**ECE 408 Project Report**

**Team Name:** fast\_af

**Names:** Omar Taha, Jason Gong, Shreyas Byndoor

**NetIDs:** otaha2, xg8, byndoor2

**Rai IDs:** 5c78c4b084318364bab96f01,

**Affiliation**: Chicago City Scholars

**Milestone 1**

Kernels that collectively consumed more than 90% of the program time:

* [CUDA memcpy HtoD]
* cudnn::detail::implicit\_convolve\_sgemm
* volta\_cgemm\_64x32\_tn
* op\_generic\_tensor\_kernel
* fft2d\_c2r\_32x32
* volta\_sgemm\_128x128\_tn
* cudnn::detail::pooling\_fw\_4d\_kernel
* fft2d\_r2c\_32x32

CUDA API calls that collectively consume more than 90% of the program time:

* cudaStreamCreateWithFlags
* cudaMemGetInfo
* cudaFree

Kernels vs. API Calls:

CUDA Kernels are simply defined as regular C functions. However, unlike typical C functions, CUDA Kernels are executed N times in parallel by N different CUDA threads. Meanwhile, CUDA APIs provide C functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc. They are not executed by each CUDA thread as a Kernel is.

Output of rai running MXNet on the CPU:

Loading fashion-mnist data... done

Loading model... done

New Inference

EvalMetric: {'accuracy': 0.8236}

9.30user 3.46system 0:05.34elapsed 239%CPU (0avgtext+0avgdata 2471560maxresident)k

0inputs+2824outputs (0major+666761minor)pagefaults 0swaps

Program run time:

0:05.34 elapsed

Output of rai running MXNet on the GPU:

Loading fashion-mnist data... done

Loading model... done

New Inference

EvalMetric: {'accuracy': 0.8236}

4.41user 3.28system 0:04.34elapsed 177%CPU (0avgtext+0avgdata 2837968maxresident)k

0inputs+4552outputs (0major+661333minor)pagefaults 0swaps

Program run time:

0:04.34 elapsed

**Milestone 2**

program execution time:

0:15.52 elapsed

Op Times:

1) Op Time: 2.826305

2) Op Time: 11.143122

**Milestone 3**

Correctness & Timing:

|  |  |  |  |
| --- | --- | --- | --- |
|  | DataSet 100 | DataSet 1000 | DataSet 10000 |
| Op Time #1 | 0.000079 | 0.000558 | 0.005636 |
| Op Time #2 | 0.000216 | 0.001974 | 0.021531 |
| Correctness | 0.84 | 0.852 | 0.8397 |

nvprof profile:

A screenshot of a social media post

Description automatically generated

**Milestone 4**

\*All Kernel Optimization Code can be found in their respective src code file

“new-forwardOp#<Optimization Number>.cuh”\*

Base Profiling:

**A screenshot of a social media post

Description automatically generated**

**A picture containing screenshot

Description automatically generated**

A picture containing umbrella, accessory

Description automatically generated

Optimization #1: Weight Matrix in Constant Memory

The Weight Matrix is a set matrix that is constantly being reused by the kernel to perform the convolution operation. By copying the weight matrix into constant memory before kernel execution, we are drastically decreasing the number of global memory reads. The kernel then reads from constant memory for the entire weight matrix.

This has resulted in a decrease in the Avg. Runtime Duration from ~5.26 ms to ~4.697 ms. This is because we are no longer doing many costly global memory reads for the weight matrix data. More importantly, from the graph below, we see that the Memory Operations utilization decreased compared to the Base Kernel Memory Operation (see above). This is good in that we are decreasing the amount of time and resources that are spent on memory operations (read/writes).

A picture containing screenshot

Description automatically generated

Optimization #2: Input Data in Shared Memory

Input Data is constantly being used by the kernel, and different threads are reusing input data. If we are not careful about the use of input data, we will have a large number of global memory reads that will cause a memory bottleneck causing the kernel to slow down due to memory latency. To address this, we will read input data once and store it into the devices shared memory. Then we are saving many global memory reads for every reuse of a input element.

We viewed that the Base Kernel had 53.22 Active Warps resulting in an 83.2% Occupancy Per SM. After the shared memory optimization, that number jumped to 61.78 Active Warps resulting in a 96.5% Occupancy Per SM. In addition to this, according to the chart below, we noticed that the Compute Utilization jumped up to ~85% from ~75%. This is due to both a jump in the Control-flow Operations as well as Arithmetic Operations. This tells us we must decrease the control divergence caused by this optimization. The L2 Cache usage was also increased to 454.5 GB/s from 276.5 GB/s; This means we are utilizing shared memory much more in the kernel

\*This analysis was done on the forward:1 Kernel but was also compared to the forward:1 Kernel of the Base Naïve implementation

