

Team Name: we_got_cookies

School Affiliation: Campus Students @ UIUC in ECE 408

Final Report for ECE 408

Name	NetID	UIN	RAI ID
Pranith Bottu	pbottu2	660359335	5d97b1a588a5ec28f9 cb941d
Jesse Wu	jessezw2	651680865	5d97b22188a5ec28f9 cb9503
Daniel Chung	jdchung2	664591251	5d97b1b488a5ec28f9 cb9438

Report Data

Include a list of all kernels that collectively consume more than 90% of the program time (m1.2.py)	[CUDA memcpy HtoD] : 30.54% volta_scudnn_128x64_relu_interior_nn_v1 : 17.89% volta_gcgemm_64x32_nt : 17.29% fft2d_c2r_32x32 : 8.80% volta_sgemm_128x128_tn : 7.84% op_generic_tensor_kernel : 6.60% cudnn::detail::pooling_fw_4d_kernel : 6.47%
Include a list of all CUDA API calls that collectively consume more than 90% of the program time (m1.2.py)	cudaStreamCreateWithFlags : 41.18% cudaMemGetInfo : 33.16% cudaFree : 21.18%
Include an explanation of the difference between kernels and API calls	A kernel is the operating system's core program that interfaces with external devices such as the CPU, RAM, etc. The kernel is also the system's lowest level program. An API is an interface for programmers to exchange data between systems, applications, and devices using various libraries. While an API may sound similar to a kernel, kernels do not have libraries that it can use unlike APIs.
Show output of rai running MXNet on the CPU (mp1.1.py)	Loading fashion-mnist data... done Loading model... done

	New Inference EvalMetric: {'accuracy': 0.8154} 16.88user 4.95system 0:08.99elapsed 242%CPU (0avgtext+0avgdata 6045996maxresident)k 0inputs+2824outputs (0major+1601964minor)pagefaults 0swaps
List program run time (mp1.1.py)	User : 16.88 System : 4.95 Elapsed : 0:08.99
Show output of rai running on MXNet on the GPU (mp1.2.py)	Loading fashion-mnist data... done Loading model... done New Inference EvalMetric: {'accuracy': 0.8154} 4.94user 3.32system 0:04.76elapsed 173%CPU (0avgtext+0avgdata 2967320maxresident)k 0inputs+1720outputs (0major+728204minor)pagefaults 0swaps
List program run time (mp1.2.py)	User : 4.94 System : 3.32 Elapsed : 0:04.76
List whole program execution time (mp2.1.py)	User : 98.22 System : 8.81 Elapsed : 1:23.89
List Op Times (mp2.1.py)	Op Time : 13.244454 Op Time : 65.510644

Correctness and Timing w/ 3 Different Dataset Sizes

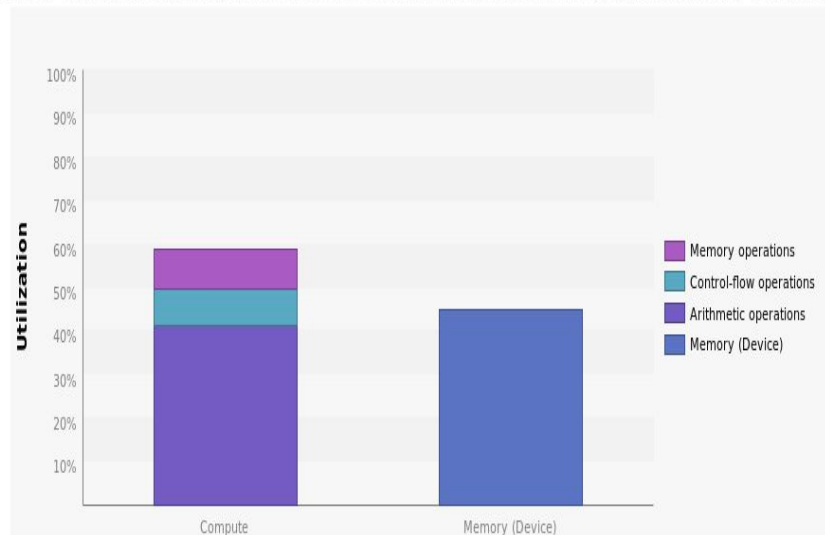
Model	Number of Images	Correctness	Timing
ece408	100	0.76	User : 5.10 System : 2.08 Elapsed : 0:05.04 Op Time : 0.000274 Op Time : 0.000931
ece408	1000	0.767	User : 4.97 System : 2.76 Elapsed : 0:04.68

			Op Time : 0.002973 Op Time : 0.009900
ece408	10000	0.7653	User : 5.22 System : 3.22 Elapsed : 0:04.87 Op Time : 0.030726 Op Time : 0.098592

Demonstration of nvprof Profiling the Execution

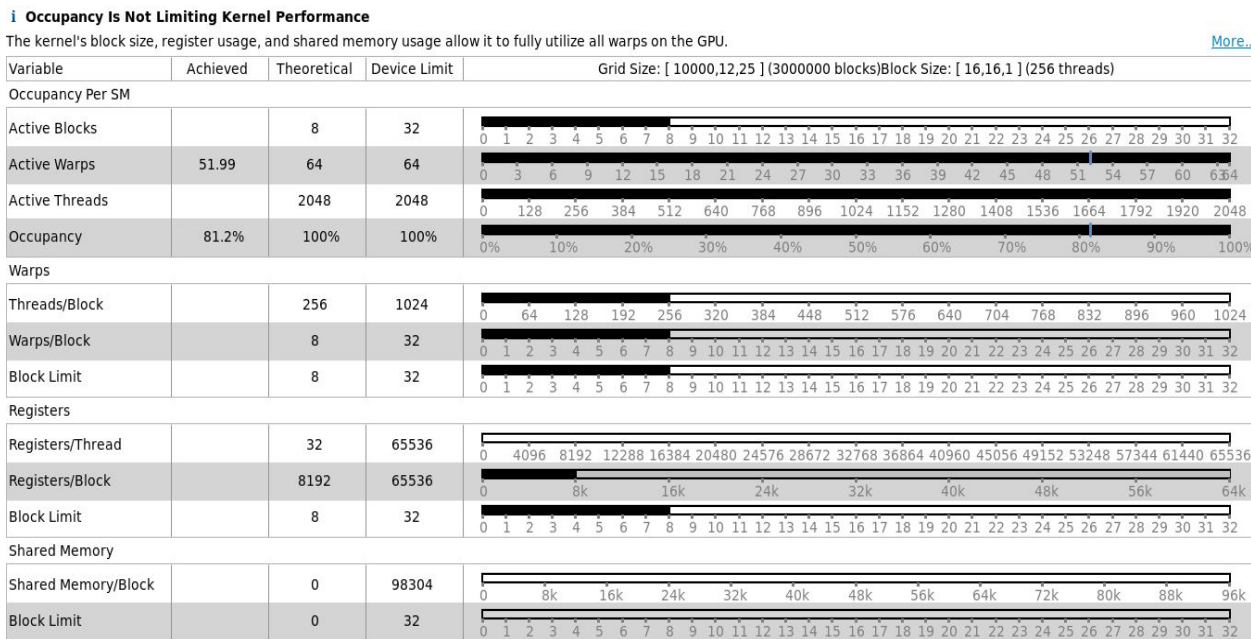
i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

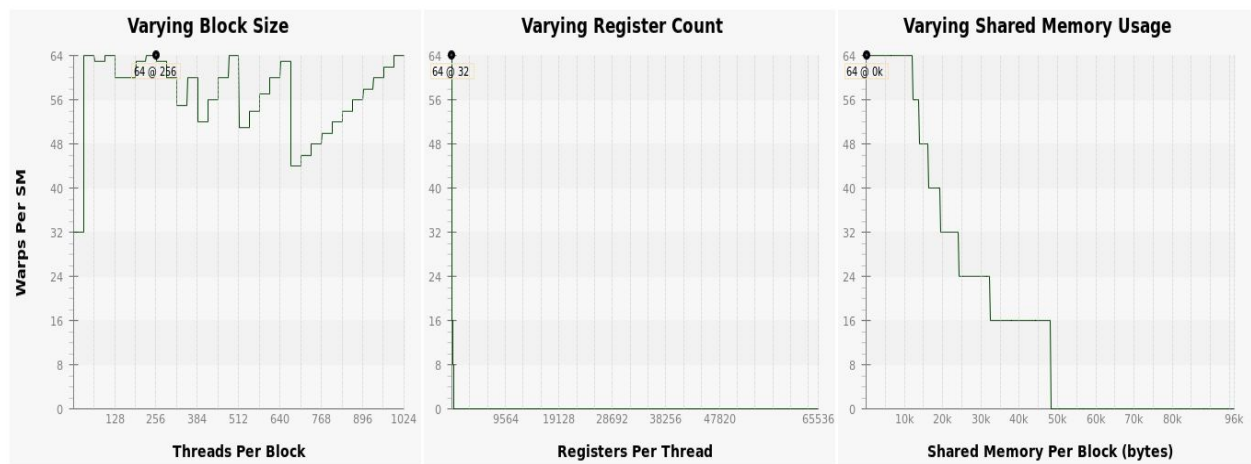


The above screenshot shows the result of running Kernel Analysis. As we can see, our current kernel has a significant latency issue, which would make sense at this stage of the project since our current kernel heavily relies on global memory accesses. Because so many threads are trying to access the global memory, and because our memory bandwidth is limited, this creates a bottleneck and negatively affects the performance of our kernel. We expect to see a decrease in latency when we move forward in optimizing our kernel. One method of doing so would be implementing shared memory to our tiling method when doing convolution computation in the

kernel code. We can also use constant memory to store our kernel mask instead of global memory.



