The rest of Q isn't even cached, because it's actually not needed. So, Q reduces to a single vector.

Now you might be wondering,
"Why do we only compute a single query vector? Don't we need the full Q matrix for Attention?"

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

During the training process, yes. In a full parallel pass (like training or encoding an entire sentence), every token is "active" simultaneously, so all need their Qs. Each token is trying to understand the sequence from its own standpoint, so each token needs its own unique Q vector to direct its "attention inquiries."

During inference, however, we only predict a single word. The focus is on the very last token in the sequence generated so far, because we use its output to predict the *next* token. We don't do anything with the output of all the other queries. The queries of past tokens are not needed because their role was to help form their own output representations at the time they were processed, and those representations have already been used to influence the current state (e.g., by being part of what the current token's Q attends to via their K/V vectors).

Performance Impact

With KV caching, the only new information we need to compute is a single Q vector, a single K vector, and a single V vector. This dramatically speeds up the computation for each Attention block, especially when dealing with larger context windows.

For example, when computing the 100th token, it'll take roughly the same amount of time to compute the K and V vectors as it did for the first token. Without a K-V cache, computing the 100th token's K and V vectors will be 100 times more expensive, as it has to perform the K and V computation 100 times.

More importantly, the K-V cache also dramatically speeds up the Attention score computation itself. The shape of Q changes from a 2D matrix to a 1D vector, and the computational complexity for calculating Attention reduces from quadratic to linear (i.e. $O(n^2) \rightarrow O(n)$). This can result in massive performance savings.

The biggest tradeoff is increased memory usage. With K-V caching, we need to maintain a large cache in GPU memory in order to reuse it across multiple inference passes. However, memory is usually cheaper than compute, and the performance improvements largely outweigh the increase in memory usage.

Implementation Details

The exact details will depend on your specific implementation. There a few key parts to consider:

1. Managing your KV cache
2. Modifying the forward pass to handle single queries (once the cache is initialized)
3. Miscellaneous changes to handle single tokens vs full input (e.g. indexing, decoding the output token, etc.)

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

In our case, we used the following plan to help implement the KV cache:

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

If the cache is not initialized:

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

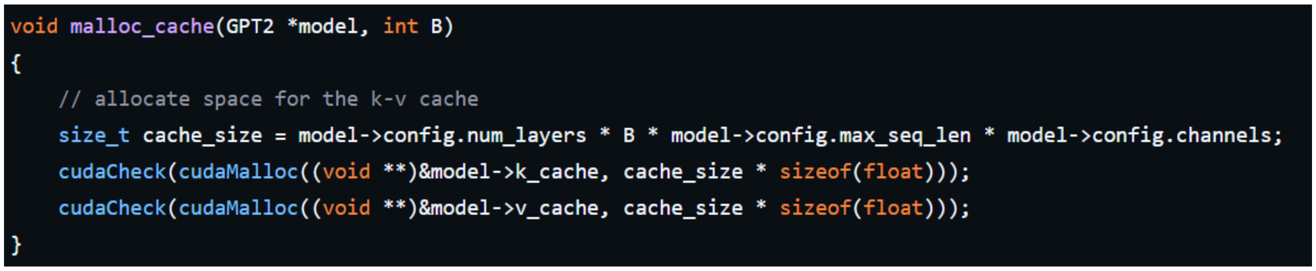
If the cache is initialized:

- Using, cache_len as an offset, encode only the last token
- Instead of computing the full Q, K, V matrices, only compute the newly generated token
- Split Q, K, and V
- Append K and V to the cache and increment cache_len
- Pass in Q, K_cache, and V_cache to the modified attention kernel.
- The attention kernel is modified to use a single query vector instead of a matrix
- Pass the output through the rest of the forward pass. Make sure you're aware that the output is now a single vector instead of a matrix.

