ECE 408 Report

Team Name: tiledtitans

Members:

- Jackson DeDobbelaere (dedobbe2)
- Matthew Grossfeld (grossfe2)
- Xinbo Wu (xinbowu2)

All on campus students

MILESTONE 1

List of all kernels that collectively consume more than 90% of the program time:

- [CUDA memcpy HtoD]
- void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)
- Volta_cgemm_64x32_tn
- void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
- Volta_sgemm_128x128_tn
- void fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=1, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int)
- void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

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

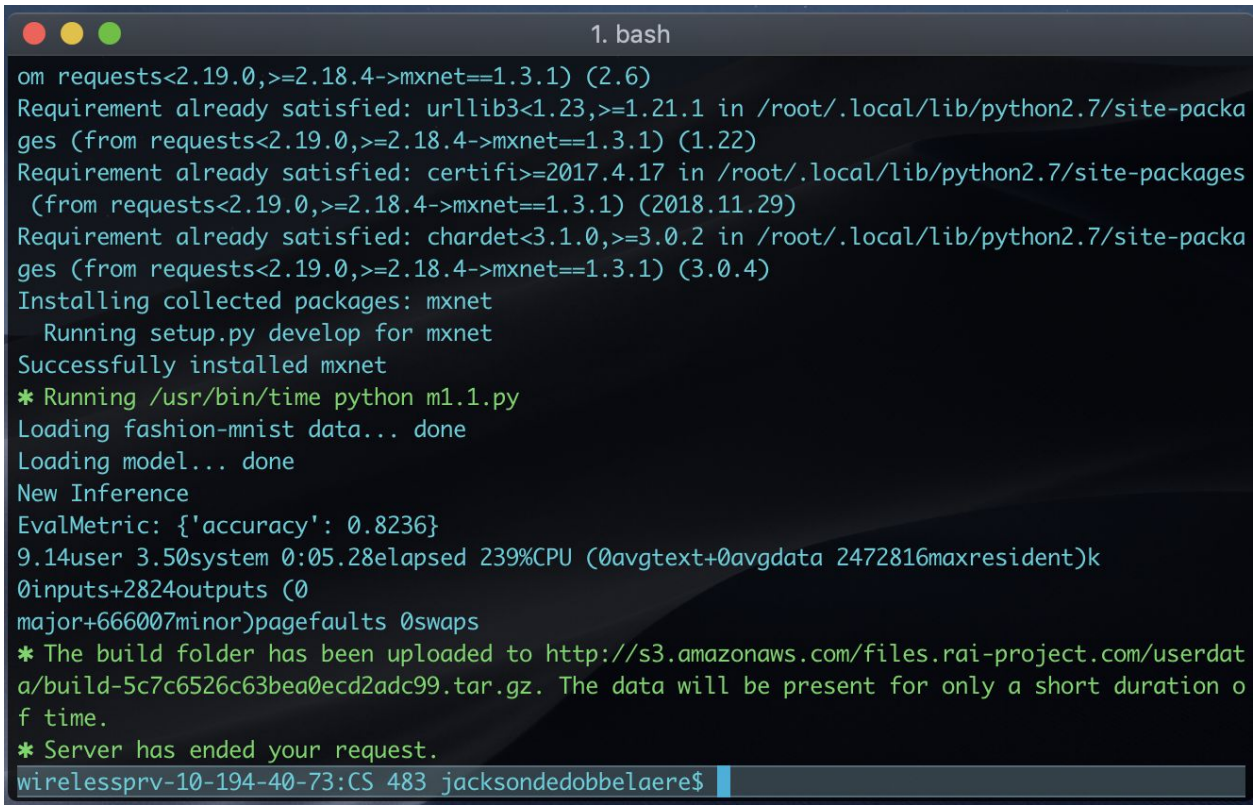
- cudaStreamCreateWithFlags
- cudaMemGetInfo
- cudaFree

Explanation of the difference between kernels and API calls:

API calls are done with the CPU and execute from the time the call is made to the time it is returned. Kernels are ran on the GPU and are measured by the total time that the kernel is executing and running instructions. cudaLaunchKernel is an API function called from the CPU

that can launch kernels from the GPU. The profiling done on the API accounts for all of the launch overhead while the kernel timing only contains a small portion of it.

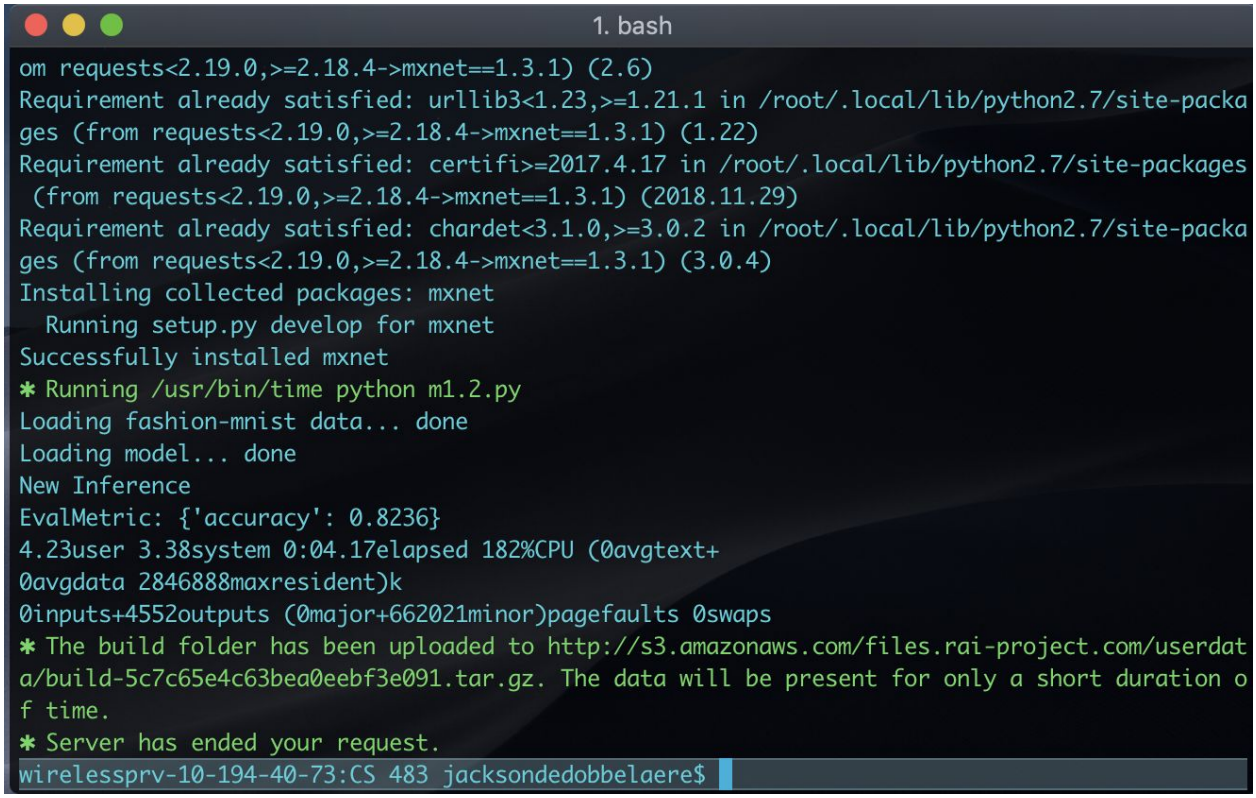
Output of rai running MXNet on the CPU:



Program run time of MXNet on the CPU:

- User: 9.14
- System: 3.50
- Elapsed: 0:05.28

Output of rai running MXNet on the GPU:



Program run time of MXNet on the GPU:

- User: 4.23
- System: 3.38
- Elapsed: 0:04.17

MILESTONE 2

100 Images:

- **Whole Program Execution Time:**
  - User: 2.77
  - System: 2.65
  - Elapsed: 0:01.03
- **Layer 1 Op Time:** 0.034247
- **Layer 2 Op Time:** 0.074316

1,000 Images:

- **Whole Program Execution Time:**
  - User: 4.24
  - System: 3.15
  - Elapsed: 0:01.95
- **Layer 1 Op Time:** 0.238204
- **Layer 2 Op Time:** 0.743339

10,000 Images:

- **Whole Program Execution Time:**
  - User: 14.97
  - System: 4.47
  - Elapsed: 0:11.37
- **Layer 1 Op Time:** 2.427247
- **Layer 2 Op Time:** 7.383249

MILESTONE 3

100 Images:

- **Correctness:** 0.84
- **Whole Program Execution Time:**
    - User: 4.36
    - System: 3.27
    - Elapsed: 0:04.34
- **Layer 1 Op Time:** 0.000102
- **Layer 2 Op Time:** 0.000236

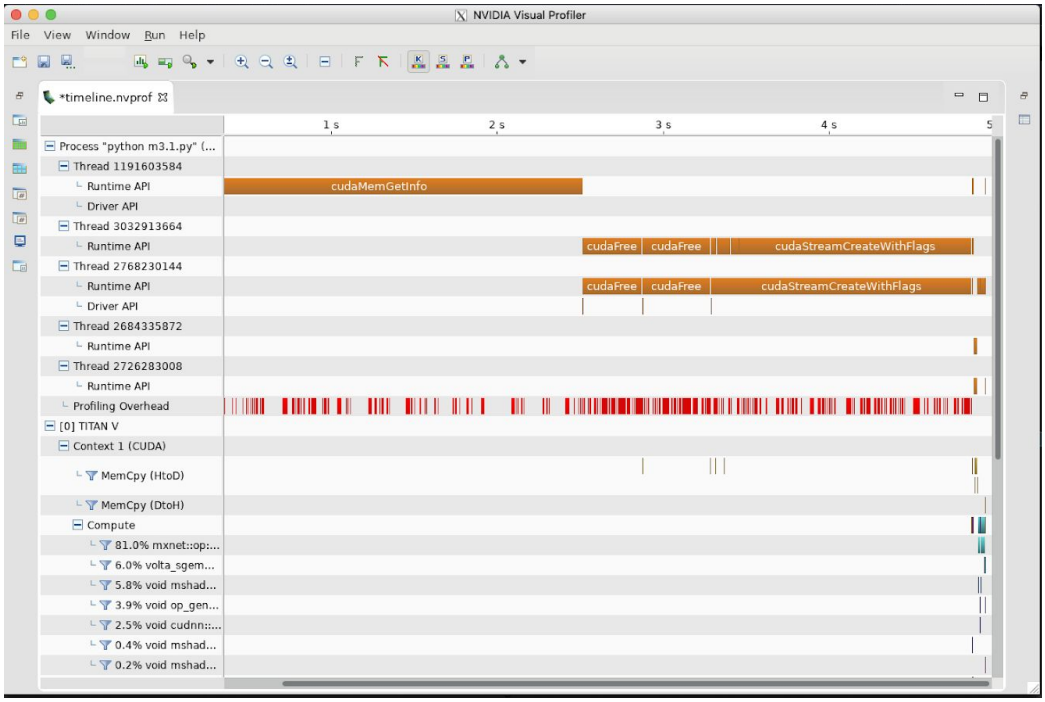
1,000 Images:

- **Correctness:** 0.852
- **Whole Program Execution Time:**
    - User: 4.42
    - System: 3.39
    - Elapsed: 0:04.22
- **Layer 1 Op Time:** 0.000908
- **Layer 2 Op Time:** 0.002467

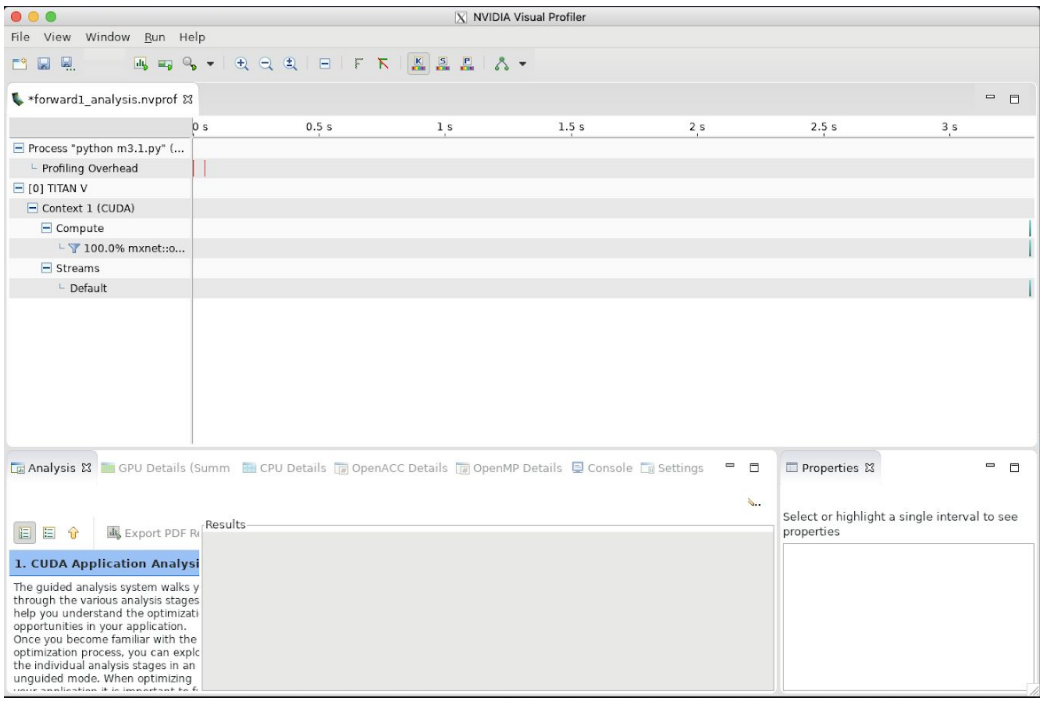
10,000 Images:

- **Correctness:** 0.8397
- **Whole Program Execution Time:**
    - User: 4.33
    - System: 3.31
    - Elapsed: 0:04.34
- **Layer 1 Op Time:** 0.009238
- **Layer 2 Op Time:** 0.024392

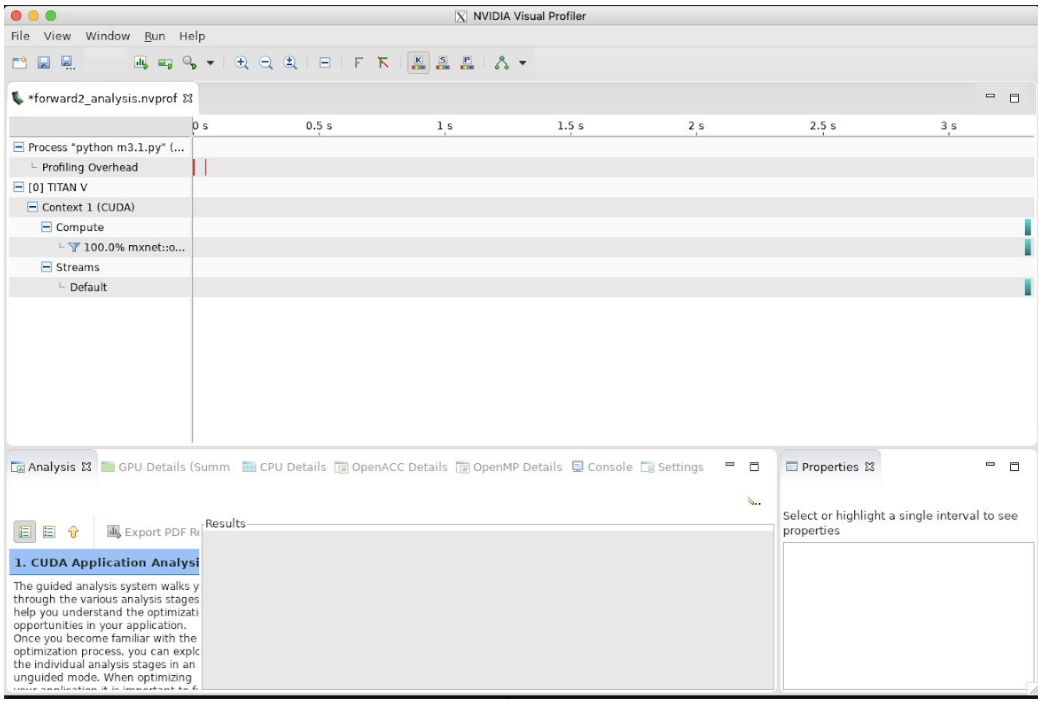
NVVP output of timeline.nvprof:



NVVP output of forward1_analysis.nvprof:



NVVP output of forward2_analysis.nvprof:



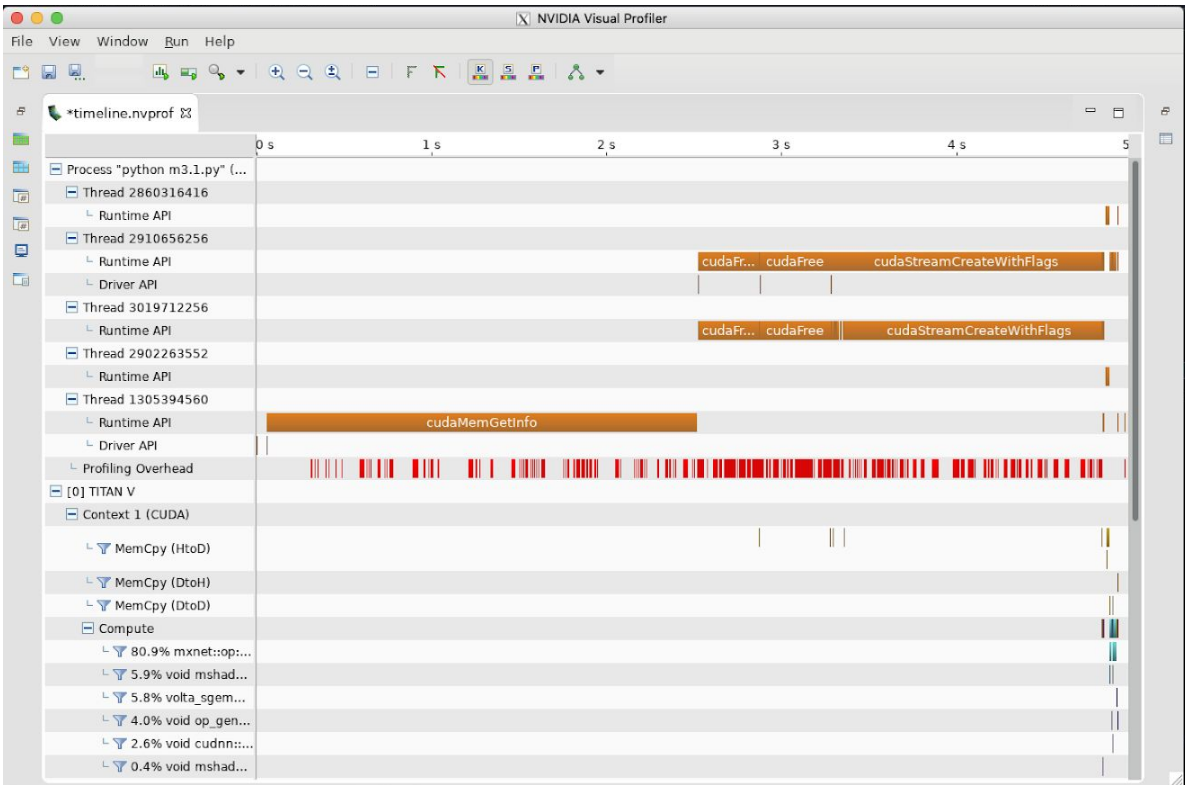
MILESTONE 4

Optimization #1: Kernel Values in Constant Memory

- **Description**

  We create an array in constant memory on our device of size $K \times K \times M \times C$ for storing our weights. This is easily done by adding the CUDA API call *cudaMemcpyToSymb*T^2K^2*ol* before calling our kernel. This allows us to only access our weights from cached memory within the device as opposed to using the global memory. If we need to access all the weights in our kernel, we will have avoided $K \times K \times M \times C$ global memory accesses.

- **Timeline Overview**

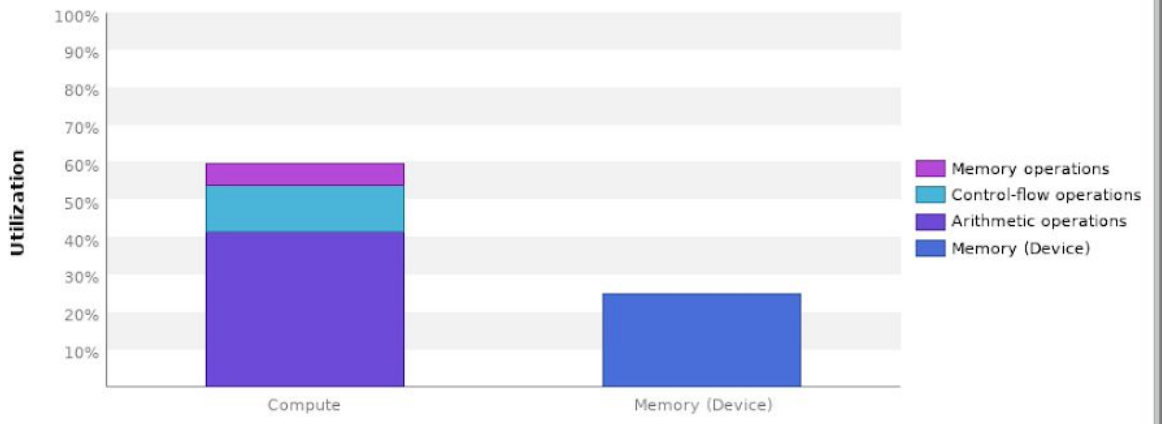


- **Forward Layer 1**

Results

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 o memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



Results

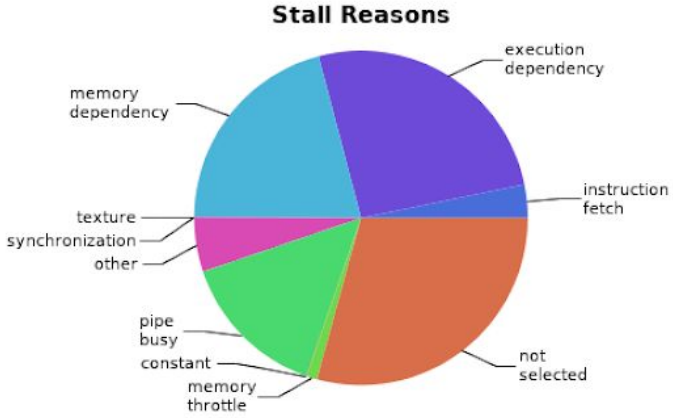
Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.

Constant - A constant load is blocked due to a miss in the constants cache.

Pipeline Busy - The compute resource(s) required by the instruction is not yet available.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Optimization: Resolve the primary stall issue; not selected.



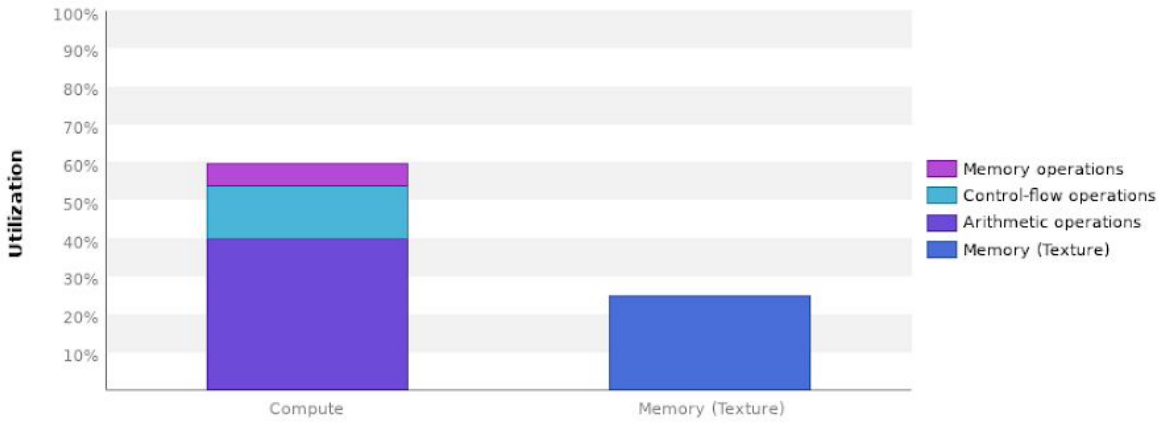
Results

Active Warps	40.15	64	64	0 10 20 30 40 50 6064
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	62.7%	100%	100%	0% 50% 100%
Warps				
Threads/Block		1024	1024	0 256 512 768 1024
Warps/Block		32	32	0 5 10 15 20 25 3032
Block Limit		2	32	0 5 10 15 20 25 3032
Registers				
Registers/Thread		32	65536	0 16384 32768 49152 65536
Registers/Block		32768	65536	0 32k 64k
Block Limit		2	32	0 5 10 15 20 25 3032
Shared Memory				
Shared Memory/Block		0	98304	0 32k 64k 96k
Block Limit		0	32	0 5 10 15 20 25 3032

