Milestone 4 Report

Name: Ben Chang **NetID**: Sc21 **UIN:** 662284074
Name: Kristopher Koepcke **NetID:** koepcke2 **UIN:** 661617573
Name: Luke Smith **NetID**: lukeis2 **UIN:** 672150982

Group: Ooga_booga

The work was distributed on a per optimization basis. We came together as a group to decide what optimizations we were going to implement then assigned each member an optimization. This is with exception to optimization 5 which Kris did because his optimization 4 did not affect performance significantly.

Optimization 4

Kernel Fusion

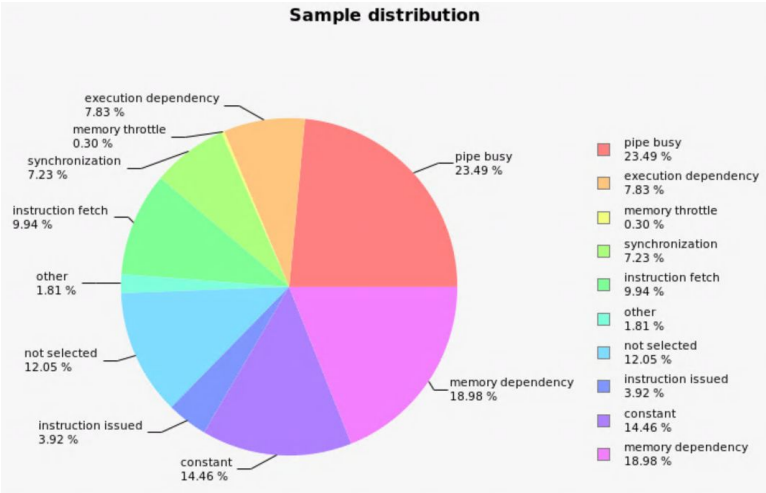
Optimization 4 was to take optimization 3, matrix multiplication with unrolling, and to combine the unroll kernel and matrix multiply kernel together into a single kernel. This would reduce overhead by creating a single data transfer from host to device instead of two data transfers. We identified this optimization after reviewing the code to optimization 3 and figuring that one kernel is better than two.

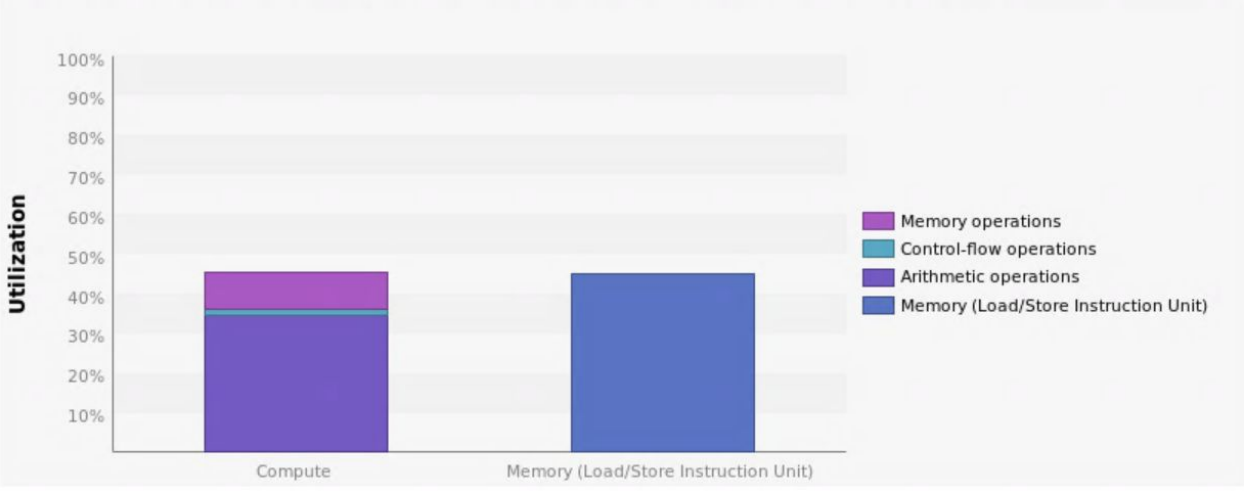
mxnet::op::forward_kernel(float*, float*, float*, int, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	5.11008 s (5,110,077,973 ns)
End	5.11009 s (5,110,087,390 ns)
Duration	9.417 µs
Stream	Default
Grid Size	[154,1,1]
Block Size	[32,32,1]
Registers/Thread	31
Shared Memory/Block	8 KiB
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	78.6%
Global Store Efficiency	88.8%
Shared Efficiency	⚠ 70%
Warp Execution Efficiency	100%
Not-Predicated-Off Warp Executi	91.9%
▾ Occupancy	
Achieved	83.5%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Compared to optimization 3, the general statistics showed here show very little change. Notably, the occupancy achieved has decreased by 0.6%, Not-Predicated-Off warp Execution decreased by 0.2%, and Global load efficiency decreased by 0.3%.