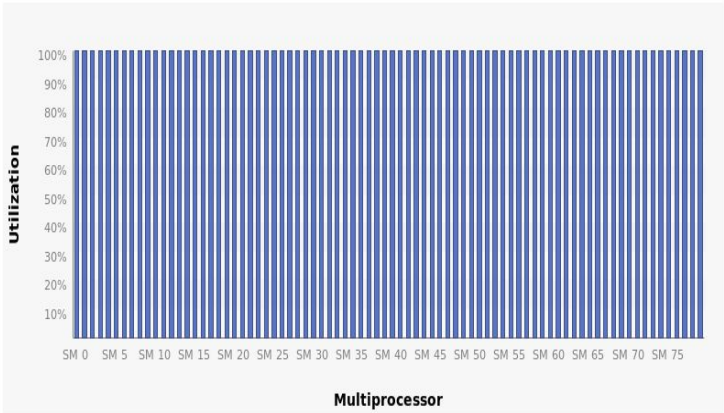
The above screenshot shows more specific data regarding the usage of blocks, threads, warps, and more. As we can see, the number of threads and warps that are active during this execution is max, as we are putting in a total of 10000 images. The active blocks are also used to its max, as we can see that 8 blocks, which are the max number of blocks, are used. Overall, we know that this act of forward convolution takes a lot of threads and memory to execute.



The above screenshot shows more important information about how the threads, blocks, and warps are utilized differently with respect to each other. It offers insight into how the code is implemented in the GPU. You can see for example how the number of warps per SM fluctuates with respect to the number of threads per block used.

i Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel. [More...](#)



The screenshot above shows the usage of all SMs during the execution of our kernel. We can see that usage is at 100% across the board. This is because 10000 images are being used, which results in every thread in SM to be used for the execution of forward convolution.

⚠ Low Global Memory Load Efficiency [kernels accounting for 100% of compute have low efficiency (57.1% avg)]
Global load efficiency indicates how well the application's global loads are using device memory bandwidth. The efficiency is the number of bytes requested divided by the number of bytes that were transferred from device memory to satisfy those requests. Because device memory transfers bytes in blocks, the alignment and access pattern of a given load determines how many blocks must be transferred and thus determines the efficiency of that load. Low efficiency indicates that one or more global memory loads have a poor access pattern or alignment. Select this result to highlight kernels with low global load efficiency. [More...](#)


⚠ Low Global Memory Store Efficiency [kernels accounting for 100% of compute have low efficiency (68.8% avg)]
Global store efficiency indicates how well the application's global stores are using device memory bandwidth. The efficiency is the number of bytes stored divided by the number of bytes that were transferred to device memory to perform those stores. Because device memory transfers bytes in blocks, the alignment and access pattern of a given store determines how many blocks must be transferred and thus determines the efficiency of that store. Low efficiency indicates that one or more global memory stores have a poor access pattern or alignment. Select this result to highlight kernels with low global store efficiency. [More...](#)

⚠ Low Warp Execution Efficiency [kernels accounting for 100% of compute have low efficiency (76.6% avg)]
Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The warp execution efficiency for these kernels is 84.8% if predicated instructions are not taken into account. These kernels' not predicated off warp execution efficiency of 76.6% is less than 100% due to divergent branches and predicated instructions. Select this result to highlight kernels with low warp execution efficiency. [More...](#)

i Kernel Optimization Priorities
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)

As you can see above, our efficiency statistics are low across the board. This makes sense because we haven't made any optimizations for this milestone yet. In the future, we may utilize techniques such as shared memory to further optimize our code. These techniques will reduce the number of global memory accesses as well as other potential time sinks, thereby increasing our efficiency.


Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

[More...](#)

Line / File	new-forward.cuh - /mxnet/src/operator/custom
39	Divergence = 16.5% [3960000 divergent executions out of 24000000 total executions]

We can see from the above screenshot that our kernel has a control divergence of 16.5%, which slows down our execution by a bit. This divergence is due to for loops of going through different input feature maps. When it's executing the input feature maps to do the convolution with the weight matrix, it goes through different bounds to compute the output feature map. During the process of each computation, we incur control divergence. In order to prevent this, instead of using threads that are adjacent to each other, we can call the thread by its warp or by block to prevent the small percentage of thread divergence.

Description of Optimizations for Optimizations 1 - 4

Optimization 1: Parameter Sweeping	<p>This entailed us modifying our TILE_WIDTH parameter to the most optimal size. The optimal size was determined through numerous runs to see which size gave the fastest Op Time. After numerous trials, we determined that the best TILE_WIDTH was 24.</p> <p>This makes sense as we know that each warp has 32 threads, and by having TILE_WIDTH of 24, we essentially have 576 threads given that we're using 2-dimensional shared memory. This is exactly divisible by 32 threads. Depending on different SM we use for our parallel execution, the hardware designs and the tile_width that gives us the most utilization of different blocks and threads in the SM will give us the biggest boost. We assume that each block will have the size 576 threads per block, and has enough blocks to utilize the blocks in the Streaming</p>
---	--

	<p>multiprocessor better than all the other block sizes.</p>
<p>Optimization 2: Tuning w/ Restrict & Loop Unrolling</p>	<p>Tuning w/ Restrict prevents the program from using different array pointers to point to the same location. By doing this, the compiler no longer has to worry about checking pointer accesses all the time. This improves the kernel performance. In our code, there were three variables we could restrict: y, x, and k.</p> <p>Loop unrolling uses register-level memory transfers and increases thread activity since it allows for more reads per thread. The one downside is that we're limited by the register memory, which is smaller than the GPU storage. In our code, there are three loops that we can unroll.</p>
<p>Optimization 3: Weight Matrix (Kernel Values) in Constant Memory</p>	<p>Storing the Weight Matrix in Constant Memory allows for quicker memory accesses to these values, as the constant memory is effectively a cache. Global memory reads are expensive and time consuming, so by storing these weight values that we'll the kernel will always use in constant memory, we can alleviate unnecessary global memory accesses. We are able to store the weight matrix in constant memory as it is small enough to fit within the 65 KB size limitation.</p>
<p>Optimization 4: Shared Memory Convolution</p>	<p>By putting input in shared memory, we save a lot of time that would otherwise have been wasted trying to access it from global memory numerous times. The way the code is written, the input is utilized in numerous locations. Each time we access global memory, there is a steep cost associated with accessed something that far away. The shared memory alternative drastically shortens that cost so that the numerous calls to input don't affect us nearly as much. However, it has to be accounted for in the cases where loading theses shared memory might take longer than using shared memory values. Therefore, the bigger, and more the contents of the elements stored in shared memory is used, the better the optime will be. Also, syncthreads() where we have to wait for other threads to be done computing, may take a lot of our optimization time. In this code, the filter banks are also moved to shared memory instead of constant memory.</p>

Implementation of the Different Optimizations for Optimizations 1 - 4:

```

1
2 #ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
3 #define MXNET_OPERATOR_NEW_FORWARD_CUH_
4 #define TILE_WIDTH 16
5 #define CACHE_SIZE 16000
6
7 #include <mxnet/base.h>
8
9 namespace mxnet
10 {
11     namespace op
12     {
13         __constant__ float weight_cache[CACHE_SIZE];
14
15         __global__ void forward_kernel(float * __restrict__ y, const float * __restrict__ x, const int B, const int M,
16         {
17             int n, m, h0, w0, h_base, w_base, h, w;
18             const int H_out = H - K + 1;
19             const int W_out = W - K + 1;
20
21             #define y4d(i3, i2, i1, i0) y[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]
22             #define x4d(i3, i2, i1, i0) x[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
23             #define k4d(i3, i2, i1, i0) weight_cache[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
24
25             int W_grid = ceil((float)W_out/TILE_WIDTH); //number of tiles in x-dimension
26             int H_grid = ceil((float)H_out/TILE_WIDTH);
27             int X_tile_width = TILE_WIDTH + K-1;
28             extern __shared__ float shmem[];
29             float* X_shared = &shmem[0];
30             n = blockIdx.x;
31             m = blockIdx.y;
32             h0 = threadIdx.y;
33             w0 = threadIdx.x;
34             h_base = blockIdx.z / H_grid*TILE_WIDTH;
35             w_base = blockIdx.z % W_grid*TILE_WIDTH;
36             h = h_base + h0;
37             w = w_base + w0;
38
39             float acc=0;
40             int c, j, i, p, q;
41
42             #pragma unroll
43             for (c = 0; c < C; c++) {
44                 #pragma unroll
45                 for (i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
46
47                     #pragma unroll
48                     for (j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
49                     {
50                         X_shared[(i - h_base)*X_tile_width+ (j - w_base)] = x4d(n, c, i, j);
51                     }
52                 }
53             }
54         }
55     }
56 }