- **Forward Layer 2**

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



Results

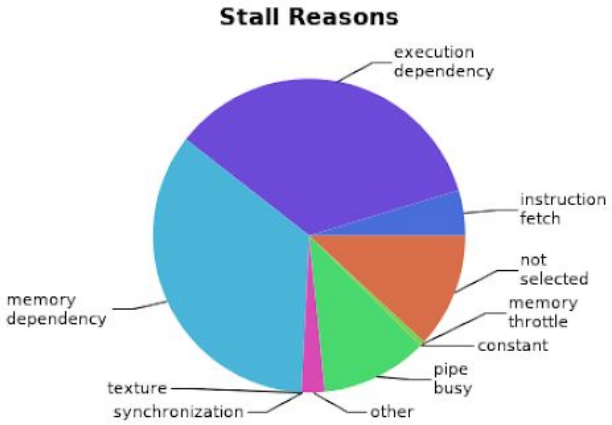
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Synchronization - The warp is blocked at a __syncthreads() call.

Constant - A constant load is blocked due to a miss in the constants cache.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.



Results

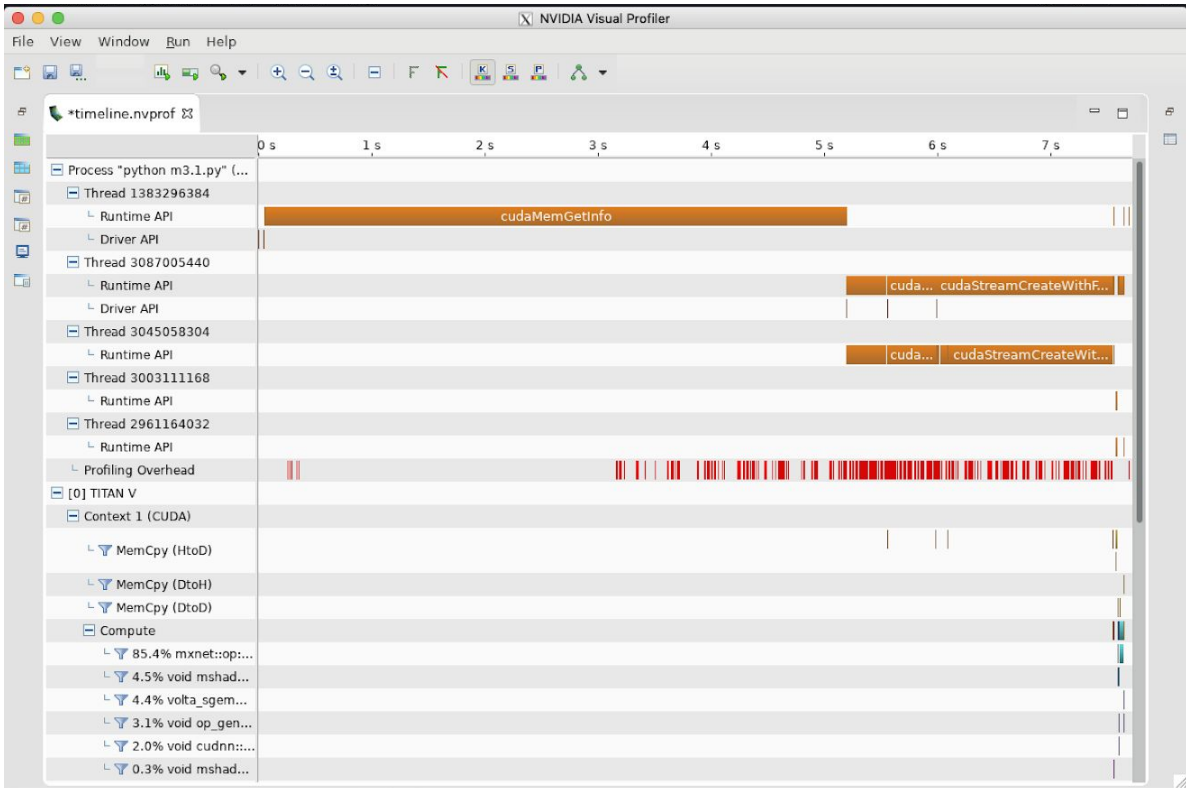
Active Blocks		2	32	0 5 10 15 20 25 3032
Active Warps	28.8	64	64	0 10 20 30 40 50 6064
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	45%	100%	100%	0% 50% 100%
Warps				
Threads/Block		1024	1024	0 256 512 768 1024
Warps/Block		32	32	0 5 10 15 20 25 3032
Block Limit		2	32	0 5 10 15 20 25 3032
Registers				
Registers/Thread		32	65536	0 16384 32768 49152 65536
Registers/Block		32768	65536	0 32k 64k
Block Limit		2	32	0 5 10 15 20 25 3032
Shared Memory				
Shared Memory/Block		0	98304	0 32k 64k 96k
Block Limit		0	32	

Optimization #2: Shared Memory Convolution

- **Description**

  We changed the size of our blocks (i.e. the amount of threads per block) from 32 to 16 so that our tiles are smaller (thus reducing the amount of memory accesses per tile). We perform tiling using strategy 1 from the lecture notes (i.e. our block size covers the output size and we load the input over multiple steps). In our kernel we instantiate an array in shared memory called *X_Shared*, of size (TILE_SIZE + K - 1)^2 that holds our input values for x. This reduces our global memory accesses significantly. More specifically, we have global memory accesses that are converted into shared memory accesses, where T is TILE_SIZE. This gives us a total reduction of $\frac{T^2K^2}{(T+K-1)^2}$.

