Team: Undecided

On Campus
Members: Shrey Chowdhary (shreyc2), Vijay Klein (vijaygk2), Alex Krysl (krysl2)

Milestone 4

Constant Memory

The implementation to include constant memory resulted in a relatively inconsequential difference in performance results. From multiple runs, the average time to completion was the same as with the non-constant memory implementation. My estimated guess as to why this is the case is that there might be optimizations on the hardware of the GPU that in effect create a constant memory kernel on its own. Nonetheless, with the foresight that the convolution kernel is constant throughout the entirety of the program, implementing it explicitly as constant memory is prudent.

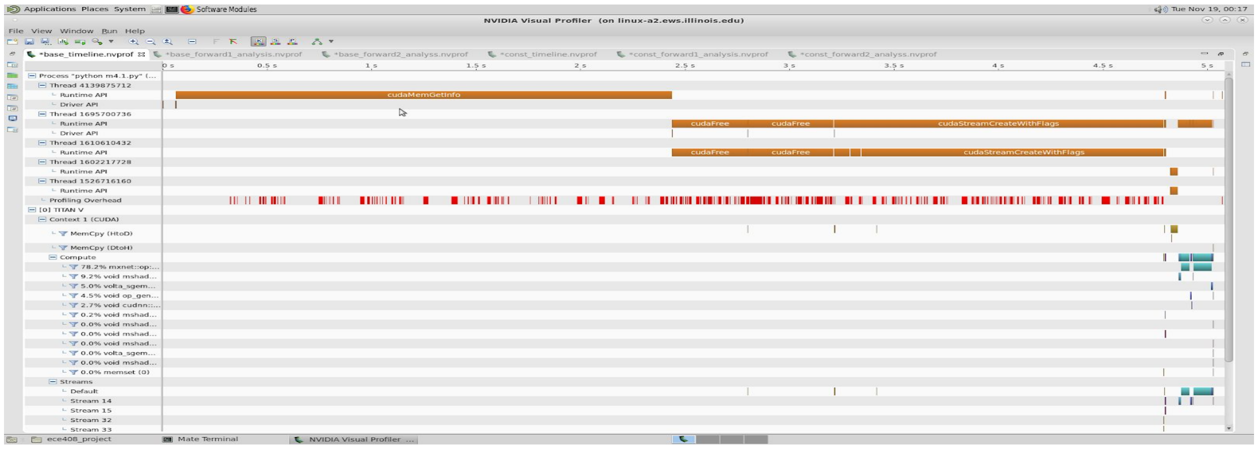
Shared Memory

The implementation of shared memory resulted in mixed performance results. On the first pass with only one input feature map and larger input matrix had a speedup as it took 87% of the time compared to the constant memory version. Meanwhile the second pass had a significant slowdown as it took 234% longer than the constant memory. There are a couple factors that caused these wildly different results. Firstly the code doesn't parallelize the input feature map (C) which means the shared code iterates sequentially through all the input feature maps to load into shared memory and then iterate again to compute. This explains why the first pass is faster, but the second is slower as it has more than one input feature map. Secondly the warp efficiency is reduced on the second pass as it goes from 90.7% to 50.4% while the first pass improves the warp efficiency compared to constant memory. I believe this is due to shared memory having inactive threads and this problem is magnified by the number of iterations of input feature map.