```

Optimization 1:
Parameter Sweeping

Optimization 3:
Constant Memory

Optimization 2:
Tuning (Restrict)

Optimization 4:
Shared Memory

Optimization 2:
Tuning (Unroll)

GPU Optimization Results for Optimizations 1 - 4 (data size = 10000)

Baseline / Optimization 1: Parameter Sweeping	User : 5.40 System : 3.13 Elapsed : 0:05.25 Op Time : 0.030781 Op Time : 0.096756
Optimization 2: Tuning w/ Restrict & Loop Unrolling	User : 5.09 System : 2.93 Elapsed : 0:04.98 Op Time : 0.027204 Op Time : 0.095210
Optimization 3: Weight Matrix (Kernel Values) in Constant Memory	User : 5.14 System : 2.99 Elapsed : 0:05.24 Op Time : 0.030429 Op Time : 0.092568
Optimization 4: Shared Memory Convolution	User : 5.35 System : 2.99 Elapsed : 0:05.99 Op Time : 0.036083 Op Time : 0.106366
All Above Optimizations in one Kernel	User : 5.14 System : 3.08 Elapsed : 0:05.00 Op Time : 0.036039 Op Time : 0.083120

From the table above, it is important to note the Op Times of our kernel that utilizes all of our optimizations. While the first layer's Op Time is somewhat slower compared to the baseline, we find that our second layer's Op Time is considerably faster than the baseline as well as each singularly applied optimization. This is mainly due to the shared memory convolution portion of the optimized kernel. Since the first layer has a small set of input data, it does not benefit from shared memory as much, as it still takes time to load the input data to shared memory. However, with a large input dataset such as the one in layer 2, the performance gains from shared memory convolution is a lot more noticeable, as the kernel doesn't spend as big of a percentage of its time

loading data into shared memory like layer 1, but rather it is large enough to actually read from shared memory more than loading into it. As such, we were able to see a fairly big performance gain on the layer 2 kernel run.

Demonstration of nvprof Profiling the Execution for Optimizations 1 - 4

Low Global Memory Load Efficiency [kernels accounting for 100% of compute have low efficiency (53.9% avg)]	
Global load efficiency indicates how well the application's global loads are using device memory bandwidth. The efficiency is the number of bytes requested divided by the number of bytes that were transferred from device memory to satisfy those requests. Because device memory transfers bytes in blocks, the alignment and access pattern of a given load determines how many blocks must be transferred and thus determines the efficiency of that load. Low efficiency indicates that one or more global memory loads have a poor access pattern or alignment. Select this result to highlight kernels with low global load efficiency.	
More...	
Low Global Memory Store Efficiency [kernels accounting for 100% of compute have low efficiency (68.8% avg)]	
Global store efficiency indicates how well the application's global stores are using device memory bandwidth. The efficiency is the number of bytes stored divided by the number of bytes that were transferred to device memory to perform those stores. Because device memory transfers bytes in blocks, the alignment and access pattern of a given store determines how many blocks must be transferred and thus determines the efficiency of that store. Low efficiency indicates that one or more global memory stores have a poor access pattern or alignment. Select this result to highlight kernels with low global store efficiency.	
More...	
Low Shared Memory Efficiency [kernels accounting for 100% of compute have low efficiency (48.6% avg)]	
Shared memory efficiency indicates how well the application's shared memory accesses are using the available shared memory bandwidth. The efficiency is the number of shared loads and stores divided by the number of shared memory transactions required to perform those loads and stores. The alignment and access pattern of a given shared memory access determines how many transfers are required and thus determines the efficiency of that access. Low efficiency indicates that one or more shared memory accesses have a poor access pattern or alignment. Select this result to highlight kernels with low shared memory efficiency.	
More...	
Kernel Optimization Priorities	
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.	
Rank	Description
100	[1 kernel instances] mxnet::op::forward_kernel(float*, float const *, int, int, int, int, int)

In our optimizations, we don't optimize global memory in any way. Therefore, it makes sense that the Global Memory Efficiency and Global Memory Store Efficiency are low. Our first three optimizations were more GPU-oriented. Our first optimization tackled the question of which TILE_WIDTH was best to optimize the thread and block usage. The second optimization was geared towards saving time on checking for valid pointers and increasing thread activity to better kernel performance. The third optimization was more for saving cost for accessing the same matrix over and over again by storing it in constant memory. The fourth optimization dealt with shared memory. None of these affected global memory so it makes sense that global memory is still "Low". The line stating "Low Shared Memory Efficiency" is not a major concern right now since we only implemented a very basic cost-saver via tiling. This is not enough to show a drastic improvement from the already bad shared memory. With time and more optimizations, this should change. As such, there is no major concern with what we're seeing right now.

Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

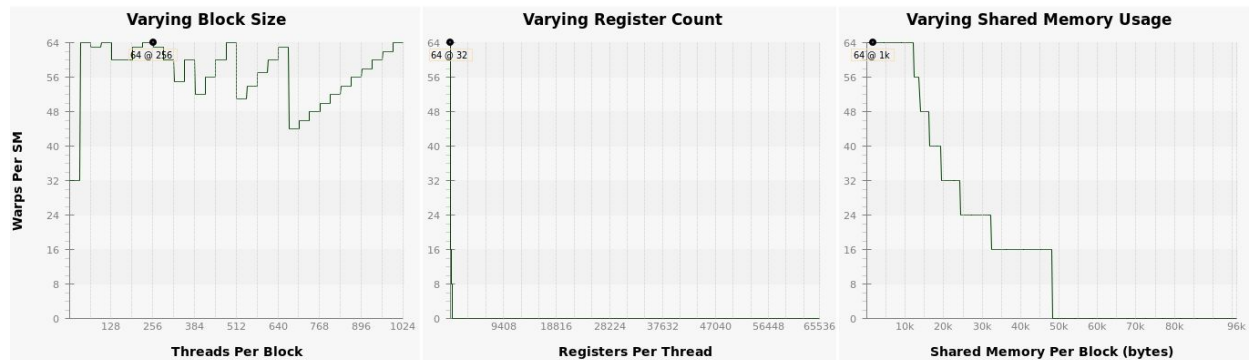
Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

[More...](#)

Line / File	new-forward.cuh - /mxnet/src/operator/custom
62	Divergence = 100% [30000000 divergent executions out of 30000000 total executions]
62	Divergence = 100% [30000000 divergent executions out of 30000000 total executions]
76	Divergence = 16.5% [3960000 divergent executions out of 24000000 total executions]

While we can see from the GPU Optimization Results table that our optimization implementations were largely successful in reducing the kernel's Op Times, it is clear from the image above that we have yet to tackle our kernel's issue of thread divergence, which is something that we are now mindful of going into the final milestone. At line 76 of new-forward.cuh, we are implementing an if statement that controls which threads are allowed to write to the output data. Since this if statement essentially blocks certain threads that doesn't meet its conditions, it makes sense that there is some divergence at this point of the kernel code as certain threads within a warp may be turned off from doing anything.

Line 62 of new-forward.cuh is an interesting case to our optimized kernel, as we have introduced 100% control divergence at that point of the kernel. We suspect that the termination condition for the for-loop we used in our kernel is causing thread branching within all warps, and this is something that we will look into further.



The above screenshot shows more important information about how the threads, blocks, and warps are utilized differently after our optimizations. It offers insight into how the code is implemented in the GPU. You can see for example how the number of warps per SM fluctuates with respect to the number of threads per block used.

i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



Although we tried to decrease our memory latency issue by using shared memory and constant memory, it seems like it didn't seem to change too much. One reason for this may be that compared to loading the values from global memory to constant or shared memory, the function actually doesn't call the elements in the input maps or filter banks as much in proportion to the amount of loading it does from global memory to constant/shared memory. This might be due to small filter bank size, or even small data inputs, which will have less read/write compared to bigger filter banks or input data.