- **Timeline Overview**

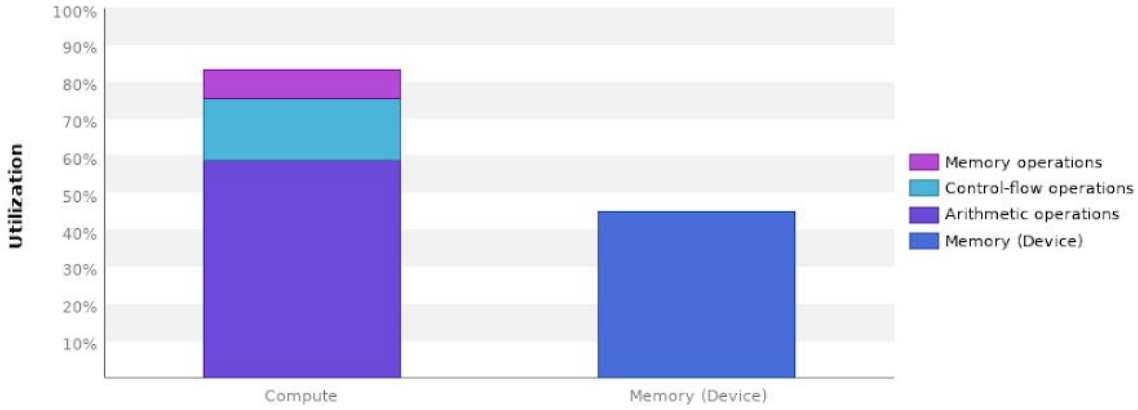


- **Forward Layer 1**

Results

Kernel Performance Is Bound By Compute

For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



Results

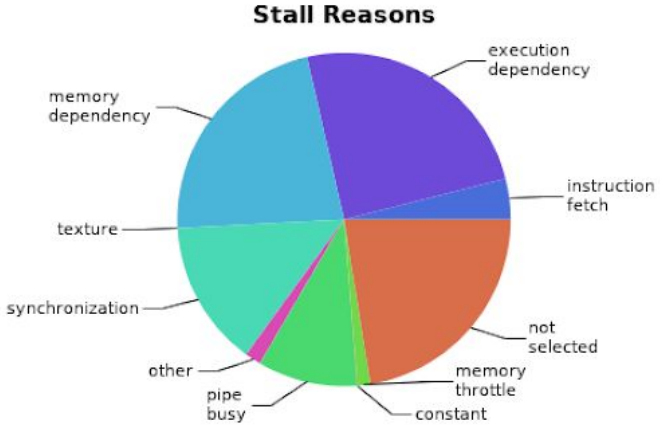
potentially be reduced by increasing instruction-level parallelism.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Synchronization - The warp is blocked at a __syncthreads() call.

Constant - A constant load is blocked due to a miss in the constants cache.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.



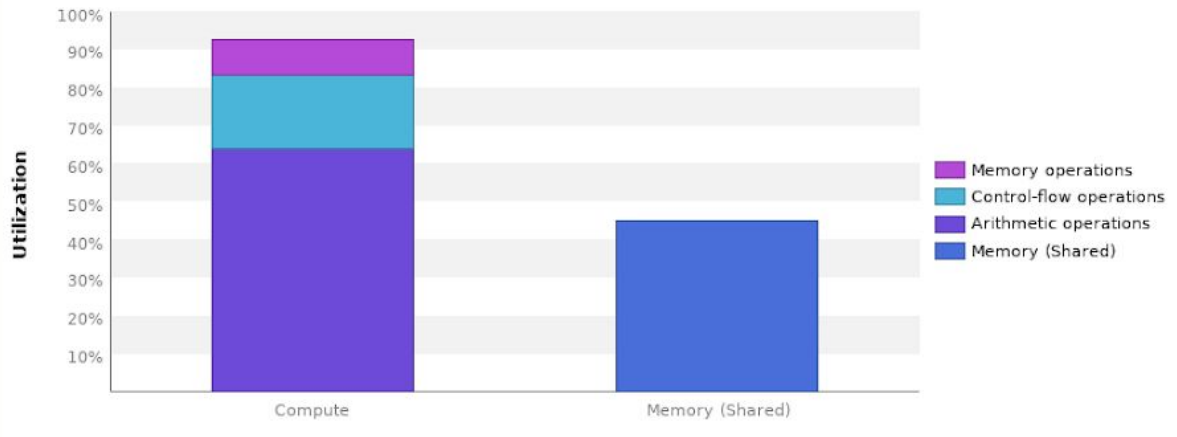
Results

Active Blocks		6	32	0 5 10 15 20 25 3032
Active Warps	45.64	48	64	0 10 20 30 40 50 6064
Active Threads		1536	2048	0 512 1024 1536 2048
Occupancy	71.3%	75%	100%	0% 50% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 5 10 15 20 25 3032
Block Limit		8	32	0 5 10 15 20 25 3032
Registers				
Registers/Thread		39	65536	0 16384 32768 49152 65536
Registers/Block		10240	65536	0 32k 64k
Block Limit		6	32	0 5 10 15 20 25 3032
Shared Memory				
Shared Memory/Block		1700	98304	0 32k 64k 96k
Block Limit		54	32	0 5 10 15 20 25 3032

- **Forward Layer 2**

i Kernel Performance Is Bound By Compute

For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



Results

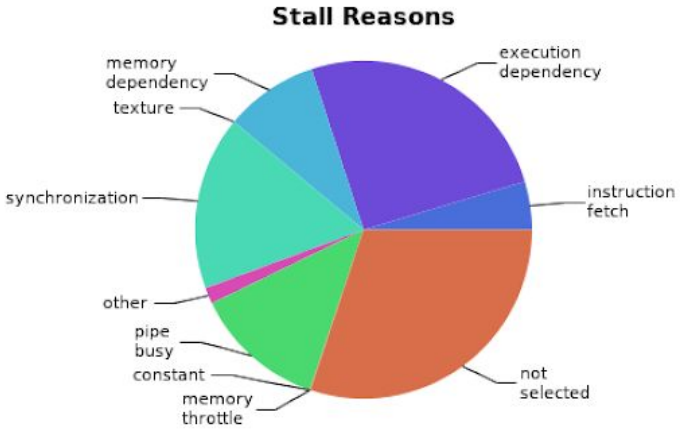
potentially be reduced by increasing instruction-level parallelism.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Synchronization - The warp is blocked at a __syncthreads() call.

Constant - A constant load is blocked due to a miss in the constants cache.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.