Below is the stall reasons, pipe busy is the largest, then memory dependency then not selected. This is actually worse than optimization 3 because the pipe busy is much lower and the memory dependency has tripled.





Here we see that the compute utilization is slightly lower than in optimization 3 while the memory utilization is the same.

This optimization was therefore not fruitful by giving almost the same metrics with a few slightly lowered than before.

Optimization 5

Remove successive kernel calls from host code

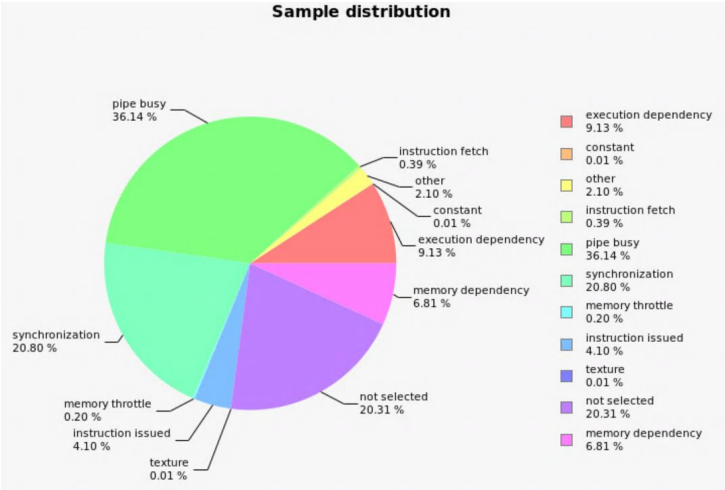
Optimization 5 was to take optimization 4 and remove the for loop that continuously called the kernel code and remove all additional functions by combining everything into one GPU kernel. This optimization was identified after realizing that using a for loop to launch the kernel more than once increases the overhead by sending data between the host and device each time the kernel is called. So to reduce this, remove the for loop and only call the kernel once.

mxnet::op::forward_kernel(float*, f...

Queued	n/a
Submitted	n/a
Start	3.56998 s (3,569,975,855 ns)
End	3.60694 s (3,606,936,187 ns)
Duration	36.96033 ms (36,960,332 ns)
Stream	Default
Grid Size	[137,1,10000]
Block Size	[32,32,1]
Registers/Thread	30
Shared Memory/Block	8 KiB
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	78.7%
Global Store Efficiency	88.8%
Shared Efficiency	⚠ 70%
Warp Execution Efficiency	99.8%
Not-Predicated-Off Warp Execution Efficiency	91.2%
▾ Occupancy	
Achieved	94%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

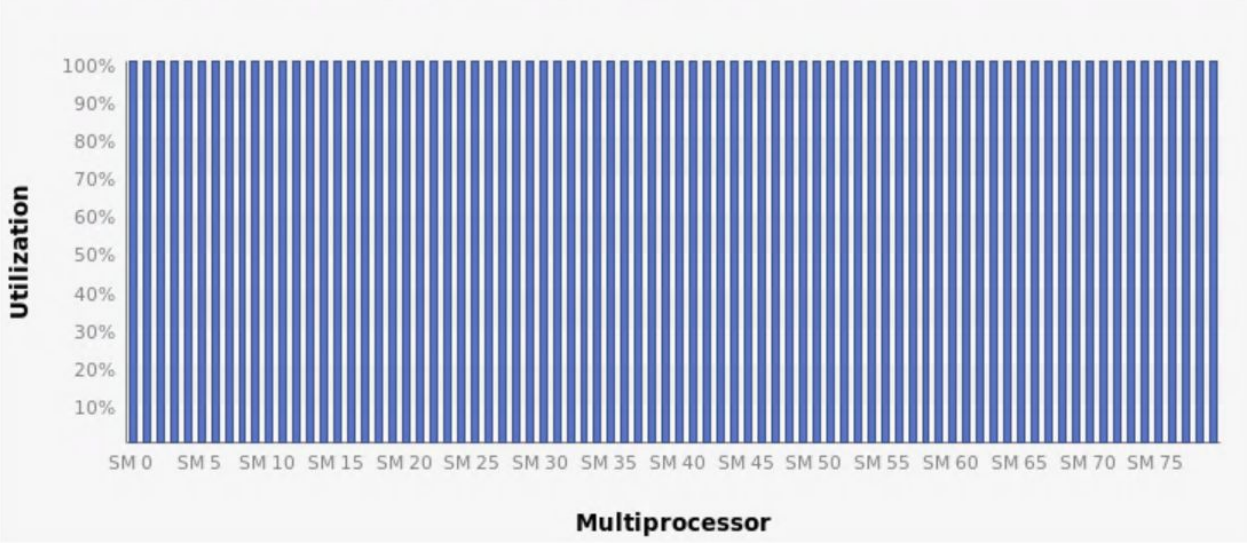
Looking at the general statistics for this optimization, we see mostly the same results with exception to the achieved occupancy which increased by 10.5%