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. [More...](#)

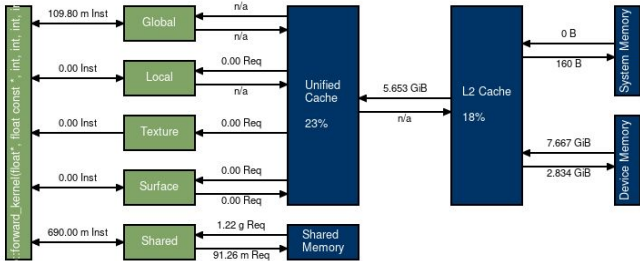
	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	1220278255	4,274.204 GB/s	
Shared Stores	91257454	319.643 GB/s	
Shared Total	1311535709	4,593.846 GB/s	<div><div></div></div>
L2 Cache			
Reads	196295292	171.888 GB/s	
Writes	95040504	83.223 GB/s	
Total	291335796	255.112 GB/s	<div><div></div></div>
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	278390414	243.776 GB/s	
Global Stores	95040000	83.223 GB/s	
Texture Reads	777650352	2,723.834 GB/s	
Unified Total	1151080766	3,050.833 GB/s	<div><div></div></div>
Device Memory			
Reads	257246590	225.261 GB/s	
Writes	95103305	83.278 GB/s	
Total	352349895	308.539 GB/s	<div><div></div></div>
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	<div><div></div></div>
Writes	5	4.378 kB/s	<div><div></div></div>

We can see from the image above that our kernel has a moderate use of shared memory. This makes sense as one of our kernel optimizations is applying tiling over a large input dataset. We can also see that our kernel utilizes some memory from cache. This is because we implemented constant memory to store the filter banks. Our rationale for this is that since the filter banks are

fairly smaller in size and we are always using these filter banks in our convolution kernel, we can avoid costly global memory reads by storing them in constant memory. Again, as mentioned before, the filter banks do not utilize much memory, so it makes sense that our cache utilization is fairly low.

i Memory Statistics

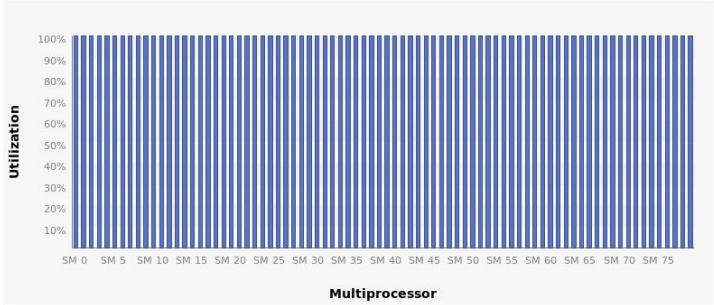
The following chart shows a summary view of the memory hierarchy of the CUDA programming model. The green nodes in the diagram depict logical memory space whereas blue nodes depicts actual hardware unit on the chip. For the various caches the reported percentage number states the cache hit rate; that is the ratio of requests that could be served with data locally available to the cache over all requests made. The links between the nodes in the diagram depict the data paths between the SMs to the memory spaces into the memory system. Different metrics are shown per data path. The data paths from the SMs to the memory spaces report the total number of memory instructions executed, it includes both read and write operations. The data path between memory spaces and "Unified Cache" or "Shared Memory" reports the total amount of memory requests made (read or write). All other data paths report the total amount of transferred memory in bytes.



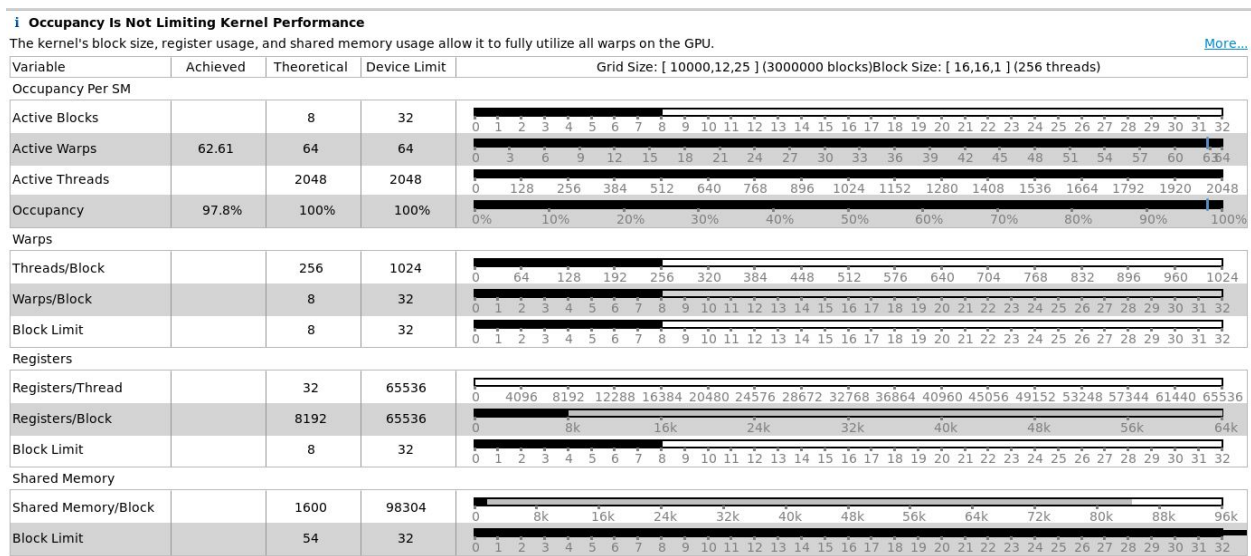
As seen above, we can see that our optimizations have definitely utilized cache and shared memory to reduce the latency issue that delays the optimization time. We know filter banks only cover a small percentage of how much cache can store, which is seen by low percentage of cache utilization. There are some read and write operations that are done using shared memory in the function, and also read and write functions in cache has been utilized straight from device memory.

i Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel. [More...](#)



It is obvious that all the threads and blocks that exist in multiprocessor are all utilized to 100%. This makes sense as there are more than 10000 data images that are coming in, and with just having the max of 2048 threads working, we have to fully utilize all the threads to compute the convolution as soon as possible. Because there are more images than threads can handle, each thread will process to compute the next convolution as soon as it's done with the designated task.



The above screenshot shows more specific data regarding the usage of blocks, threads, warps, and more. As we can see, the number of threads and warps that are active during this execution is max, as we are putting in a total of 10000 images.

The active blocks are also used to its max, as we can see that 8 blocks, which are the max number of blocks, are used. Another thing to note is that the shared memory usage has gone up a lot compared to Milestone 3. In Milestone 3, the diagram displayed 0 “Shared Memory/Block” and 0 “Block Limit” due to the lack of usage. This time around, our shared memory optimization raised those values to 1600 and 54 respectively. The other values were unaffected since our optimizations weren’t geared towards impacting them. In the future, this might change.

Overall, we know that this act of forward convolution takes a lot of threads and memory to execute. The effect of that is seen with the bulk amount of threads and memory used. While shared memory gives a new form of storage and usage, there is still a lot of computations being done and the magnitude of the numbers seen reflect this.

Description of Optimizations 5 & 6

<p>Optimization 5: Kernel Fusion for Unrolling & Matrix Multiplication</p>	<p>Kernel fusion allows for converting 2 or more kernels into 1 so that we're able to reuse relevant data and reduce redundant loads and stores. This saves us a lot of time that would otherwise have been spent doing other tasks. The "reuse" of data is particularly useful because it prevents wasteful use of resources retrieving it numerous times. We eliminate the transfers of memory to 2 separate kernels.</p> <p>We also converted our matrix multiplication into one large GEMM matrix-matrix multiplication. We realign our inputs so that each row of the matrix contains all the input values necessary to compute one element of an output feature. The reason this helps us is that this gives us the regular patterns of memory access throughout the computation of each output. Also, by putting the block into one sequential block, it reduces the forward operation of the convolution layer that needs to be done into just a simple one large matrix-matrix multiplication.</p>
<p>Optimization 6: Multiple Kernel Implementations for Different Layer Sizes</p>	<p>Using multiple kernel implementations for different layer sizes allows us to tailor to each convolution</p> <p>M= (data width size #1: 12) , (data width size #2: 24)</p> <p>We were given two data inputs to test our convolutions of different optimizations. We decided to split the input data into multiple kernel implementations based on their size, as we know that the bigger that matrix is, the more elements that will be accessed throughout the code. Thus, this will allow more efficient use of shared memory if there are more data elements. However, with small data</p>

	<p>input, loading the data from global to shared actually take up more time than accessing it directly. Therefore, in the two kernels we split up, we made it so that one convolution use global memory and constant memory while the other one used shared memory and global memory. By splitting the data like this, we are able to match the data input into most optimal convolution that can give us the fastest compute time.</p>
--	---

Implementation of the Different Optimizations for Optimizations 5 & 6:

The Actual Splitting done in the Host Code

```

if(M % TILE_WIDTH_ONE != 0)
{
    //printf("in the first pass");
    W_grid = ceil((float)W_out/TILE_WIDTH_ONE);
    H_grid = ceil((float)H_out/TILE_WIDTH_ONE);
    Z = H_grid * W_grid;
    dim3 blockDim(TILE_WIDTH_ONE,TILE_WIDTH_ONE,1);
    dim3 gridDim(B,M,Z);
    int kernelSize = w.shape[0] * w.shape[1] * w.shape[2] * w.shape[3];
    cudaMemcpyToSymbol(weight_cache, w.dptr_, sizeof(float)*kernelSize, 0, cudaMemcpyDefault);
    forward_kernel<<gridDim, blockDim>>>(y.dptr_,x.dptr_, B,M,C,H,W,K);
}
else{
    //printf("in the second pass");
    dim3 blockDim(TILE_WIDTH_TWO,TILE_WIDTH_TWO,1);
    dim3 gridDim(ceil(1.0*H_out*W_out/TILE_WIDTH_TWO),ceil(1.0*M/TILE_WIDTH_TWO),B);
    forward_kernel_two<<gridDim, blockDim>>>(y.dptr_,x.dptr_,w.dptr_, B,M,C,H,W,K);
}

