Parallel Reduction

Minseob Shin

The reduction operator is a type of operator that is commonly used in parallel programming to reduce the elements of an array into a single result and we will use this optimization technique to speed up a couple of our kernels. Observing our kernels, we were able to identify that layernorm_forward_kernel and softmax_forward_kernel compute a sum of values. Computing a sum is one of the classic optimizations achieved by reduction operation, as it reduces a set of input values to one value (in our case, this will be the sum). Note that this is possible because computing the sum is commutative and, importantly, associative.

There are two reasons for the reduction optimization

1) The parallel reduction operator reduces the steptime from O(N) to O(log N) by organizing the computation as a binary tree structure rather than a sequential scan.
2) The parallel reduction algorithm is work-efficient, meaning that it requires the same amount of work as a sequential algorithm.

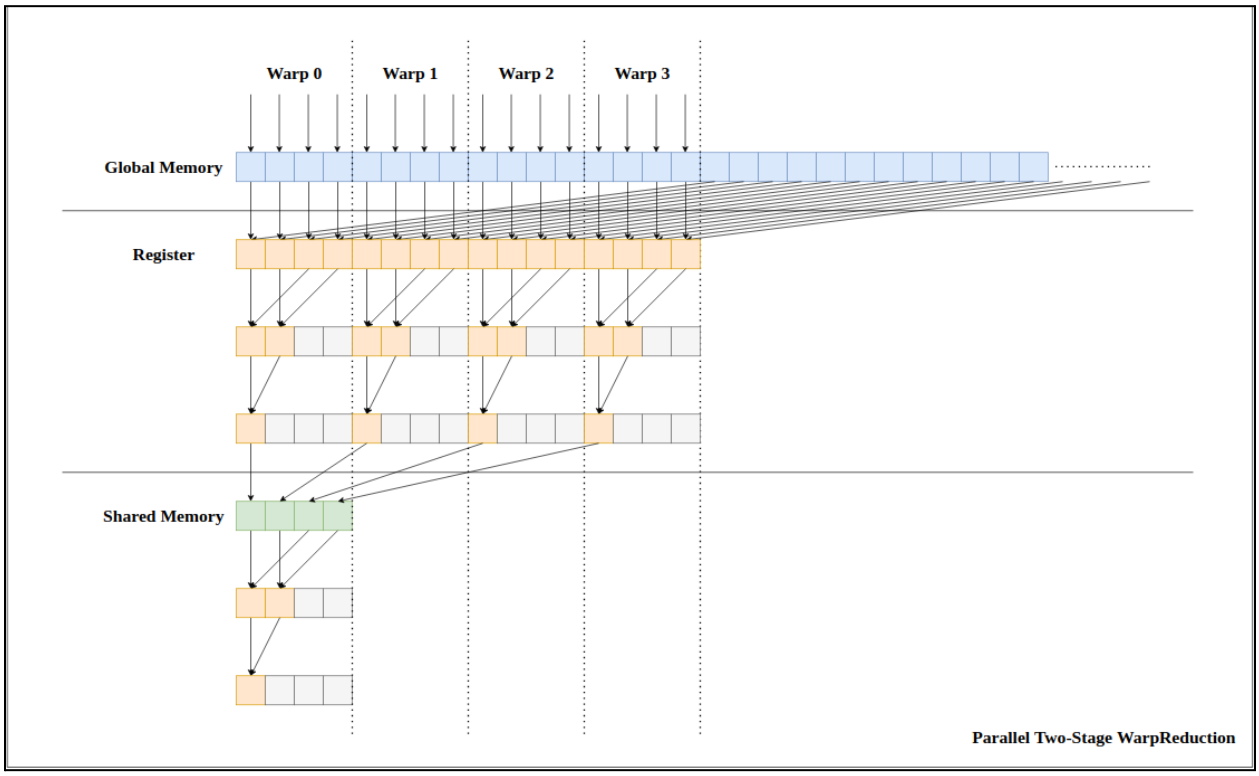

Figure 1. Two-Stage Warp Reduction Algorithm

For N input values, the number of operations is

$$\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \ldots + 1 = N - 1$$

Let’s take a look at our baseline implementation for both layernorm_forward and softmax_forward kernel.

Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
5,133,523	25	205,340.9	205,472.0	190,752	214,880	3,975.6	layernorm_forward_kernel
1,483,196	12	123,599.7	123,456.0	123,071	124,576	469.1	softmax_forward_kernel

Figure 2. Baseline Implementation Profile

Our baseline implementation follows naive implementation where each thread individually computes the sum of the input columns. Layernorm_forward_kernel lets each thread loop through the input column values three times. Each for computing the mean, variance, and reciprocal standard deviation. Similarly, softmax_forward_kernel consists of three major loops that only compute through their own position. This is due to the decoder-only architecture of GPT-2. This results in a total runtime of 5,133,523 ns for layernorm_forward_kernel and 1,483,196 ns for softmax_forward_kernel.

```
float sum = 0.0f;
for (int c = 0; c < C; ++c) {
    sum += inp[b * T * C + t * C + c];
}
```

Figure 3. Naive Summation

Our optimized implementation follows Figure 1. Our implementation of parallel reduction is the two-stage warp[1] reduction. It first loads the sum of two (or more) input values from global memory to registers. Once values are loaded to registers, each warp performs warp level reduction using warp shuffle functions[2] to broadcast the intermediate values from one thread to another. All warps store their computed value to shared memory and synchronization. Finally, one warp performs warp reduction to acquire the single result. This implementation of parallel reduction is highly efficient because it utilizes registers and removes unnecessary overhead of thread block synchronization. Note that fully computed intermediate values needed for the next set of computation were loaded into shared memory by thread 0 to broadcast them across the thread within the block. Also, we chose to implement thread coarsening[3] along with parallel reduction due to the possibility of non-coarsened thread blocks being serialized by the hardware, as shown in Figure 5. After applying the parallel reduction optimization, we were able to obtain 36.5x speed up in layernorm_forward_kernel and 11.4x speed up in softmax_forward_kernel.

Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
140,578	12	11,714.8	11,712.0	11,648	11,872	64.6	softmax_forward_kernel
129,055	25	5,162.2	5,023.0	4,672	5,888	438.2	layernorm_forward_kernel

Figure 4. Optimized Implementation Profile

[1] A warp is a group of 32 threads that are scheduled and executed together as a single unit on an NVIDIA GPU
[2] A warp shuffle function in CUDA (and GPU programming) is a special operation that allows threads within the same *warp* to directly exchange register values with each other
[3] Thread coarsening is a CUDA optimization technique where each thread computes multiple outputs rather than a single output

```
for (int tile = tx; tile <= own_pos; tile += BLOCK_DIM)
{
    maxval = fmaxf(maxval, inp[base + tile]);
}
```

Figure 5. Parallel Reduction Max Computation

In Nsight Compute, we observed the number of FLOPs (Floating-point operations) to verify that parallel reduction is indeed work-efficient. Let's look at baseline and optimized profiles for instance.