Code Snippets

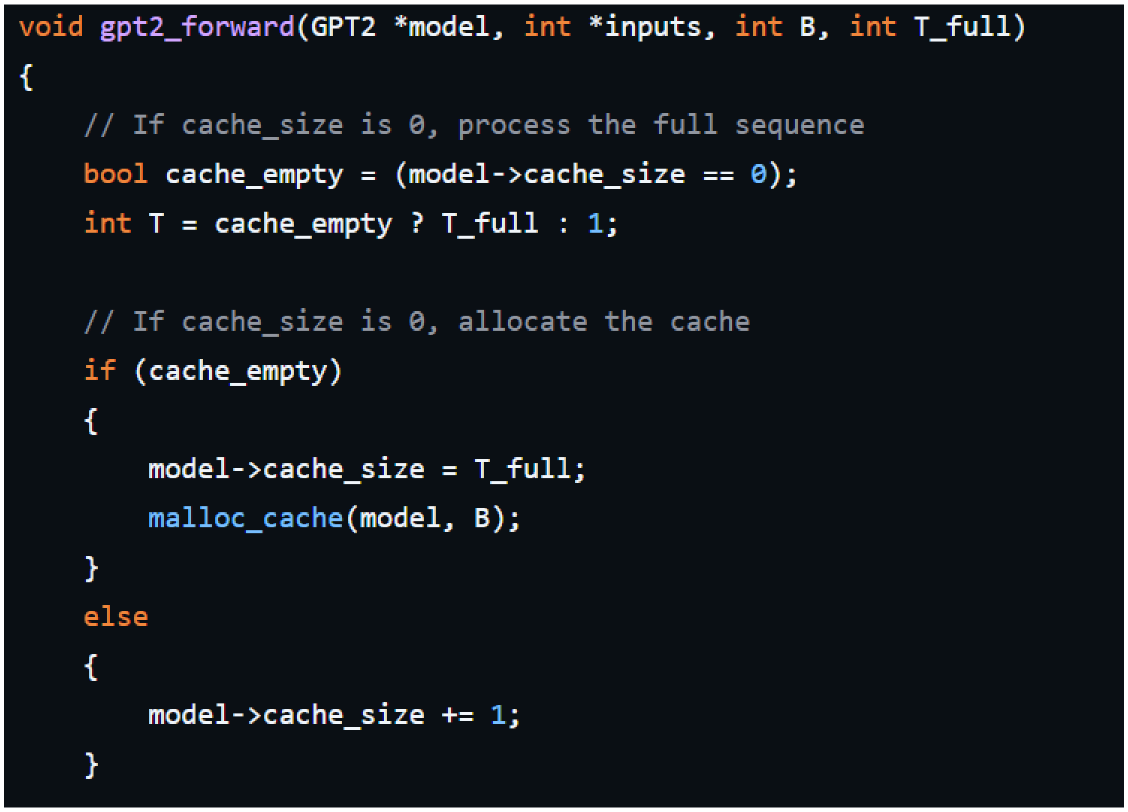
Here are some code snippets that might give you a feel for some of the things you'll need to watch out for:

Allocating the cache



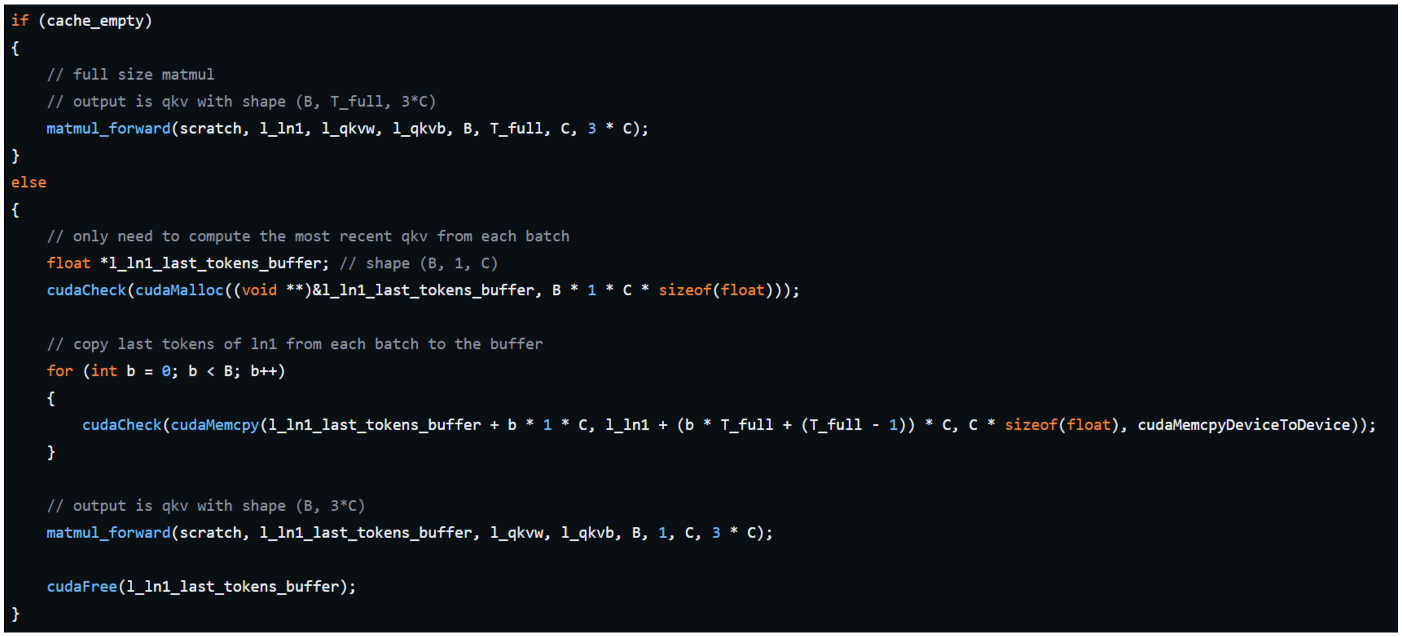
```
void malloc_cache(GPT2 *model, int B)
{
    // allocate space for the k-v cache
    size_t cache_size = model->config.num_layers * B * model->config.max_seq_len * model->config.channels;
    cudaCheck(cudaMalloc((void **)&model->k_cache, cache_size * sizeof(float)));
    cudaCheck(cudaMalloc((void **)&model->v_cache, cache_size * sizeof(float)));
}
```

Handling single-token vs full length



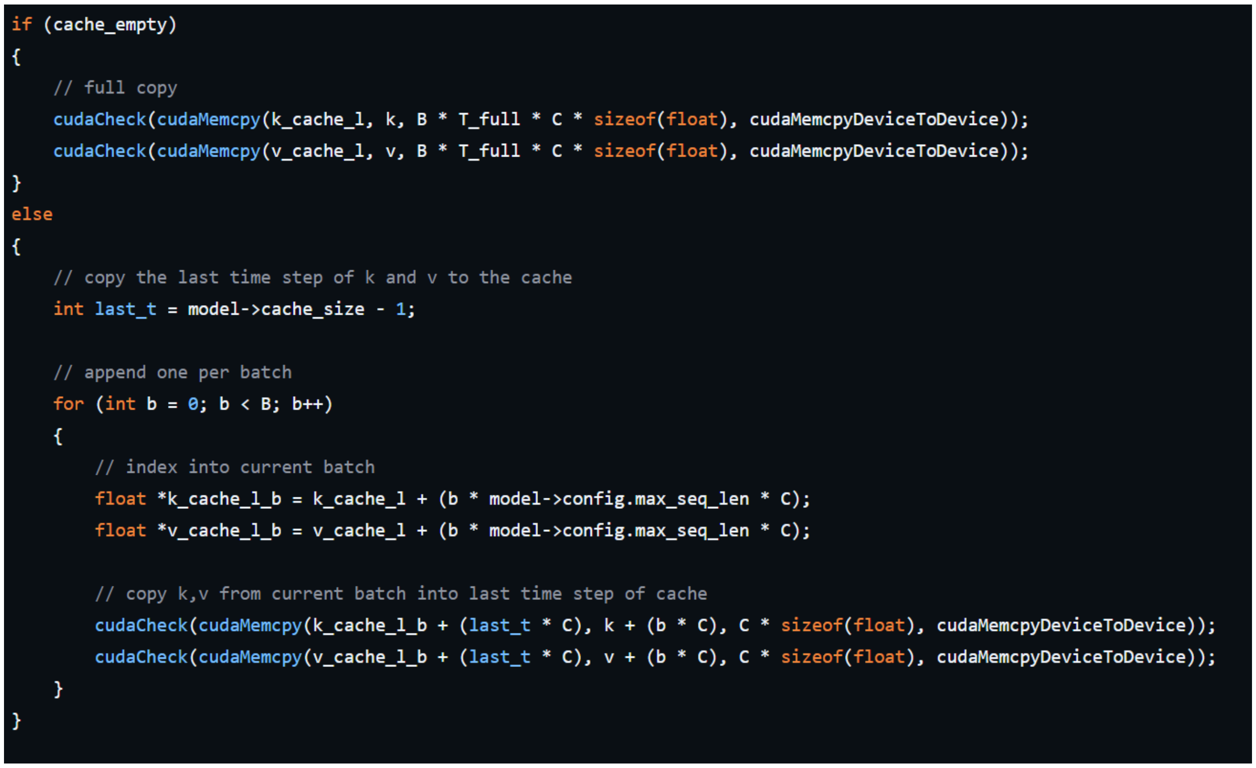
```
void gpt2_forward(GPT2 *model, int *inputs, int B, int T_full)
{
    // If cache_size is 0, process the full sequence
    bool cache_empty = (model->cache_size == 0);
    int T = cache_empty ? T_full : 1;

    // If cache_size is 0, allocate the cache
    if (cache_empty)
    {
        model->cache_size = T_full;
        malloc_cache(model, B);
    }
    else
    {
        model->cache_size += 1;
    }
```