Results

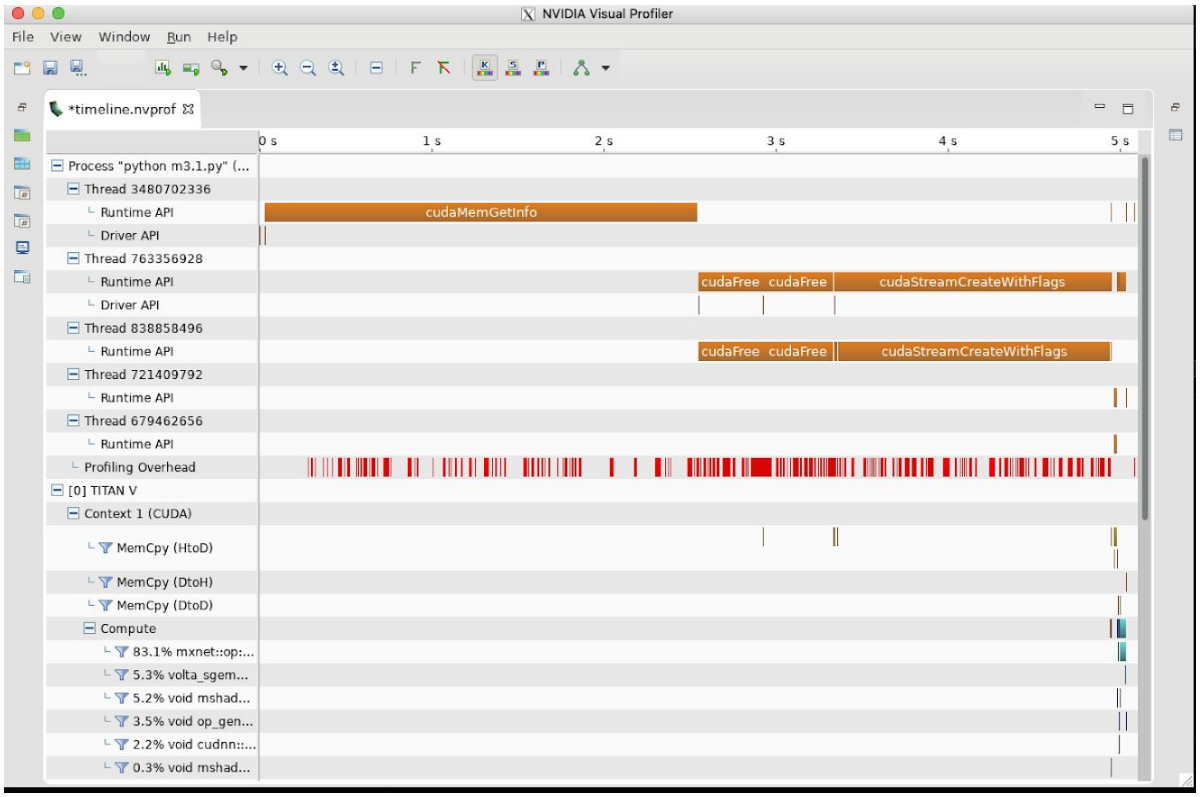
Active Blocks		6	32	0 5 10 15 20 25 3032
Active Warps	47.54	48	64	0 10 20 30 40 50 6064
Active Threads		1536	2048	0 512 1024 1536 2048
Occupancy	74.3%	75%	100%	0% 50% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 5 10 15 20 25 3032
Block Limit		8	32	0 5 10 15 20 25 3032
Registers				
Registers/Thread		39	65536	0 16384 32768 49152 65536
Registers/Block		10240	65536	0 32k 64k
Block Limit		6	32	0 5 10 15 20 25 3032
Shared Memory				
Shared Memory/Block		1700	98304	0 32k 64k 96k
Block Limit		54	32	0 5 10 15 20 25 3032

