CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

Milestone 1

How we divided up our works: We discussed our approach and implemented it together

[kernel calls that collectively consume more than 90% of the program time]:

Time(%)	Time	Calls	Avg	Min	Max	Name
34.07%	118.49ms	9	13.166ms	13.149ms	13.180ms	fermiPlusCgemmLDS 128_batched
27.01%	93.927ms	1	93.927ms	93.927ms	93.927ms	cudnn::detail::implicit _convolve_sgemm
12.69%	44.128ms	9	4.9031ms	2.6906ms	6.2766ms	fft2d_c2r_32x32
8.2%	28.514ms	1	28.514ms	28.514ms	28.514ms	sgemm_sm35_ldg_tn _128x8x256x16x32
6.42%	22.331ms	14	1.5951ms	1.5360us	21.504ms	[CUDA memcpy HtoD]
4.07%	14.156ms	2	7.0781ms	252.38us	13.904ms	cudnn::detail::activati on_fw_4d_kernel

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

Time(%)	Time	Calls	Avg	Min	Max	Name
43.61%	1.93912s	18	107.73ms	15.637us	969.21ms	cudaStreamCreateWi thFlags
27.11%	1.20548s	10	120.55ms	1.2190us	342.20ms	cudaFree
20.62%	917.01ms	27	33.963ms	236.99us	908.89ms	cudaMemGetInfo

API calls and kernel calls are both parts of the CUDA programming interface. Kernels are functions defined by the __global__ specifier which we are used to using in our homeworks. API calls are executed on the host, but have access to the global memory of the device. Kernel calls are executed on the device itself. The runtime API consists of cudaMemcpyTosymbol, cudaMalloc, and other functions used to access global variables. Generally the main tradeoff between the two is that kernel launches are more complex to implement but provide more fine-grained control while the runtime API makes device code management easier and cleaner.

There’s no significant performance difference between API and kernel calls, based on the statistics in the tables above.

[Output of rai running MXNet on the CPU]:

```
✱ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
EvalMetric: {'accuracy': 0.8444}
```

[m1.1.1.py program run time]:

```
13.26user 11.98system 0:11.66elapsed 216%CPU (0avgtext+0avgdata
2821748maxresident)k
0inputs+2624outputs (0major+38226minor)pagefaults 0swaps
```

[Output of rai running MXNet on the GPU]:

```
✱ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data...
done
Loading model...
[23:46:06] src/operator/./.cudnn_algoreg-inl.h:112: Running performance tests to find the best
convolution algorithm, this can take a while... (setting env variable
MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
done
New Inference
EvalMetric: {'accuracy': 0.8444}
```

[m1.2.py program run time]:

```
2.29user 1.06system 0:02.82elapsed 118%CPU (0avgtext+0avgdata 1138624maxresident)k
0inputs+3136outputs (0major+153677minor)pagefaults 0swaps
```

CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

Milestone 2

How we divided up our works: We discussed our approach and implemented it together

10,000 images

[m2.1.py program output]:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 7.463296
Op Time: 25.670930
Correctness: 0.8451 Model: ece408
```

[m2.1.py program runtime]:

```
37.64user 1.29system 0:36.93elapsed 105%CPU (0avgtext+0avgdata 2813404maxresident)k
0inputs+0outputs (0major+35786minor)pagefaults 0swaps
```

100 images

[m2.1.py program output]:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.074735
Op Time: 0.255830
Correctness: 0.88 Model: ece408
```

[m2.1.py program runtime]:

```
1.10user 0.58system 0:01.09elapsed 154%CPU (0avgtext+0avgdata 175624maxresident)k
0inputs+0outputs (0major+31500minor)pagefaults 0swaps
```

CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

10 images

[m2.1.py program output]:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.007455
Op Time: 0.025613
Correctness: 1.0 Model: ece408
```

[m2.1.py program runtime]:

```
0.76user 0.40system 0:00.72elapsed 160%CPU (0avgtext+0avgdata 169980maxresident)k
0inputs+0outputs (0major+29052minor)pagefaults 0swaps
```

Milestone 3

How we divided up our works: We discussed our approach and implemented it together

10,000 images

[m3.1.py program output]:

```
✱ Running python m3.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.203084
Op Time: 0.508383
Correctness: 0.8451 Model: ece408
```

CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

100 images

[m3.1.py 100 program output]:

```
✱ Running python m3.1.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.001499
Op Time: 0.004704
Correctness: 0.88 Model: ece408
```

10 images

[m3.1.py 10 program output]:

```
✱ Running python m3.1.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000259
Op Time: 0.000765
Correctness: 1.0 Model: ece408
```

the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

Our approach to optimization

The sequential implementation involves nested loops (7 for-loops), we wanted to implement a basic parallel implementation to speed up the execution in a way that workload are evenly distributed among thread blocks. We first layout the output feature maps into a giant output feature map, with dimension of (B * H_out , M * W_out), where B is the number of batches and M is the number of output feature maps in each batch. Then, we distribution the tasks in the thread grid, such that each thread would process a output of a corresponding output feature map. The time complexity is reduced because the work is now evenly distributed, and each thread will have O(C*K*K) work, in contrast, it had O(B*M*H*W*C*K*K) amount of work for a single process originally.

Combined matrix for basic optimization

Output feature maps

Figure 1. Example of mapping between output feature maps to a matrix used for optimization in this milestone

Initial Performance Results

Unsurprisingly, the basic, parallel implementation has improved in time complexity compare to the sequential version. In milestone 2, the program ‘m2.1.py 100’ took about 0.074735 in the first layer and 0.255830 for the second layer convolution, while in milestone 3, with a simple, parallel implementation, operation time has decreased to 0.002174 and 0.004704, respectively (the execution details generated by nvvp are shown below).

Figure 2. Kernel execution summary

Figure 3. “Forward_kernel” individual kernel analysis

Figure 4. Memory bandwidth and Utilization

CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

Milestone 4

Note: For this milestone, optimization 1 and 3 implementation are independent to each other. Optimization 2 was developed based on Optimization 1

How we divided up our works: Larry Poon (Optimization 1), Gerald Kozel (Optimization 3), Waleed Ramahi (Optimization 2)

Optimization 1 - ***Matrix Unrolling***

The first optimization is matrix unrolling. The idea behind is the layout the input features, output features maps and weights in matrix representation so that the convolution layer can be easily parallelized. Here is an example to illustrate our optimization:

Figure 5. Matrix Unrolling Optimization Sample

As you can see from the graph, now, the mask has dimension of MxK*K*C, input features' dimension is K*K*CxH_out*W_out, and output features' dimension is MxH_out*W_out. We have used three independent kernels to accomplish this task. In the first kernel, we load the original mask into the unrolled mask, the second kernel is the load the input

features into the unrolled input features, the third kernel loads the output features into unrolled output features.
The following is the outputs of the programs run as required:

Performance Results

10000 images

[m4.1.py 10000 program output]:

```
✱ Running python m4.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.490018
Op Time: 0.742920
Correctness: 0.8451 Model: ece408
```

100 images

[m4.1.py 100 program output]:

```
✱ Running python m4.1.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.005656
Op Time: 0.009813
Correctness: 0.88 Model: ece408
```

10 images

[m4.1.py 10 program output]:

```
✱ Running python m4.1.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000926
Op Time: 0.001081
Correctness: 1.0 Model: ece408
```

Performance Analysis

Figure 6. Kernels Computation time ranking

Figure 7.GPU memory bandwidth utilization

Figure 8. Unroll_multiply kernel detail analysis

Figure 9. Additional analysis with nvvp

Matrix Unrolling - Analysis

As we can see from the execution time from the output and nvvp. The implementation caused a slowdown, which we expected. First of all, having three separate kernels for loading caused overhead of context switching, memory transfer between host and device. This can be improved by fusing all kernels into one, utilize concurrency in the copy engine in PCIe (as suggested in figure 8). Secondly, there is contention when access the original data structures, input/output features, as every thread block are reading/writing to the same memory location. This explains why the unroll_multiply kernel took the most computation time because there is high access rate of global memory in that kernel, it can be fixed by utilizing shared memory to resolve the contention.

Optimization 2 - ***Storing masks in constant memory***

Another potential optimization for a convolutional neural network is the placement of the convolution filters in the device's constant global memory, as opposed to its ordinary global memory. The justification behind this is that, for many cuda applications, the global memory bandwidth can become a major limiting factor on the performance of a program. Whenever there is data that needs to be loaded multiple times, the impact of global memory bandwidth becomes very significant. Constant global memory is different because, since it is unchanging, it is very well cached and thus can be loaded much faster. The convolutional filters of a convolution layer are constant, so they are a perfect candidate for this optimization.

Figure 10. Unrolled weight matrix in constant memory

Performance Results

10000 images

[m4.1.py 10000 program output]:

```
✱ Running python m4.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.490576
Op Time: 0.716588
Correctness: 0.8451 Model: ece408
```

100 images

[m4.1.py 100 program output]:

```
✱ Running python m4.1.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.005859
Op Time: 0.009528
Correctness: 0.88 Model: ece408
```

10 images

[m4.1.py 10 program output]:

```
✱ Running python m4.1.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000989
Op Time: 0.001117
Correctness: 1.0 Model: ece408
```

Storing masks in constant memory - Analysis

After implementing the constant memory optimization, we found that our program actually performed worse than it had prior. While this is a little bit counter intuitive, it is easy to explain. The fact is, in putting the weights into constant memory, we introduced additional steps. These steps were to copy the unrolled matrix from the device back to the host, and then from the host to the constant memory. This will obviously take some time. The performance benefits we gained from moving the weights into constant memory were not significant enough to offset the additional overhead introduced by it.

Figure 11. Kernel Optimization properties

However, using constant memory for the mask matrix does improve the performance in the unroll_multiply kernel, it has become not the bottleneck of the performance. The amounts of global memory reads and stores have significantly dropped (comparing with figure 8).

Figure 12. Constant memory bandwidth utilization

Optimization 3 - *Shared Memory Convolution*

How to run: to run this program, simply use the code in shared_memory_convolution.cuh or make the program run with shared_memory_convolution.cuh instead of new-forward.cuh

An approach to optimize the basic forward kernel is to use shared memory tiling for both the input features and the weight masks. In theory this should reduce global memory traffic in the kernel which is known to be much slower than shared or constant memory traffic. In order to do so we modified the existing kernel to create a single shared memory matrix with size of (TILE_WIDTH + K-1)^2 for the input matrix and K*K for the filter matrix. We pass this memory into the kernel as we do not know K at compile time. This allows us to create a single extern shared memory array where we choose two start points for both input and filters. Over all input channels (C) we load a tile from X into shared memory by indexing positions around the thread's position in the grid. Additionally we load the mask matrix in a similar way but we only load one weight per thread. Finally we do the actual convolution in the same way as the suboptimal parallel version by iterating over K by K and summing X*W.

Performance Results

10000 images

[m4.1.py 10000 program output]:

```
✱ Running python m4.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.124374
Op Time: 0.387863
Correctness: 0.8451 Model: ece408
```

100 images

[m4.1.py 100 program output]:

```
✱ Running python m4.1.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.001269
Op Time: 0.004807
Correctness: 0.88 Model: ece408
```

10 images

[m4.1.py 10 program output]:

```
✱ Running python m4.1.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000163
Op Time: 0.000521
Correctness: 1.0 Model: ece408
```

Shared Memory Convolution - Analysis

Thus far, shared memory convolution is our fastest implementation of the forward kernel. This optimization operates on the fact that shared memory is quicker to access than global memory. A way to calculate reuse is to find the number of global memory accesses replaced by shared memory accesses. We need to load (TILE_WIDTH + K-1)^2+K*K elements to shared memory. For our specific implementation we had TILE_WIDTH=16 and K= 50. Each output element needs to access K^2 elements. Therefore we have 16^2*50^2 global memory accesses converted into shared memory accesses. Theoretically, we should have a speed up of 640000/6725 = 95x but in reality it is much lower than that (~1.6667). An explanation for the difference is the addition of adding the weight matrix to shared memory within the kernel as well, taking up more time and additional loads to memory. NVprof gives us a better view of time spent doing what below:

Figure 13. Constant memory bandwidth utilization

Figure 13. Constant memory bandwidth utilization

Figure 14. Memory bandwidth utilization with shared memory

The kernel appears to have high amounts of divergence (see figure 13)which is possibly due to a missing _syncthreads() after one of the boundary checking if statements. This could explain some of the missing speed up. Here we can see a 1,032.669 GB/s bandwidth for our shared loads compared to the original nvprof for milestone 3 having 149 GB/s for global loads giving about a 7x speed up in loads.

✱ Running python m4.1.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000204
Op Time: 0.000487
Correctness: 1.0 Model: ece408

Final Optimizations

Note: The following optimizations are based on the implementation for Optimization 3 (Shared Memory Convolution), since it yielded the best performance among the implementations we worked on in the last milestone. The optimizations are done sequentially i.e. Optimizations 6&7 are built on top of Optimization 4&5

How we divided up our works: Larry Poon (Optimization 4-7), Gerald Kozel (Optimization 8), Waleed Ramahi (Optimization 9)

Optimization 4&5 - *Tuning with __restrict__ directives and loop unrolling*

For this optimization, we utilized the ***restrict*** keyboard. The keyboard is used in pointer declaration of intent made to the compiler. The keyword signals compiler that the target point is the only pointer that points to object it interacts with. Theoretically, using this keyword would give us some speedup as it limits the effect of pointer aliasing, overhead occurs when compiler is constrainting on program execution order. To achieve this, we simply appended the keyword __restrict__ before the pointer declarations in the function signature of forward_kernel() and forward(), as follows:

.

```
__global__ void forward_kernel(float * __restrict__ y, const float * __restrict__ x, const float * __restrict__ k, const int B, const int M, const int C, const int H, const int W, const int K)
```

Figure 15. forward _kernel function signature

```
void forward(mshadow::Tensor<gpu, 4, DType> &y, const mshadow::Tensor<gpu, 4, DType> &x, const mshadow::Tensor<gpu, 4, DType> &w)
```

Figure 16. forward _kernel function signature

The second feature that comes along with this optimization is loop unrolling. Loop unrolling allows us to reduce the number of branch instructions and the extra time and space for declaring iterators and integer calculations in a for-loop. To achieve this, we unroll the for-loop for calculating the sum of the products of the input feature maps and weight masks. The following snippet showcase the change in code for loop unrolling:

Figure 17. Before unrolling

Figure 18. After unrolling

Performance Results

10000 images

[final.py 10000 program output]:

```
✱ Running python final.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.108973
Op Time: 0.320080
Correctness: 0.8451 Model: ece408
```

100 images

[final.py 100program output]:

```
✱ Running python final.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.001077
Op Time: 0.003857
Correctness: 0.88 Model: ece408
```

10 images

[final.py 10 program output]:

```
✱ Running python final.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000142
Op Time: 0.000422
Correctness: 1.0 Model: ece408
```

Tuning with __restrict__ directives and loop unrolling - **Analysis**

As we expected, the features of restrict and loop unrolling combined gives us speedup of x1.1 on the default data set. Also, the bottleneck of the performance persists after this optimization as the logic of the implementation has not been changed much, our approach has a focus on compiler-level optimization. As a result, we can still see 100% control divergence rate, out of 38400000 executions (see Figure 19). Also, threads in warps are not fully utilized, there is high percentage of threads within warps are idle (see Figure 20). We are considering exploiting more parallelism in the program to efficiently utilize the resources available, and this is one of our next optimizations.

Figure 19. Only 78.6% of warp efficiency and 100% control divergence

, Figure 20. 18% of the threads were inactive during runtime

CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

Optimization 6&7 - *Exploiting parallelism in input channels & Atomic operations*

In this section, we are looking into the effect of further parallelizing the convolution layer by adding one more dimension in the thread blocks, from two to three dimensions. This is motivated by the fact that the amount of inactive within thread blocks were significantly large after the optimization 4&5 (18% of idle threads).

The additional layer of thread block (z dimension) help parallelize the calculation of productions of input feature maps and weight masks for each channel. For example, threads with threadIdx.z = 0 would calculate the output feature maps in the part where the correspond to the first channel of the input channel maps.

The second feature we have included is atomic operations. Since we have extra layer to facilitate parallelism now, memory contention could cause significant slow down: each output feature elements will be incremented by threadBlockDim.z number of threads (i.e. dimension of C) times. Thus, through atomic operations, we could reduce memory contention overhead.

Performance Results

10000 images

[final.py 10000 program output]:

```
✱ Running python final.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.115499
Op Time: 0.322260
Correctness: 0.8451 Model: ece408
```

100 images

[final.py 100program output]:

```
✱ Running python final.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.001215
Op Time: 0.003804
Correctness: 0.88 Model: ece408
```

10 images

[final.py 10 program output]:

```
✱ Running python final.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000148
Op Time: 0.000402
Correctness: 1.0 Model: ece408
```

Exploiting parallelism in input channels & Atomic operations - Analysis

As expected we have improved the efficiency of threads in each thread block, we have reduced the runtime of the second layer of convolution (x1.01 speedup) for smaller dataset from the last optimization. The implementation has utilized 5% more of the threads (see Figure 22.). Also, the bandwidth usage has also dropped to medium level due to we have higher concurrency (see Figure 24.). By adding another dimension, utilizing more threads, we have also soften the problem with control divergence, the percentage is halved (see Figure 21.). However, due to having more threads in execution. The synchronization overhead has become severe (see Figure 23.), and we hope to resolve that in the next optimization, by tuning our parameters (Optimization 9) such that we can identify a good grid and block dimension to minimize the synchronization overhead, while keeping the efficiency in each warp.

, Figure 21. Control divergence dropped from 100% to 50%

, Figure 22. Number of inactive threads dropped from 18% to 11%

, Figure 23. Long stalling time attributes to synchronization

, Figure 24. Level of bandwidth utilization has dropped due to parallelism

Optimization 8 - *Corner Turning*

The basis of this optimization is to load shared memory for our matrix multiplication in a coalesced way. Coalescing refers to accessing and delivering consecutive locations in memory so as to ensure we receive the entirety of each DRAM burst as we store our shared memory. If we organize our threads into a consecutive pattern when loading these arrays we should have a noticeable speed up. For our original code before optimization we were accessing and storing the shared memory arrays X and W in a row-major order. In our corner turned code we changed the storage pattern to a column-major order in order to coalesce our thread's storage to the weight array. We didn't do corner turning for the input feature maps because it won't be fruitful. Each thread has a designated tile to load into the shared memory, so adjacent threads can't access adjacent memory address in anyway. Below is the output of the corner turned code and nvprof outputs:

10000 images

[final.py 10000 program output]:

```
✱ Running python final.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.107390
Op Time: 0.315025
Correctness: 0.8451 Model: ece408
```

100 images

[final.py 100program output]:

```
✱ Running python final.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.001053
Op Time: 0.003804
Correctness: 0.88 Model: ece408
```

10 images

[final.py 10 program output]:

```
✱ Running python final.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000143
Op Time: 0.000414
Correctness: 1.0 Model: ece408
```

Corner Turning - **Analysis**

As seen by the outputs we have a very slight speed up compared to our previous loop unrolled version. Nvprof gives us insight into our shared memory alignment below:

, Figure 25. Access pattern in the shared memory

We can see that our ratio for shared load transactions/access is still somewhat off, explaining only the slight increase in performance. We are probably still missing coalescing of our write for X or read for X and W. However, we can observe a better result when looking into the memory bandwidth and utilization tab:

, Figure 26. Shared Total Bandwidth increased

For our L1/Shared Total Bandwidth, we increased from 1,137 GB/s to 1,406 GB/s. This is probably the explanation for our speed up after implementing corner turning. Overall, corner turning on W gave us better memory utilization by coalescing access allowing for higher bandwidth in our implementation.

Optimization 9 - *TILE_WIDTH Sweeping*

Because of the way nVidia's CUDA devices schedule and execute threads, with a warp size of 32, and because of control divergence, different sizes of our threadblocks can yield differing performance results. One potential way to optimize our program using this knowledge is to determine a value for the threadblock dimensions that minimizes our performance times. This is what this optimization is about. We ran our program with various **TILE_WIDTH**s at 10000 images and used the output ***Op Time***s to determine what an ideal **TILE_WIDTH** would be to

yield the best possible performance. We swept **TILE_WIDTH** from 4 to 32 in increments of 4 and compiled all of the outputs to determine an optimal size.

TILE_WIDTH Sweep - Analysis

Rather than present all of the individual outputs for each specific **TILE_WIDTH** (since this would clutter up this report quite a bit) we compiled the most important outputs, that is, the ***Op Time***s, and placed them into a table for varying block dimensions. Additionally, we placed these results into a column chart to better visualize trends in the data. The results were the following:

, Figure 27.Results of TILE_WIDTH sweeping

These results were quite interesting, but they can also be reasoned out. We can see that for Layer 1, the best performance was achieved at a **TILE_WIDTH** of 16, with 32 coming in at a close second. Layer 2 didn't seem to follow the trend as nicely as Layer 1, but it did exhibit similar patterns. A width of 28 provided the best performance as opposed to 16 although 16 was a close second and 32 a close third. The fact that 28 provided the best performance was a bit of an outlier, but the general trend can be explained.

We expect that block sizes which yield a multiple of 32 will provide the best performance, since blocks with a dimension that is not a multiple of 32 will have bigger issues with control divergence. The fact that 16 provides a better performance than 32 can likely be explained by the fact that a block size of 256 provides more granularity than a block size of 1024, therefore making boundary conditions less of an issue as the boundaries will have less diverging threads. Also, it could be due to smaller block dimension can minimize the overhead of synchronization stalling (see Figure 28.), which we have observed to be a bottleneck of the performance in Optimization 6&7. We are observing a similar graph for execution counts with all threads combined, so the work distribution to threads doesn't attribute performance improvement.

, Figure 28. New pie chart of stall reasons

, Figure 29. Similar execution count as above

Larry Poon (Chicago-651278853), Gerald Kozel (Chicago-671266382), & Waleed Ramahi (UIUC-667747340)

References

- Textbook Chapter 16 (https://wiki.illinois.edu/wiki/download/attachments/658396897/3rd-Edition-Chapter16-case-study-DNN-FINAL-corrected.pdf?version=1&modificationDate=1515766344000&)

- Textbook Chapter 5 (https://wiki.illinois.edu/wiki/download/attachments/658396897/3rd-Edition-Chapter05-performance-FINAL-corrected.pdf?version=1&modificationDate=1515766350000&)

- Piazza posts (@367, @310)