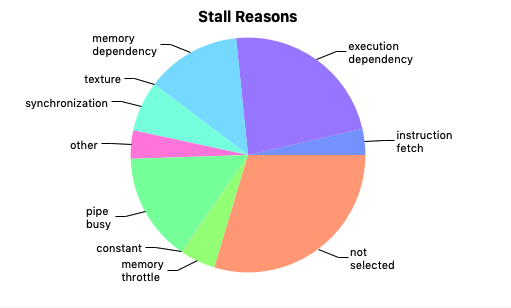
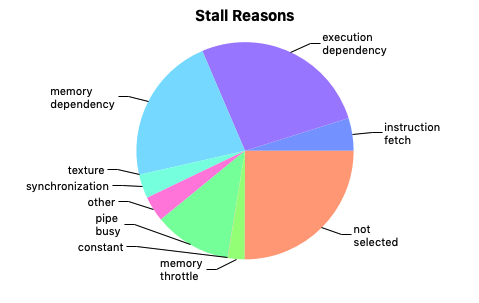
A picture containing screenshot

Description automatically generated

Optimization #3: Double Buffering

To ensure that all shared data is up to date and has not change, we must have 2 syncthreads() API calls. The first is to ensure data is loaded, and the second is to ensure the data has been consumed before rewriting the data in the next iteration. In an effort to get around this, we utilized Double Buffering. The idea was to have 2 sections of shared memory in which we switch the pointers each iteration. This way, we eliminate the need for the second syncthreads() API call.

We found that this optimization greatly reduced the stall due to synchronization. The two charts below show the effect before and after Double Buffering. The left shows before, and the right shows after. As we can see, the synchronization stall was approximately halved. However, although the synchronization stall was decreased, the operational run time of the kernel did not seem to be affected.



**Final Submission**

Optimization #4: Unroll + Shared-Memory Matrix Multiply

We implemented the Data Unrolling & Shared-Memory Matrix Multiply. When doing Data unrolling, we are trying to optimize the compute efficiency in turning the convolution into a matrix multiply. We can optimize a matrix multiply far better with advanced algorithms than optimizing the convolution. Below is the compute analysis of the Matrix Multiply. We see that relatively, the amount of computation being done is being controlled the memory operations and arithmetic operations. The memory operations is due to reading all of the data into shared memory. We would like to increase the amount of computation being done. The utilization is low because the size of the data being operated on in each matrix multiply kernel invocation is very small. This can be something we try to optimize later on.

We saw that the total Operation Time increased roughly by a factor of 4 to 0.20 seconds from 0.05 seconds. We hypothesis this is because of the fact that we first need to unroll data as well as do a matrix multiply on the data, which could be causing the increase in time.

A picture containing screenshot

Description automatically generated

Optimization #5: Tuning with Restrict & Loop Unrolling

A common problem that is encountered when using pointer data is Pointer Aliasing. Pointer Aliaing occurs when pointers overlaps. When a compiler can’t determine if two pointers alias, it assumes that they do. Take a look at this function for example.

void example1(float \*a, float \*b, float \*c, int i) {

a[i] = a[i] + c[i];

b[i] = b[i] + c[i];

}

\*Function above taken from NVIDIA CUDA Article

Here, the compiler will reload the c pointer before the 2nd line in the case that c was modified. Thus extra code is being generated and run throughout the function. In order to get around this, we add the \_\_restrict\_\_ keyword to the pointers. We are then promising that there is no pointer aliasing and thus getting rid of the extra code being run.

To increase Instruction-level parallelism (ILP) we utilized the benefit of Loop Unrolling. We do this by adding #pragma unroll before all 3 of our loops in our base Kernel.

We see the effects of this combined optimization in the decreased runtime. The runtime of the kernel dropped to 5.0372ms from 5.2596ms. Also looking in the graph below and comparing with the base Kernel graph above, we see that the total compute utilization increased to 60% from ~52%. This means that by unrolling, we are increasing the amount of computation being done by the kernel.

A picture containing screenshot

Description automatically generated

Optimization #6: Sweeping Parameters

Finding the right parameters to launch kernels with is key. Having an optimized set of parameters will allow you to fully take advantage of the hardware and its capabilities. Below is our findings in sweeping the launch parameters for TILE\_WIDTH = 8, 16, and 32, respectively.

Note #1: This is being built off of our shared memory convolution kernel.

Note #2: Op Times listed below is of forward:1 kernel pass

TILE\_WIDTH = 8

Op Time = 9.315ms

A picture containing screenshot

Description automatically generated

TILE\_WIDTH = 16

Op Time = 5.9357ms

A picture containing screenshot

Description automatically generated

TILE\_WIDTH = 32

Op Time = 14.85ms

A picture containing screenshot

Description automatically generated

We can see from the stats and analytics above that TILE\_WIDTH = 16 is the best possible option. It has a much higher compute utilization as well as a significantly lower operation time. Thus we decided to use TILE\_WIDTH=16 for our kernel runs.