Below we see the lag reasons in the sample distribution pie chart. The pipe busy is back at over 35% and memory dependency back at below 10%. A great improvement over optimization 4, but at similar levels to optimization 3 but now with synchronization issues at over 20%.



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	2105322307	7,291.094 GB/s	
Shared Stores	88121468	305.18 GB/s	
Shared Total	2193443775	7,596.274 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	36860830	31.914 GB/s	
Writes	73560498	63.688 GB/s	
Total	110421328	95.602 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	238274000	206.296 GB/s	
Global Stores	73560000	63.688 GB/s	
Texture Reads	2246794975	7,781.038 GB/s	
Unified Total	2558628975	8,051.022 GB/s	Idle Low Medium High Max
Device Memory			
Reads	6132982	5.31 GB/s	
Writes	65339386	56.57 GB/s	
Total	71472368	61.88 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	4.328 kB/s	

Here we see that the memory utilization has greatly improved over optimization 4 and optimization 3, as was expected from reducing the amount of memory movement between device and host. We see that that shared memory and unified cache utilization is ~2.25 times as high as optimization 4 and optimization 3.



We also see that multiprocessor divergence is non-existent, another improvement over optimization 3.

This concludes that optimization 5 was a vast improvement over optimization 4 and optimization 3.

Optimization 6

Sweeping various parameters to find best values

For this optimization, we took the code from Optimization 5, and scanned various block sizes in order to find the best TILE_SIZE for the code. After trying various parameters, we found out that the best sizes for the two inputs are 16 and 24. If the output layers is 12, then 16 has the better run time. If the output layer is 24, then 24 has the best run time. Given this information. TILE_WIDTH of 24 gives us a more drastic change in runtime, so that is the value that we used.

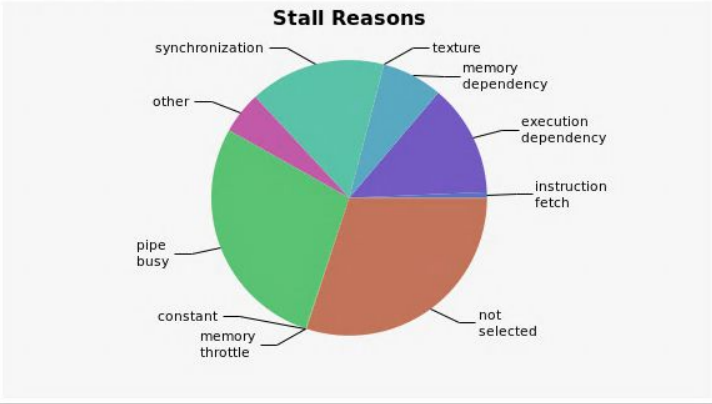
mxnet::op::forward_kernel(float*, float*, float*, int, int, int,...	
Queued	n/a
Submitted	n/a
Start	3.44286 s (3,442,8
End	3.48032 s (3,480,3
Duration	37.45974 ms (37,4
Stream	Default
Grid Size	[182,1,10000]
Block Size	[24,24,1]
Registers/Thread	30
Shared Memory/Block	4.5 KiB
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	⚠ 70.9%
Global Store Efficiency	80%
Shared Efficiency	⚠ 55.9%
Warp Execution Efficiency	99.5%
Not-Predicated-Off Warp Execution Efficiency	90.6%
▾ Occupancy	
Achieved	80.6%
Theoretical	84.4%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