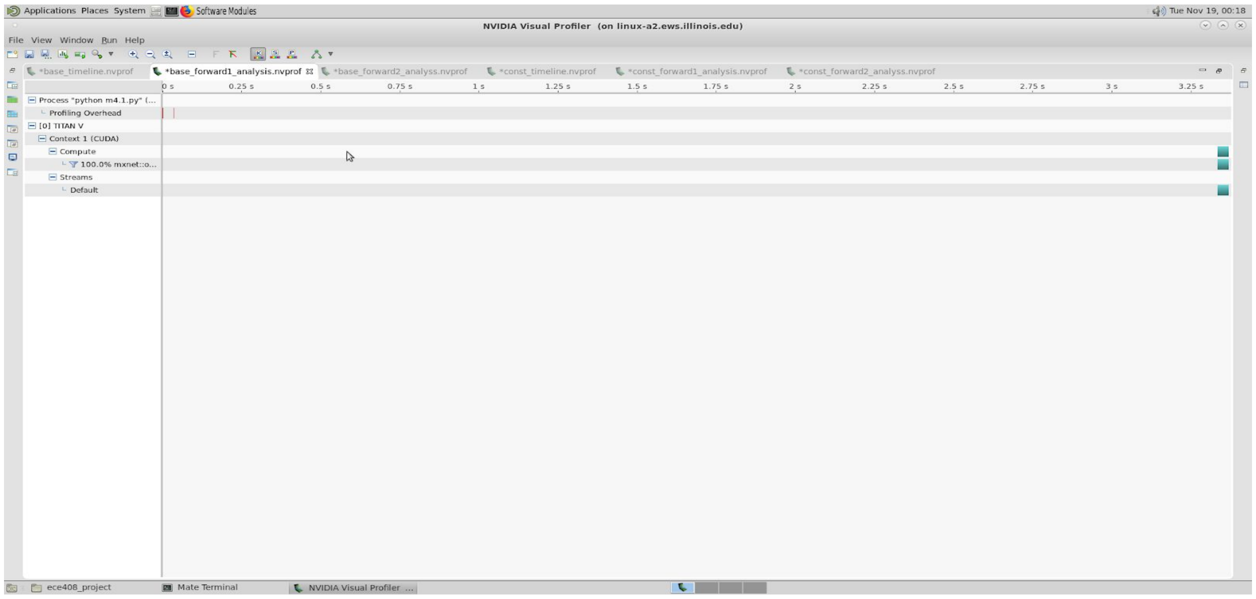
Loop Unrolling/Restricting pointers

The implementation of loop unrolling and restricting pointers was rather straightforward. The compiler is smart enough to optimize the code itself with the direction of pragmas; however, it did require adding the restrict keyword to the input pointers. The restrict keyword ensured that the pointers do not alias (overlap), which enables the compiler to do many more optimizations. The results of this optimization were quite successful as, using the shared memory optimization as a baseline, the first pass went from 29 to 26ms, taking only 89% of the execution time, and the second pass had a significant speedup as well, being reduced from 182 to 165ms, taking only 90% of the execution time. Unrolling the loops will make the binary size larger, due to more instructions, but it makes sense that it would improve the running time, especially when combined with restricting the pointers. This is because the compiler can safely rearrange groups of memory access together which reduces latency when compared to interleaving computations and memory accesses. This is heavily supported by the nvprof output, where the memory analysis shows greater throughput in every metric after applying the loop unrolling, for example device memory reads increasing from 150 GB/s to 170 GB/s for the first pass.