Optimization #3: Tuning with Restrict and Loop Unrolling

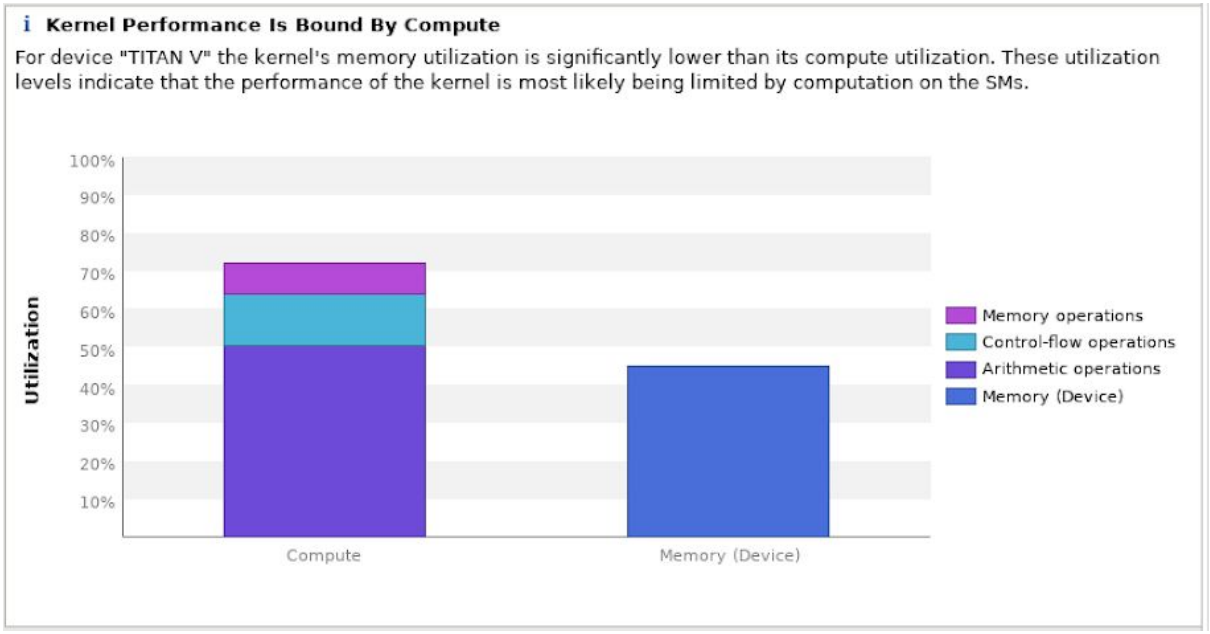
- **Description**

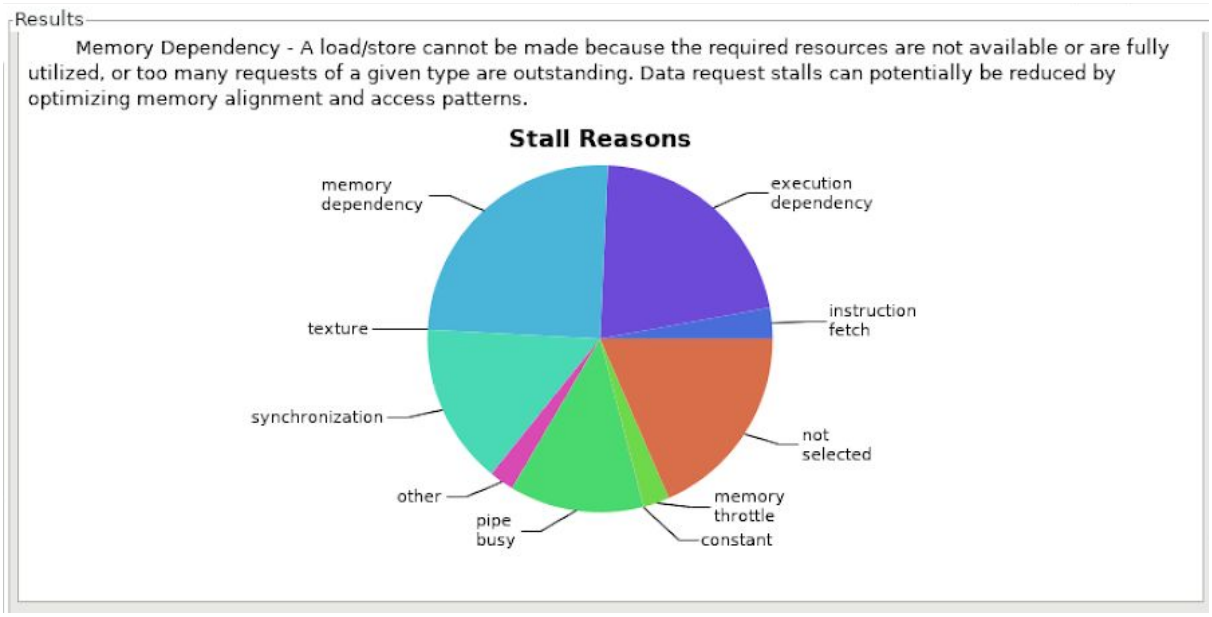
  We tune our implementation by adding the *restrict* keyword to our arrays. That is, we add **__restrict__** for our x, y, and w arrays. According to an article from Nvidia (https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/), adding this keyword to our arrays “greatly helps the compiler optimize code”. Along with this, in CUDA, using the restrict keyword caches our values so that we have faster data accesses. We also perform loop unrolling in our kernel. We unroll so that we use less threads but each thread ends up performing more work. In our code, we unroll five times when calculating our *acc* value.

- **Timeline Overview**



- **Forward Layer 1**





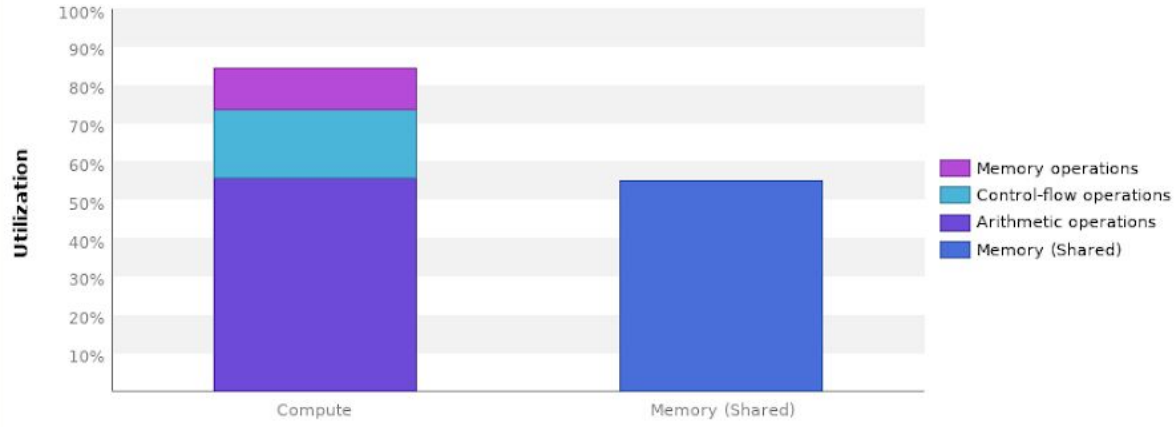
Results

Active Blocks		6	32	0 5 10 15 20 25 3032
Active Warps	45.64	48	64	0 10 20 30 40 50 6064
Active Threads		1536	2048	0 512 1024 1536 2048
Occupancy	71.3%	75%	100%	0% 50% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 5 10 15 20 25 3032
Block Limit		8	32	0 5 10 15 20 25 3032
Registers				
Registers/Thread		39	65536	0 16384 32768 49152 65536
Registers/Block		10240	65536	0 32k 64k
Block Limit		6	32	0 5 10 15 20 25 3032
Shared Memory				
Shared Memory/Block		1700	98304	0 32k 64k 96k
Block Limit		54	32	0 5 10 15 20 25 3032

- **Forward Layer 2**

i Kernel Performance Is Bound By Compute

For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



Results

potentially be reduced by increasing instruction-level parallelism.

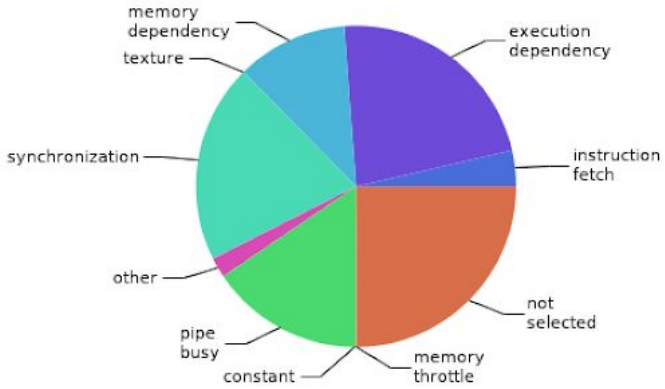
Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Synchronization - The warp is blocked at a __syncthreads() call.

Constant - A constant load is blocked due to a miss in the constants cache.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

Stall Reasons



Results

Active Blocks		6	32	0 5 10 15 20 25 3032
Active Warps	47.47	48	64	0 10 20 30 40 50 6064
Active Threads		1536	2048	0 512 1024 1536 2048
Occupancy	74.2%	75%	100%	0% 50% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 5 10 15 20 25 3032
Block Limit		8	32	0 5 10 15 20 25 3032
Registers				
Registers/Thread		39	65536	0 16384 32768 49152 65536
Registers/Block		10240	65536	0 32k 64k
Block Limit		6	32	0 5 10 15 20 25 3032
Shared Memory				
Shared Memory/Block		1700	98304	0 32k 64k 96k
Block Limit		54	32	0 5 10 15 20 25 3032