Example of how you might handle single-token vs full length when computing Q,K,V



```
if (cache_empty)
{
    // full size matmul
    // output is qkv with shape (B, T_full, 3*C)
    matmul_forward(scratch, l_ln1, l_qkvw, l_qkvb, B, T_full, C, 3 * C);
}
else
{
    // only need to compute the most recent qkv from each batch
    float *l_ln1_last_tokens_buffer; // shape (B, 1, C)
    cudaCheck(cudaMalloc((void **)&l_ln1_last_tokens_buffer, B * 1 * C * sizeof(float)));

    // copy last tokens of ln1 from each batch to the buffer
    for (int b = 0; b < B; b++)
    {
        cudaCheck(cudaMemcpy(l_ln1_last_tokens_buffer + b * 1 * C, l_ln1 + (b * T_full + (T_full - 1)) * C, C * sizeof(float), cudaMemcpyDeviceToDevice));
    }

    // output is qkv with shape (B, 3*C)
    matmul_forward(scratch, l_ln1_last_tokens_buffer, l_qkvw, l_qkvb, B, 1, C, 3 * C);

    cudaFree(l_ln1_last_tokens_buffer);
}
```

Managing the cache and copying K and V



```
if (cache_empty)
{
    // full copy
    cudaCheck(cudaMemcpy(k_cache_l, k, B * T_full * C * sizeof(float), cudaMemcpyDeviceToDevice));
    cudaCheck(cudaMemcpy(v_cache_l, v, B * T_full * C * sizeof(float), cudaMemcpyDeviceToDevice));
}
else
{
    // copy the last time step of k and v to the cache
    int last_t = model->cache_size - 1;

    // append one per batch
    for (int b = 0; b < B; b++)
    {
        // index into current batch
        float *k_cache_l_b = k_cache_l + (b * model->config.max_seq_len * C);
        float *v_cache_l_b = v_cache_l + (b * model->config.max_seq_len * C);

        // copy k,v from current batch into last time step of cache
        cudaCheck(cudaMemcpy(k_cache_l_b + (last_t * C), k + (b * C), C * sizeof(float), cudaMemcpyDeviceToDevice));
        cudaCheck(cudaMemcpy(v_cache_l_b + (last_t * C), v + (b * C), C * sizeof(float), cudaMemcpyDeviceToDevice));
    }
}
```

6. Discussion

All in all, many of these optimization techniques were useful in accelerating the speed that our GPT-2 model behaved. Furthermore, it would be interesting to see how some of these techniques could be combined to yield further optimizations, and what combination of these optimizations would be useful in trying to be faster than the cuBLAS library and some of its functions.