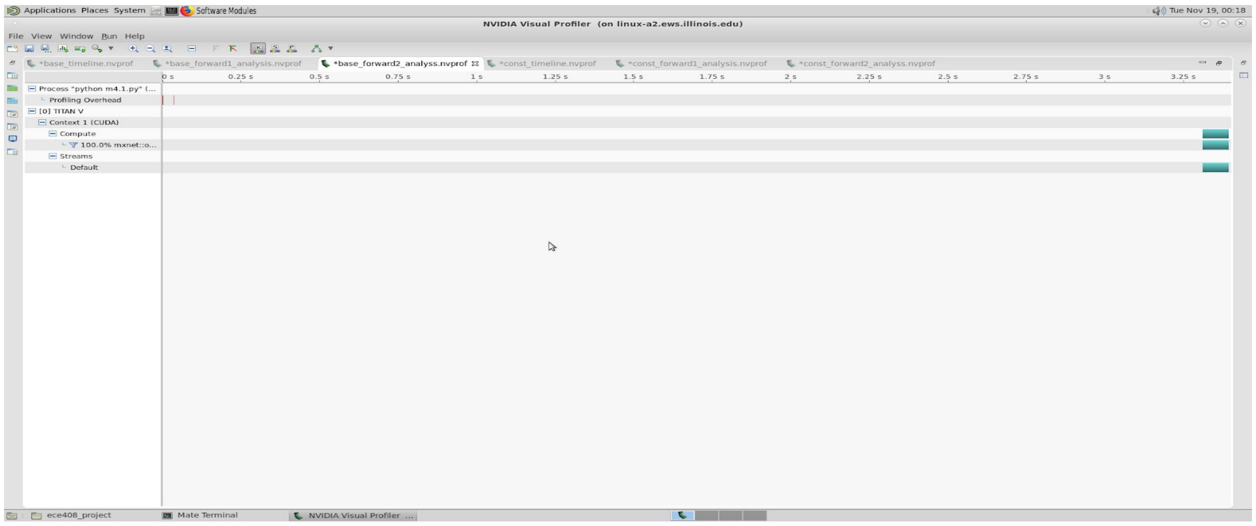
Base_timeline.nvprof - NVVP



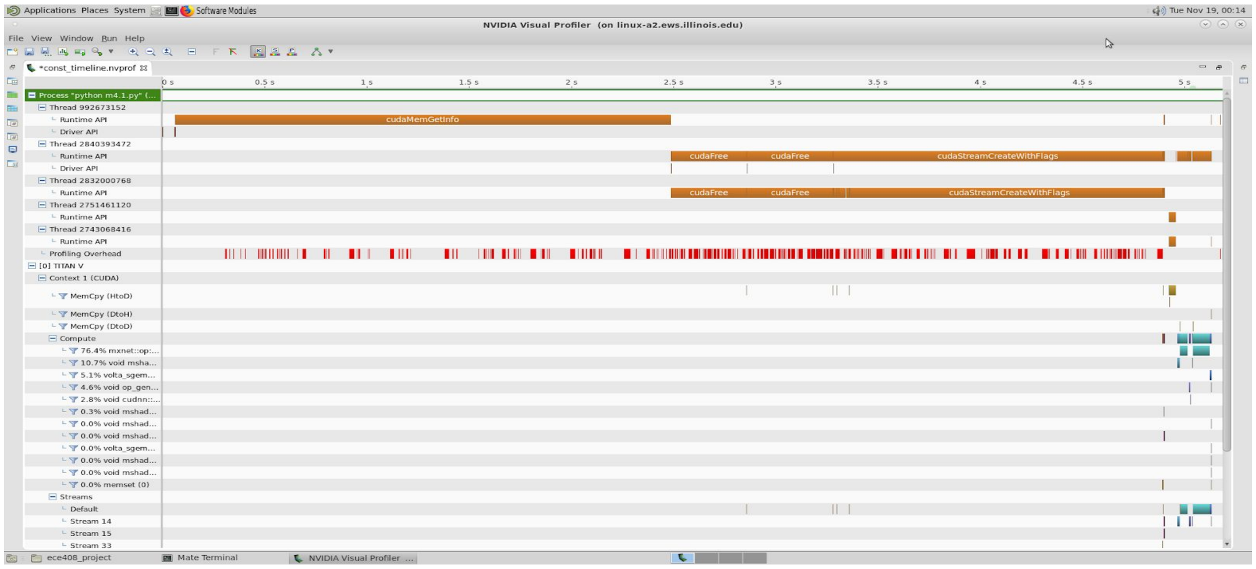
Base_forward1_anaylsis.nvprof - NVVP



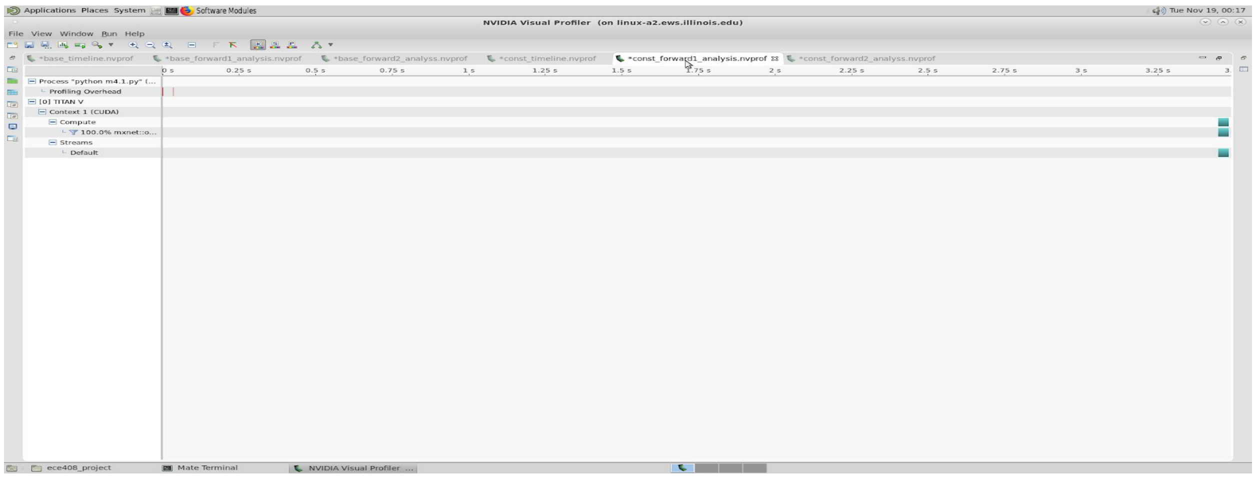
Base_forward2_anaylsis.nvprof - NVVP


NVIDIA Visual Profiler (on linux-a2.ews.illinois.edu)
Tue Nov 19, 00:18
*base_timeline.nvprof
*base_forward1_analysis.nvprof