```

```

__global__ void forward_kernel(__restrict__ __attribute__((aligned(16))) float* __restrict__ y, const float* __restrict__ x, const float* __restrict__ w, const int W, const int H, const int C, const
{
    #define xdd(i1, i2, i0) x[(i1) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]

    __shared__ float Mat_X[TILE_WIDTH_TWO][TILE_WIDTH_TWO];
    __shared__ float Mat_K[TILE_WIDTH_TWO][TILE_WIDTH_TWO];
    int temp_h, temp_w, temp_c, temp_p, temp_q;

    const int H_out = H - K + 1;
    const int W_out = W - K + 1;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int row = blockIdx.y * TILE_WIDTH_TWO + ty;
    int col = blockIdx.x * TILE_WIDTH_TWO + tx;
    int img = blockIdx.z;

    int width = C*K*K; // The number of elements need to get each output element
    int height = H_out * W_out; // number of elements in each output feature map

    float result = 0;

    #pragma unroll 4
    for(int i = 0; i < ceil(width/(1.0*TILE_WIDTH_TWO)); i++)
    {
        int temp_row = TILE_WIDTH_TWO * i + ty;
        int temp_col = TILE_WIDTH_TWO * i + tx;

        if(temp_col < width && row < H)
        {
            Mat_K[ty][tx] = k[row * width + temp_col];
        }
        else{
            Mat_K[ty][tx]=0;
        }

        if(temp_row < width && col < height)
        {
            temp_h = col/W_out;
            temp_w = col%W_out;
            temp_c = temp_row / (K*K);
            temp_row = temp_row % (K*K);
            temp_p = temp_row / K;
            temp_q = temp_row % K;
            Mat_X[ty][tx] = xdd(img, temp_c, temp_h+temp_p, temp_w+temp_q);
        }
        else{
            Mat_X[ty][tx] = 0;
        }

        __syncthreads();
    }
    #pragma unroll 24
    for(int j = 0; j < TILE_WIDTH_TWO; j++)
    {
        result += Mat_K[ty][j]*Mat_X[j][tx];
    }
    __syncthreads();
}

if(row%W && col < height)
{
    int index_off = img*W*height;
    y[index_off+row*height+col] = result;
}

#undef xdd

```

Optimization 5:
Matrix Multiply
Kernel

```

constant__ float weight_cache[CACHE_SIZE];
global__ void forward_kernel(float * __restrict__ y, const float * __restrict__ x, const int B, const int M, const int C, const int H, const int W, const int K)
{
    /*
     * Modify this function to implement the forward pass described in Chapter 16.
     * We have added an additional dimension to the tensors to support an entire mini-batch.
     * The goal here is to be correct AND fast.
     * We have some nice #defs for you below to simplify indexing. Feel free to use them, or create your own.
     */
    int n,m,h,w,c,p,q;
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;

    // An example use of these macros:
    // float a = y4d(0,0,0,0);
    // y4d(0,0,0,0) = a
    #define y4d(i1, i2, i1, i0) y[(i1) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]
    #define x4d(i1, i2, i1, i0) x[(i1) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
    #define k4d(i1, i2, i1, i0) weight_cache[(i1) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    int W_grid = ceil((float)W_out/TILE_WIDTH_ONE);

    n = blockIdx.x;
    m = blockIdx.y;
    h = blockIdx.z / W_grid * TILE_WIDTH_ONE + threadIdx.y;
    w = blockIdx.z % W_grid * TILE_WIDTH_ONE + threadIdx.x;
    float acc=0;
    if(h<H_out && w<W_out)
    {
        for(c=0; c<C; c++){
            for(p=0; p<K; p++){
                #pragma unroll 5
                for(q=0; q<K; q++){
                    {
                        acc += x4d(n,c,h+p,w+q)*k4d(m,c,p,q);
                    }
                }
            }
        }
        y4d(n,m,h,w) = acc;
    }

    #undef y4d
    #undef x4d
    #undef k4d
}

```

*Optimization 6:
2nd Kernel Implementation*

GPU Optimization Results for 5 & 6 (data size = 10000)

Optimization 5: Kernel Fusion for Unrolling & Matrix Multiplication	User : 5.13 System : 4.13 Elapsed : 0:05.72 Op Time : 0.040524 Op Time : 0.054739
Optimization 6: Multiple Kernel Implementations for Different Layer Sizes	User : 5.27 System : 3.33 Elapsed : 0:04.84 Op Time : 0.019133 Op Time : 0.052187 Note* These times are also a result from the use of constant memory, pragma unrolls and variable restricts, and parameter sweeping along with optimization 6.

Demonstration of nvprof Profiling the Execution for Optimizations 5 & 6

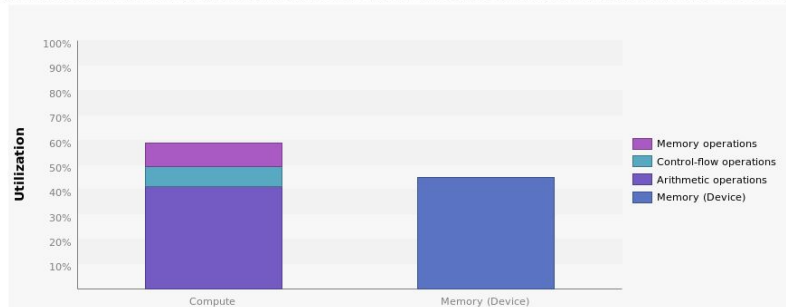
Optimization Approach and Results for Optimizations 1 - 6

Baseline Recap:

```
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.030720
Op Time: 0.098429
Correctness: 0.7653 Model: ece408
5.21user 3.24system 0:04.91elapsed 172%CPU (0avgtext+0avgdata 2954012maxresident
)k
0inputs+4560outputs (0major+732194minor)pagefaults 0swaps
```

Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



We can see from the NVPROF analysis that the baseline kernel is largely bounded by memory latency, which makes sense as it is reading data entirely from global memory.

Divergent Branches

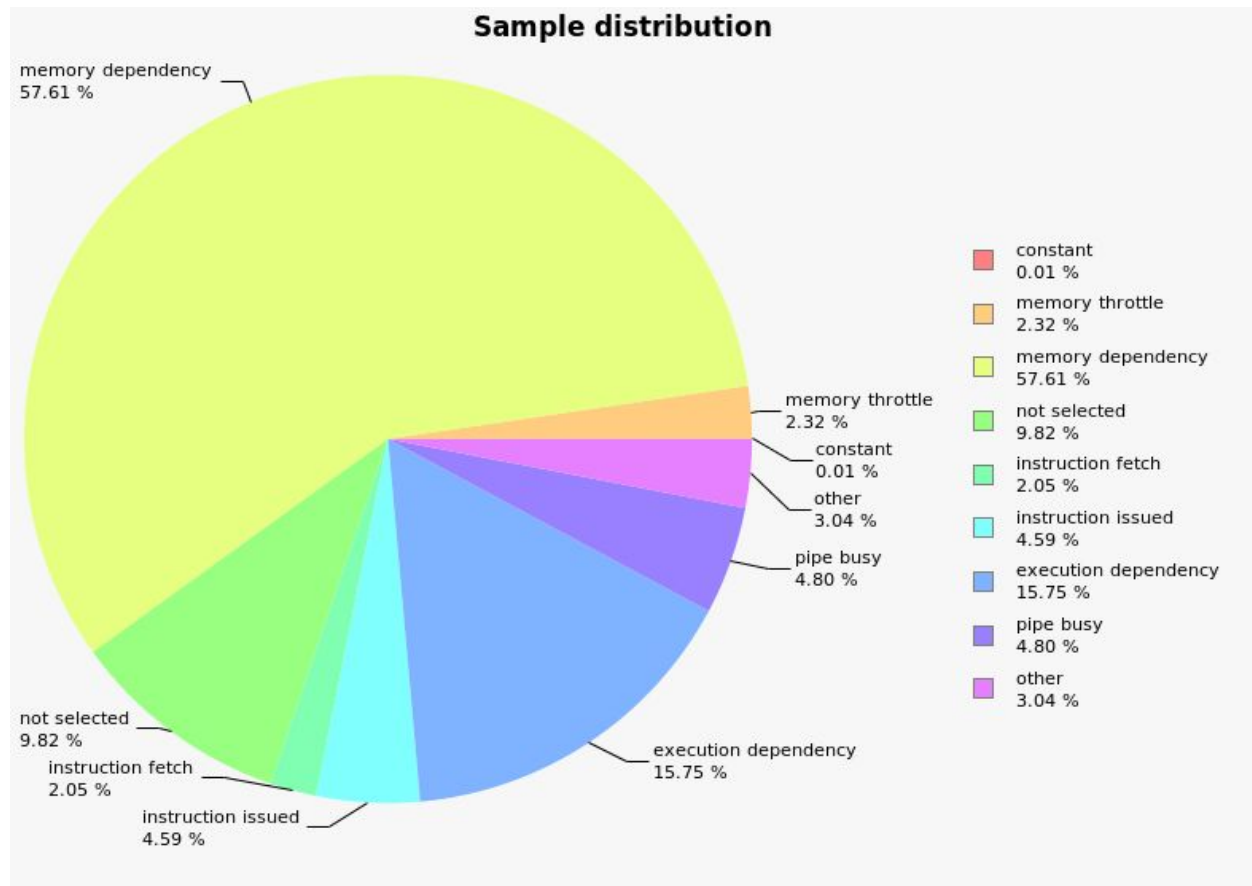
Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

[More...](#)

Line / File	new-forward.cuh - /mxnet/src/operator/custom
39	Divergence = 16.5% [3960000 divergent executions out of 24000000 total executions]

The baseline kernel also has some control divergence, however we do not think this is the major limiting factor of the baseline kernel's performance, as discussed above performance is bounded by memory latency.