mxnet::op::forward_kernel(float*, float*, float*, int, int, int,..	
Queued	n/a
Submitted	n/a
Start	3.58941 s (3,589,
End	3.63711 s (3,637,
Duration	47.69311 ms (47,
Stream	Default
Grid Size	[36,1,10000]
Block Size	[24,24,1]
Registers/Thread	30
Shared Memory/Block	4.5 KiB
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	76%
Global Store Efficiency	⚠ 68.2%
Shared Efficiency	⚠ 56.4%
Warp Execution Efficiency	98.6%
Not-Predicated-Off Warp Execution Efficiency	90.5%
▾ Occupancy	
Achieved	83.8%
Theoretical	84.4%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

12 output channel

24 output channel

For this report I will be focusing on the kernel called upon the 24 output channel kernel.



From the stall reasons chart to the left, we can see that the biggest chunk of the stall is due to the pipe being busy. This is a great improvement on the base code which had mostly memory dependency as the reason for stall. Not selected was also a big reason for stall, which means that warps were

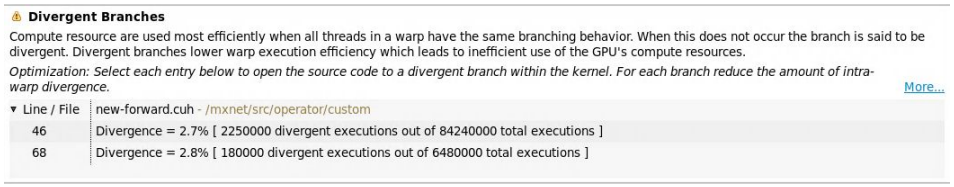
ready to be issued, but a different warp was issued instead. This means that it would be more effective if we were able to have more warps on the SM than what we currently do.
From the image below, we can see that there are 30 registers per thread, 18432 registers per block. This means that only a max of 3 blocks can be utilized at once. This also means that there are only 54 warps which can be run simultaneously. Improving this would decrease the time used by Not Selected

Occupancy Per SM			
Active Blocks		3	32
Active Warps	53.61	54	64
Active Threads		1728	2048
Occupancy	83.8%	84.4%	100%
Warps			
Threads/Block		576	1024
Warps/Block		18	32
Block Limit		3	32
Registers			
Registers/Thread		30	65536
Registers/Block		18432	65536
Block Limit		3	32

From the utilization we can see that the kernel is bound by both Memory and compute utilization. Most of the compute is used by arithmetic operations, which is good, and the memory is mostly limited by shared memory usage. This also indicates that there is a lot less global memory bandwidth, as we are mostly using shared memory in order to bypass global memory's high access times.

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	3042129685	8,164.546 GB/s	
Shared Stores	170316877	457.101 GB/s	
Shared Total	3212446562	8,621.646 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	89790222	60.245 GB/s	
Writes	36990504	24.819 GB/s	
Total	126780726	85.064 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	758219659	508.732 GB/s	
Global Stores	36990000	24.819 GB/s	
Texture Reads	3267961643	8,770.639 GB/s	
Unified Total	4063171302	9,304.19 GB/s	Idle Low Medium High Max
Device Memory			
Reads	16353974	10.973 GB/s	
Writes	25238132	16.934 GB/s	
Total	41592106	27.906 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	3.354 kB/s	Idle Low Medium High Max

The memory utilization tab shows us more. The shared Memory usage is pretty high, while there are a lot less global loads and stores. This indications that the kernel is using shared memory more efficiently compared the the slow global memory. This chart also indicates that shared memory access could be better. Better access patterns could definitely increase shared memory bandwidth further.