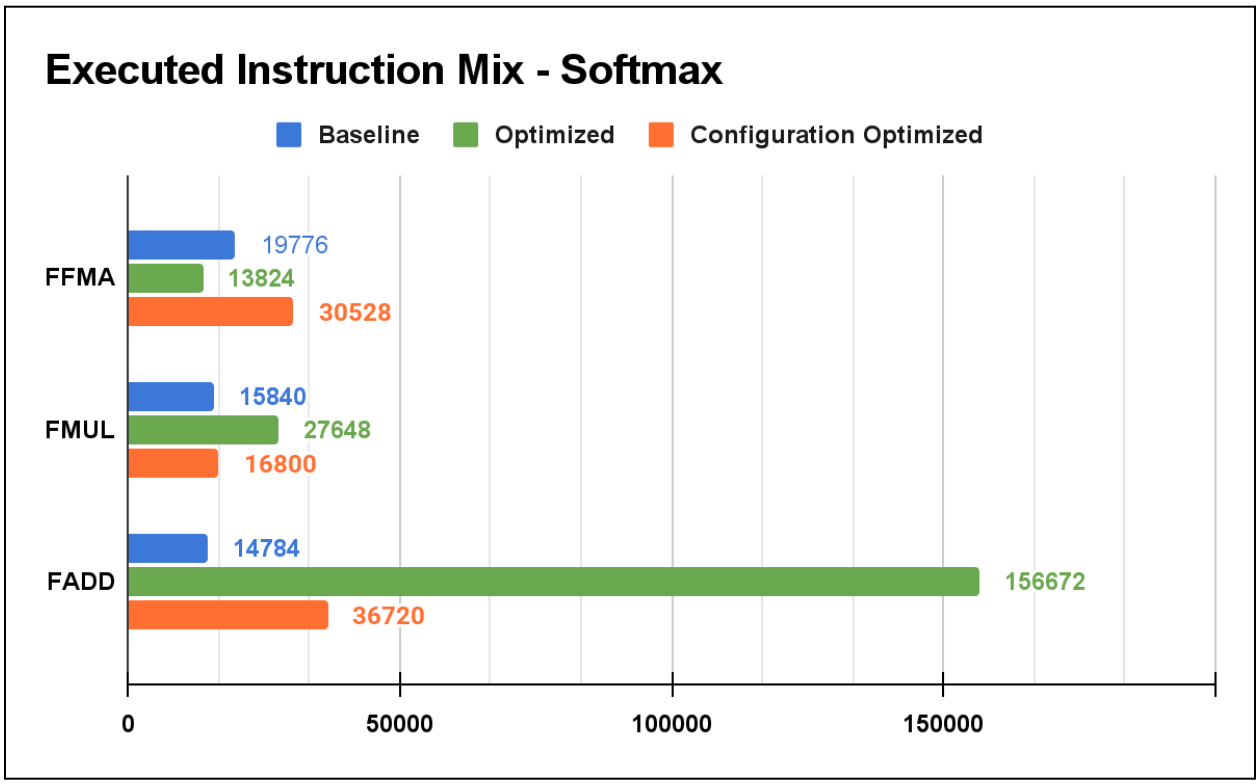

Figure 6. Softmax Floating-Point Instruction Statistics

We could observe that the FADD significantly increased from 24,197 to 156,672, increasing over 6x times. We believed that this increase was not reasonable, and concluded that the reduction was not as parallel as we wanted it to be due to the large number of inputs and limited device resources. In order to compensate for this, we changed the kernel launch configuration. For the configuration optimization, we changed the BLOCK_DIM from 256 to 32 to increase the coarse factor by 8 and removed the overhead from two-stage warp reductions. This results in a total number of FADD of 36,720, which is a reasonable increase from the baseline implementation as parallel reduction uses slightly more FADD to reduce the computed values from each block to the final value.

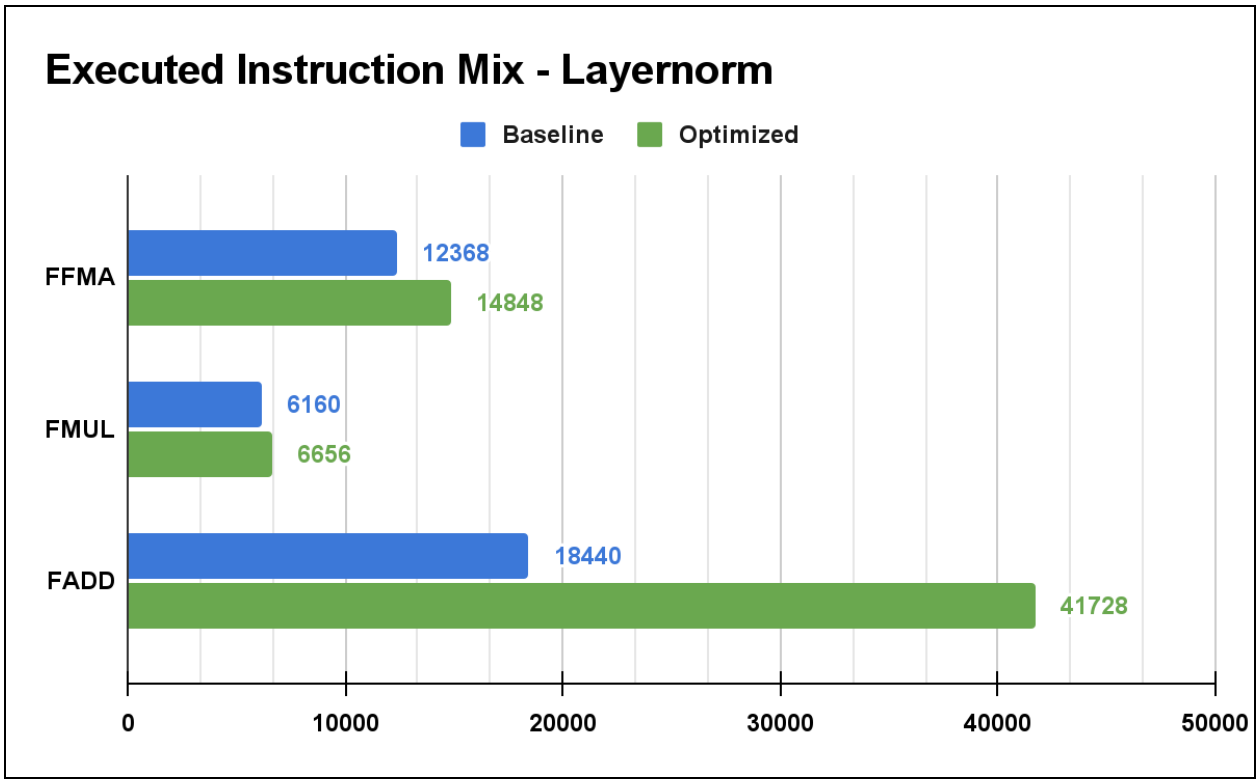

Figure 7. Layernorm Floating-Point Instruction Statistics

For the same reason, we could observe a similar increase in FADD for layernorm_forward_kernel, as it increases from 18,440 to 41,728. Note that we didn't include Configuration Optimized statistics because our optimized configuration turned out to be the best configuration for layernorm_forward_kernel. To conclude, this demonstrates that the parallel reduction algorithm is work-efficient and can result in a significant amount of speed up for specific use cases.