We can see here from the sample distribution generated by NVPROF that our baseline kernel has quite a substantial memory dependency percentage, which we believe is one of the major causes of the baseline kernel's slow performance. So, one of our major motivations in improving the baseline kernel is to reduce this dependency.

i Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU.

[More...](#)

Variable	Achieved	Theoretical	Device Limit	Grid Size: [10000,12,25] (3000000 blocks)Block Size: [16,16,1] (256 threads)
Occupancy Per SM				
Active Blocks		8	32	
Active Warps	51.96	64	64	
Active Threads		2048	2048	
Occupancy	81.2%	100%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
Registers				
Registers/Thread		32	65536	
Registers/Block		8192	65536	
Block Limit		8	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	

Here we can see that the baseline kernel is fully utilizing all possible warps allowed by the GPU.

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory.

[More...](#)

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	
L2 Cache			
Reads	151512421	159.488 GB/s	
Writes	95040498	100.043 GB/s	
Total	246552919	259.531 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2969929607	3.126.258 GB/s	
Global Stores	95040000	100.043 GB/s	
Texture Reads	1285537873	5.412.819 GB/s	
Unified Total	4350507480	8,639.119 GB/s	
Device Memory			
Reads	201756700	212.377 GB/s	
Writes	95093954	100.099 GB/s	
Total	296850654	312.476 GB/s	
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	5	5.263 kB/s	

Here we confirm that the baseline kernel is using solely global memory.

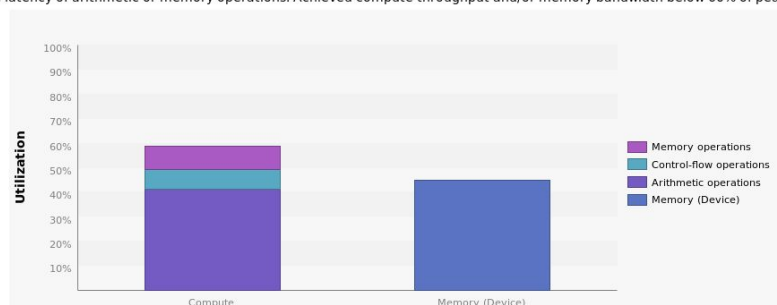
Optimization 1: Parameter Sweeping

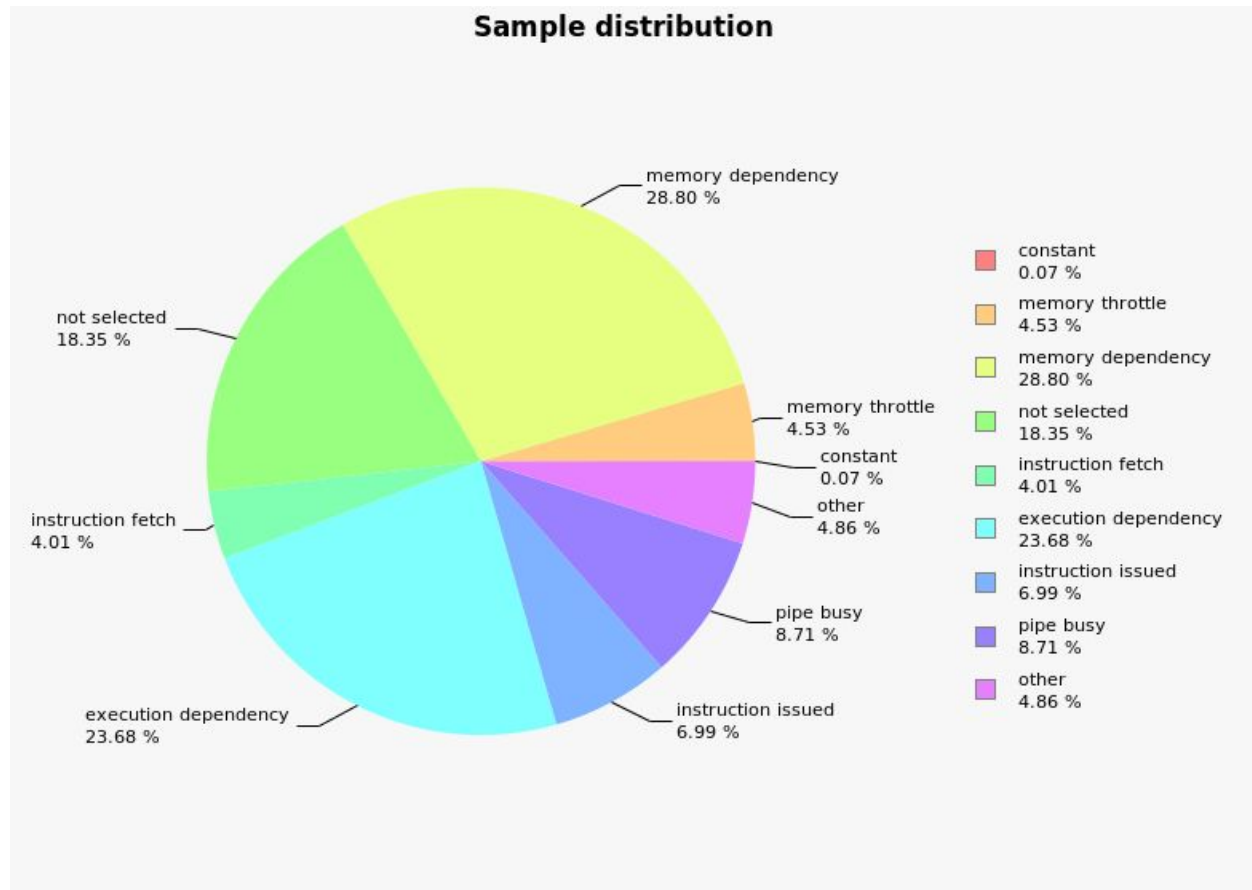
```
New Inference
Op Time: 0.025352
Op Time: 0.138414
Correctness: 0.7653 Model: ece408
5.12user 4.21system 0:05.70elapsed 163%CPU (0avgtext+0avgdata 2973380maxresident)k
0inputs+4568outputs (0major+730206minor)pagefaults 0swaps
```

While it may seem trivial, tuning certain parameters such as the kernel's tile width has the potential to greatly improve the performance of kernels. This lies in the hardware of the streaming multiprocessor that we're using to execute this code. We've initialized our tile width to be that of 24, and because we also declared our block size to be a 2-D block with each width being the tile width, it is important to have our actual block size in the streaming multiprocessor to match what we declared. We realized that using 24 as our tile width is the most optimal, and this makes sense as having 576 threads in a block is divisible by 32, which is the number of threads in a warp. If there were to be more threads than the actual hardware block in SM could handle, then it would have been to be pushed to different blocks, which would lead to inefficiencies that would slow down the computation time dramatically. Same goes for the opposite case where not every thread is actively working due to small tile width. Therefore, although parameter tuning seems simple, it really makes a difference in the computation time.

i Kernel Performance is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.





After tuning our kernel's TILE_WIDTH size, we found that the most optimal size for the first pass of the kernel is 24, and we can see that we have reduced our kernel's memory dependency by almost half, from 57.61% to 28.7%. This optimization gave us an increase in performance for the first pass, reducing the first Op Time from 0.030720 to 0.025352, which is roughly a 16.67% improvement. However, the second pass performs worse, which we believe is because the TILE_WIDTH of 24 is not the optimal tile width value for the second layer. While we did not notice this at first during Milestone 4, we reran NVPROF on each of our first three optimizations and discovered this difference in improvement to the different passes, which ultimately became one of the reasons that lead us to

considering implementing different kernel implementations as one of our optimizations for the 5th milestone 6th optimization.