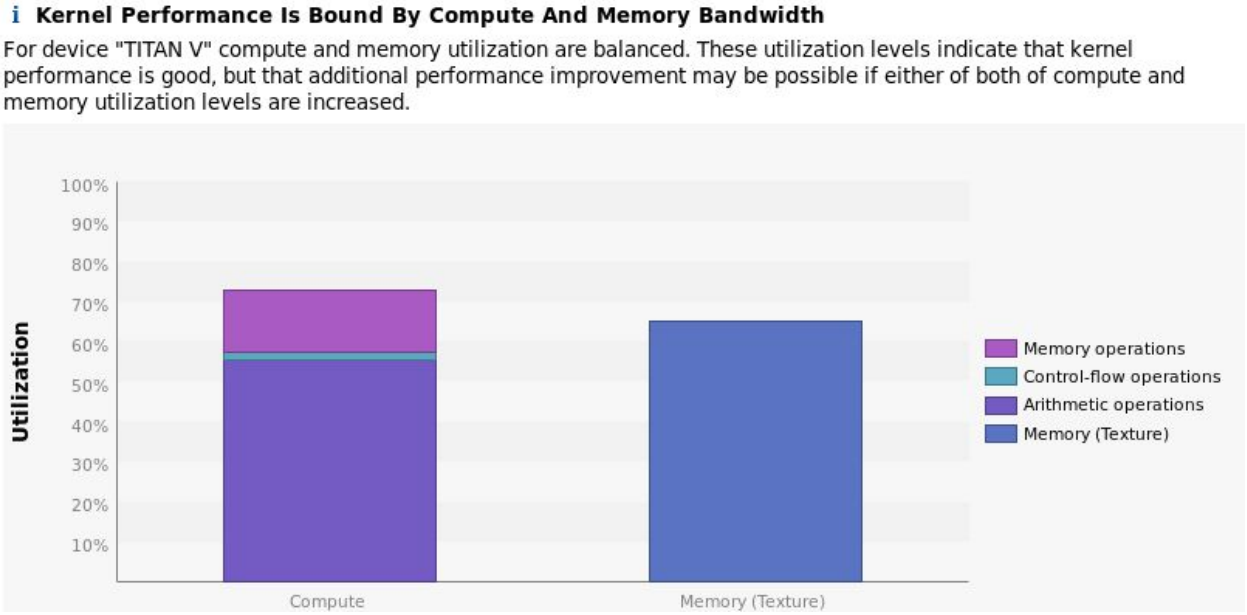


The low percentage of divergence is also a good sign, as less divergence means less wasted resources. So why did we choose the optimization, and why did we choose to use a TILE_WIDTH of 24? Well the output feature maps are of size 12 and 24. If we choose a TILE_WIDTH of anything larger than 24, many threads will not be used, leading to increased divergence. Smaller values would lead to less efficient use of shared memory usage, which would also lead to decreased performance. A size of 24 also leads to blocks with dimensions 24 * 24, which is 576 threads. This number is divisible by 32, and as such is made up of entire warps. This helps to limit divergence, thus increasing performance even more. Using a TILE_WIDTH of 16 greatly increases the speed of the 12 output layer kernel, but reduces the speed of the 24 output layer kernel. This issue could be addressed by having two different kernels, depending on how many output layers there are.

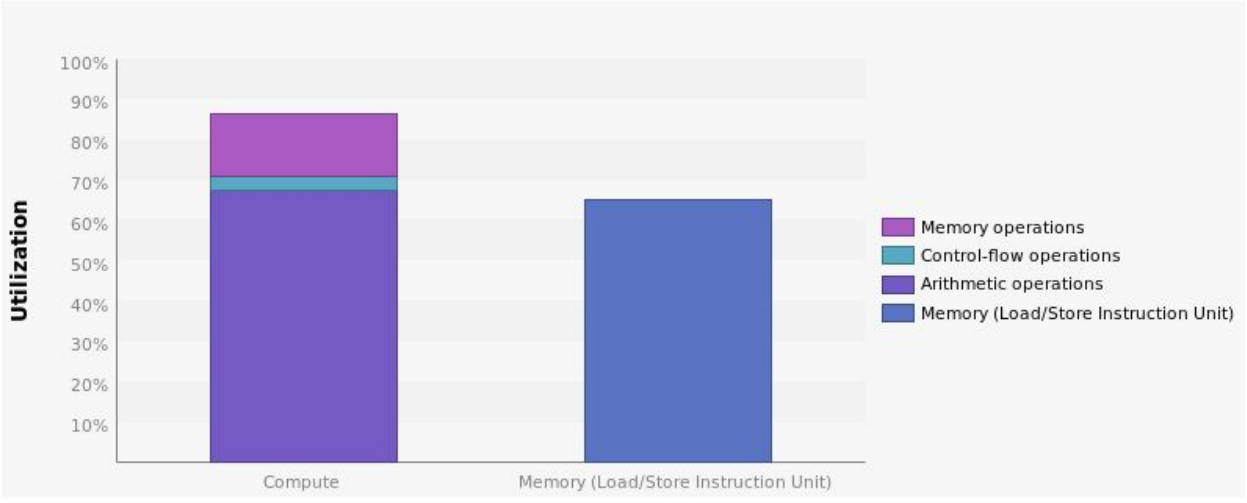
Optimization 7 (Extra Credit)

In optimization 12 we used two separate kernels to handle inputs of varying sizes. This optimization builds off of optimization 6 where we tried varying input sizes but with one kernel, so many of the findings apply here as well, therefore we will cover the overall results of running two separate kernels. Forward24 is called when M = 24 which creates shared memory allocations with block sizes of 24. When M = 12 forward12 is called and share memory allocations are created with block sizes of 16. This results in the overall utilization staying fairly high throughout the entire run as evidenced by the following charts taken from nvvp.