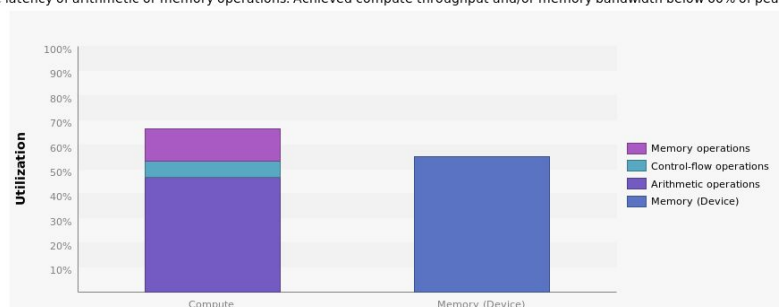
Optimization 2: Tuning w/Restrict & Loop Unrolling (Optimization 1 included):

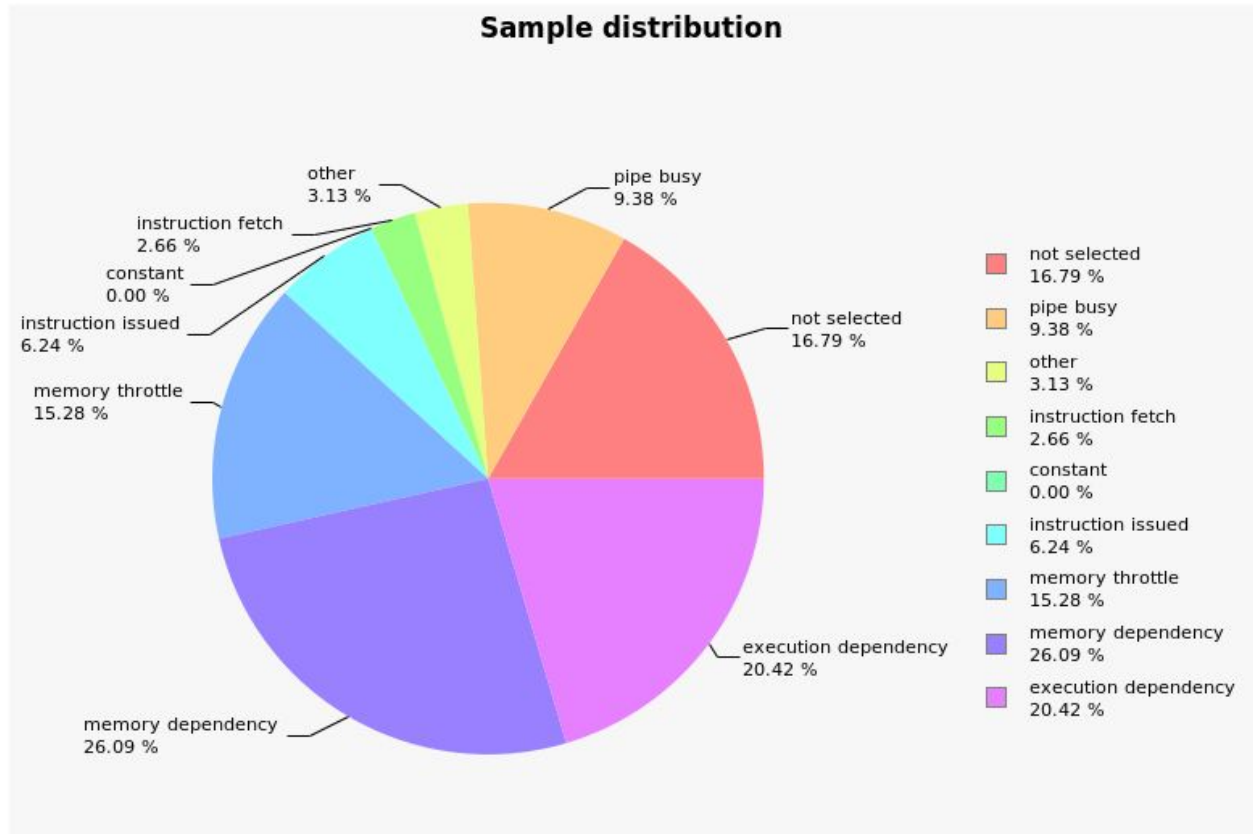
```
New Inference
Op Time: 0.019568
Op Time: 0.116580
Correctness: 0.7653 Model: ece408
5.46user 3.38system 0:05.21elapsed 169%CPU (0avgtext+0avgdata 2954808maxresident)k
0inputs+4560outputs (0major+733143minor)pagefaults 0swaps
```

We noticed that our kernel after our first optimization (parameter sweeping) had a fairly high execution dependency (an increase from the baseline), which can be caused by computational overhead. So, we sought to use loop unrolling and restricts to reduce the delays caused by some computational overhead. Our rationale for this is because loop unrolling essentially removes the overhead that is inherent with a loop, which can speed up some delay with accessing memory. We also used restricts on pointer variables because restricts allow the compiler to not repeatedly check pointer accesses, as restricted pointers should never share the same memory location with another pointer. Since the compiler doesn't have to repeatedly check the pointer accesses, this also reduces some overhead in our kernel.

i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.





We can see that both our execution and memory dependency have decreased after adding the loop unrolling and restrict optimizations to our initial parameter sweeping optimization. This makes sense, as we placed our loop unroll on a loop that was iterating memory accesses and floating point operations, which reduced some delay caused by overhead in both the memory accesses computation. The Op Times above also reflect this improvement, as both forward passes experience a performance increase. The first pass went from 0.025352 to 0.019568, nearly a 25% increase in speed, while the second pass went from 0.138414 to 0.116580.

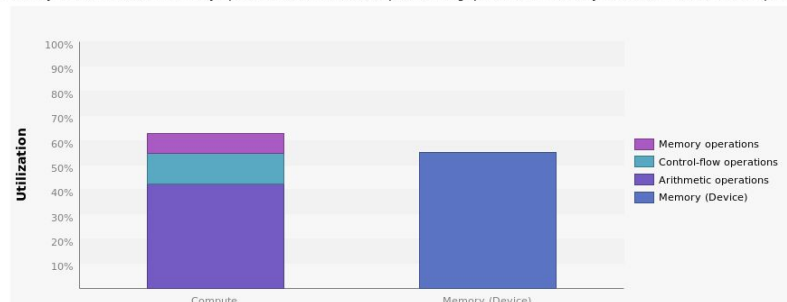
Optimization 3: Weight Matrix in Constant Memory (Optimizations 1+2 included):

```
New Inference
Op Time: 0.019024
Op Time: 0.096173
Correctness: 0.7653 Model: ece408
4.90user 3.48system 0:04.85elapsed 172%CPU (0avgtext+0avgdata 2972104maxresident)k
0inputs+4688outputs (0major+733393minor)pagefaults 0swaps
```

For our third optimization, we sought to store the weight matrix in constant memory in order to further reduce dependencies caused by memory latency. Since the weight matrix is fairly small in size compared to the input matrix, and the weight matrix values experience significant reuse, it made sense to store them in constant memory, as it is the fastest memory type in terms of memory accesses and is still large enough to store the entire matrix.

i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



We can see in the Op Times above that constant memory did not improve our performances by much, especially in the first forward pass. This is because the dataset of the first pass is too small to really benefit from the speed constant memory provides for reused values. We can see a larger increase in performance in the second pass as that dataset is much larger and reuses the weight matrix a lot more. The bar graph above shows that our compute utilization is lower than that of

the second optimization, as we introduce some overhead with loading the weight matrix from global memory to constant memory.

Optimization 4: Shared Memory Convolution (Optimizations 1+2+3 included):

```
New Inference
Op Time: 0.038274
Op Time: 0.087122
Correctness: 0.7653 Model: ece408
5.61user 3.05system 0:05.43elapsed 159%CPU (0avgtext+0avgdata 2949840maxresident)k
0inputs+4696outputs (0major+731086minor)pagefaults 0swaps
```

For our fourth optimization, we decided to use shared memory convolution, as the process of convolution involves reusing the same values repeatedly. Instead of constantly accessing global memory to retrieve a value we've used before, we can instead use the quicker shared memory to store our heavily reused values, thereby reducing memory latency. However, as seen above, our first forward pass's Op Time increased compared to the kernel with our earlier optimizations. For the same reason as constant memory, we believe that this is due to the first dataset being too small to benefit from the use of shared memory, and that the overhead introduced from loading global to shared memory actually made the first forward pass's performance worse. The second forward pass did improve however, as its dataset is large enough to benefit from shared memory.

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. [More...](#)

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	1223126072	4,217.555 GB/s	
Shared Stores	91488684	315.469 GB/s	
Shared Total	1314614756	4,533.025 GB/s	
L2 Cache			
Reads	195750542	168.746 GB/s	
Writes	95040022	81.929 GB/s	
Total	290790564	250.674 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	278529253	240.105 GB/s	
Global Stores	95040000	81.929 GB/s	
Texture Reads	777753791	2,681.833 GB/s	
Unified Total	1151323044	3,003.866 GB/s	
Device Memory			
Reads	255710704	220.434 GB/s	
Writes	95103121	81.983 GB/s	
Total	350813825	302.417 GB/s	
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	5	4.31 kB/s	

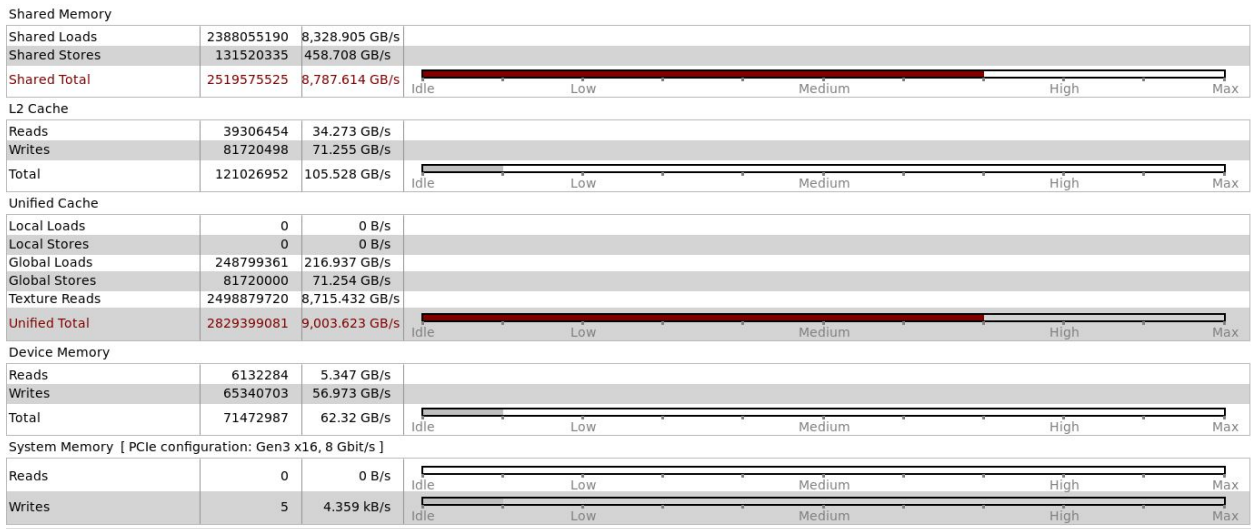
While there is a moderate amount of shared memory usage, it did not provide much of a performance increase. For our final convolutional kernel, we decided to abandon shared memory convolution as explained later in this report.

Optimization 5: Kernel Fusion for Unrolling & Matrix Multiply (Optimization 1+2 included):

```
New Inference
Op Time: 0.040524
Op Time: 0.054739
Correctness: 0.7653 Model: ece408
5.13user 4.13system 0:05.72elapsed 161%CPU (0avgtext+0avgdata 2983552maxresident)k
0inputs+4568outputs (0major+732288minor)pagefaults 0swaps
```

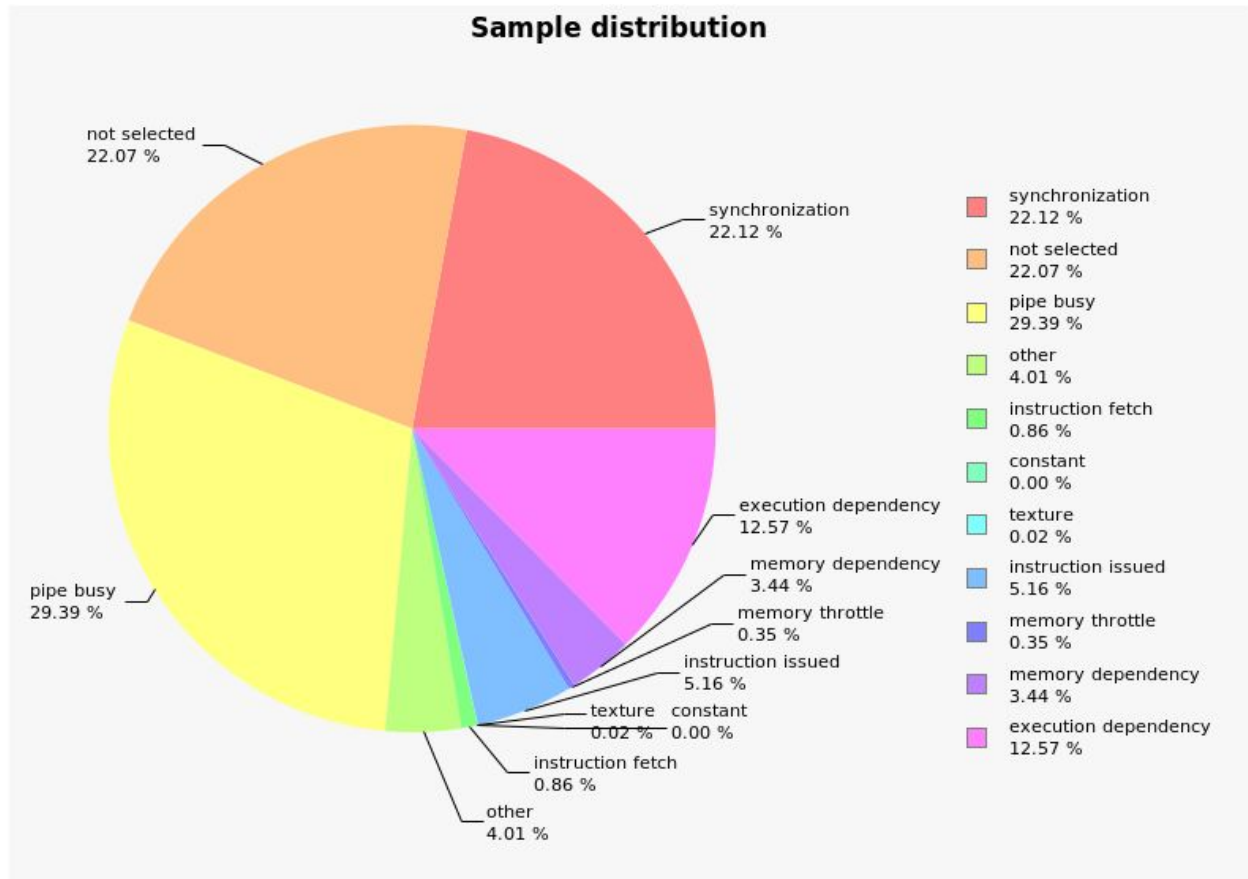
As seen from above, we were able to shorten our time by using shared memory and unrolling/matrix multiply optimization. Kernel fusion allowed us to convert 2 of our kernels into 1, allowing us to reuse the data and prevent the transfers of memory into 2 separate kernels. By incorporating the GEMM matrix-matrix multiplication, we were able to realign our inputs to have each row containing all

input values necessary for every output value, thus giving us regular patterns of memory access throughout the computation and also reducing the forward operation of the convolution layer that was needed.



As seen in the NVPROF, we used shared memory to access our elements. This is because we believed by using shared memory implementation for this optimization, we would be able to access the elements within GEMM matrix that has its inputs already expanded access the memory quicker. Overall, we are able to see some dramatic increase in our optimization time, added with different optimizations that we have already implemented.

Because of this heavy utilization of shared memory, our memory latency has also decreased by a dramatic amount, as seen in the figure below. This decrease greatly increased our Op Time performance for the second pass.



Optimization 6: Multiple Kernel Implementations for Different Layer Sizes

(Optimizations 1+2+3+5 included):

```
New Inference
Op Time: 0.019138
Op Time: 0.049319
Correctness: 0.7653 Model: ece408
5.18user 2.96system 0:04.90elapsed 166%CPU (0avgtext+0avgdata 2955824maxresident)k
0inputs+4784outputs (0major+731969minor)pagefaults 0swaps
```

Since we were given two data inputs to test our convolutions of different optimizations. We decided to split the input data into multiple kernel implementations based on their size, as we know that the bigger that matrix is, the more elements that will be accessed throughout the code.

Thus, this will allow more efficient use of shared memory if there are more data elements. However, with small data input, loading the data from global to shared actually take up more time than accessing it directly.

We've noticed that our 5th optimization performed quite well in the second pass, but horribly in the first. We also noticed that a kernel with a combination of our first three optimizations, the first pass performed well, but the second one didn't work as well. All of this has combined us to think that the reason why there was such a difference in the performance between the two dataset was due to its size. With smaller data input, the loading of the data from the global memory would have more time toll as we're accessing elements less than the one with the bigger data.

By splitting up the data into two different sizes, smaller one getting access directly from global memory and constant while bigger one getting from shared memory (including the load from global to shared), we were able to see dramatic increase in our optime.

Further Insights

We realized that having more optimizations don't necessarily translate to having a good optime. Although we tried to implement all 6 optimizations we came up with, and tried to combine them all together to get our best optime, it didn't give us the best result. With studying our code and understanding how memory works, we realized that the memory latency that occurs from fetching the data globally, and loading it to shared memory, takes way longer than using it directly from global memory. If we were given more elements, and each of these given input matrices are bigger, than maybe our optimizations could have worked better as we would be utilizing the elements we loaded in the shared memory more effectively. Overall, we were able to finesse which optimizations worked the best, and which ones to abandon.

Division of Labor (All Optimizations)

The division of labor was evenly distributed. We all worked on each individual optimization as a group so that it made the understanding of each optimization significantly easier. There were

times when we sometimes split up to handle different optimizations if one was giving trouble. Sometimes Daniel and Jesse would finish a particular optimization while Pranith looked at the next one to get a good idea of what to do. Any questions we had could be tackled together as opposed to having one person struggle while the others do not know what to do. This let us bounce ideas for different implementations properly. The report was also handled as team by using Google Drive. We were all able to edit and do different sections of the report in parallel, saving even more time.

References (All Optimizations)

- Chapter 16 of ECE 408 Textbook
- Lecture Slides for ECE 408
- TAs during office hours answered more conceptual questions
- Past Exam: Fall 2018 Question 5c

Suggestions for Next Year

Please require less optimizations - 4 is good enough. There is no need to do a full 6.