Forward24 utilization



Forward12 utilization



Looking at the properties of each kernel, we can see that the runtime of the actual kernels is very fast at only 1.88ms for forward 12 and 4.84ms for forward 24. This averages out to 3.36 ms which is out best time yet given the input size.

Forward 12 properties

mxnet::op::forward_kernel12(float*, float*, float*, int, int, int, int, int

Queued	n/a
Submitted	n/a
Start	3.52543 s (3,525,426,439 ns)
End	3.52731 s (3,527,314,549 ns)
Duration	1.88811 ms (1,888,110 ns)
Stream	Default
Grid Size	[273,1,1000]
Block Size	[16,16,1]
Registers/Thread	30
Shared Memory/Block	2 KiB
Launch Type	Normal
▼ Efficiency	
Global Load Efficiency	⚠ 66.9%
Global Store Efficiency	80%
Shared Efficiency	⚠ 41.4%
Warp Execution Efficiency	97.9%
Not-Predicated-Off Warp Execution Efficien	88.4%
▼ Occupancy	
Achieved	95.8%

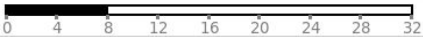
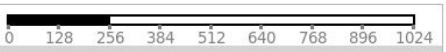
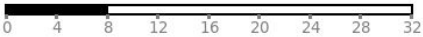
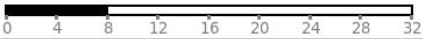
Forward 24 properties

mxnet::op::forward_kernel24(float*, float*, float*, int, int, int, int, int,

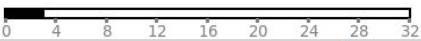
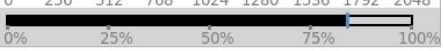
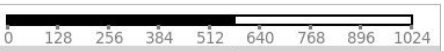
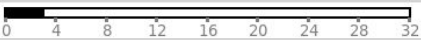
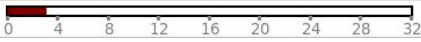
Queued	n/a
Submitted	n/a
Start	3.62952 s (3,629,522,436 ns)
End	3.63436 s (3,634,359,525 ns)
Duration	4.83709 ms (4,837,089 ns)
Stream	Default
Grid Size	[36,1,1000]
Block Size	[24,24,1]
Registers/Thread	30
Shared Memory/Block	4.5 KiB
Launch Type	Normal
▼ Efficiency	
Global Load Efficiency	76%
Global Store Efficiency	⚠ 68.2%
Shared Efficiency	⚠ 56.4%
Warp Execution Efficiency	98.6%
Not-Predicated-Off Warp Execution Efficien	90.5%
▼ Occupancy	
Achieved	83.7%

Additionally we can see that the overall occupancy in each kernel is very high. In forward 12 it is 95.8% while in forward 24 it is 83.7%.

Forward 12 usage

Occupancy Per SM				
Active Blocks		8	32	
Active Warps	61.32	64	64	
Active Threads		2048	2048	
Occupancy	95.8%	100%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
Registers				
Registers/Thread		30	65536	
Registers/Block		8192	65536	
Block Limit		8	32	
Shared Memory				
Shared Memory/Block		2048	98304	
Block Limit		48	32	

Forward 24 usage

Occupancy Per SM				
Active Blocks		3	32	
Active Warps	53.55	54	64	
Active Threads		1728	2048	
Occupancy	83.7%	84.4%	100%	
Warps				
Threads/Block		576	1024	
Warps/Block		18	32	
Block Limit		3	32	
Registers				
Registers/Thread		30	65536	
Registers/Block		18432	65536	
Block Limit		3	32	
Shared Memory				
Shared Memory/Block		4608	98304	
Block Limit		21	32	

In the sample distribution charts we can see the pipe being busy is a large percentage of overall stall reasons as opposed to unoptimized code with large amounts being stalled due to memory reasons. We can also see that the memory dependency section is very small at only 6.74%

Forward24 sample distribution