*base_forward2_analyss.nvprof
*const_timeline.nvprof
*const_forward1_analysis.nvprof
*const_forward2_analyss.nvprof
Process "python m4.1.py" (...
Profiling Overhead
[0] TITAN V
Context 1 (CUDA)
Compute
100.0% mxnet::o...
Streams
Default
ece408_project
Mate Terminal
NVIDIA Visual Profiler ...

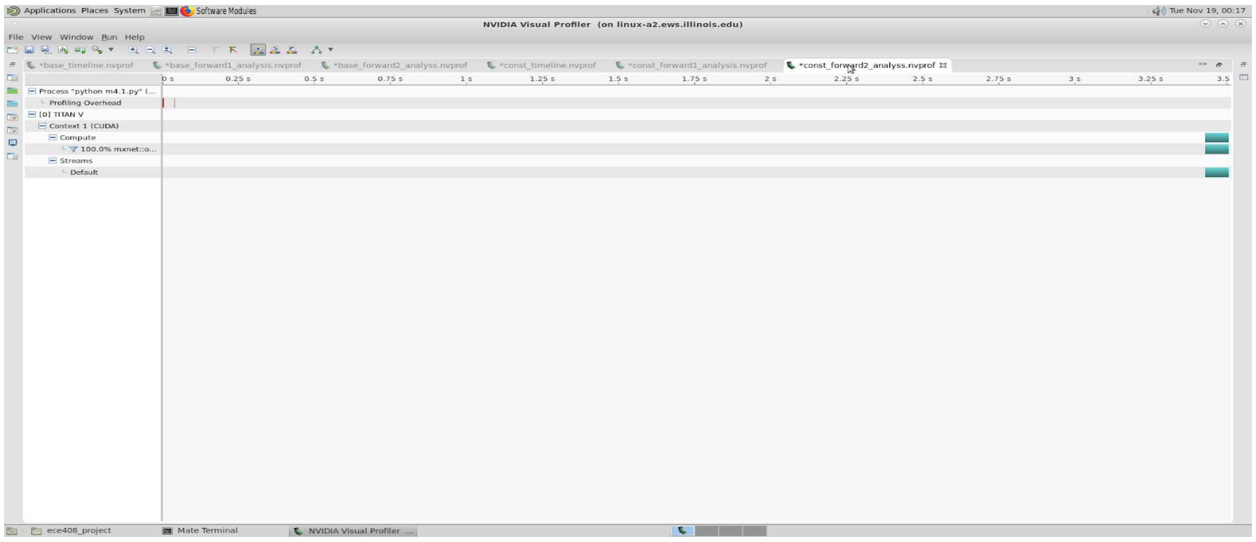
Const_timeline.nvprof - NVVP



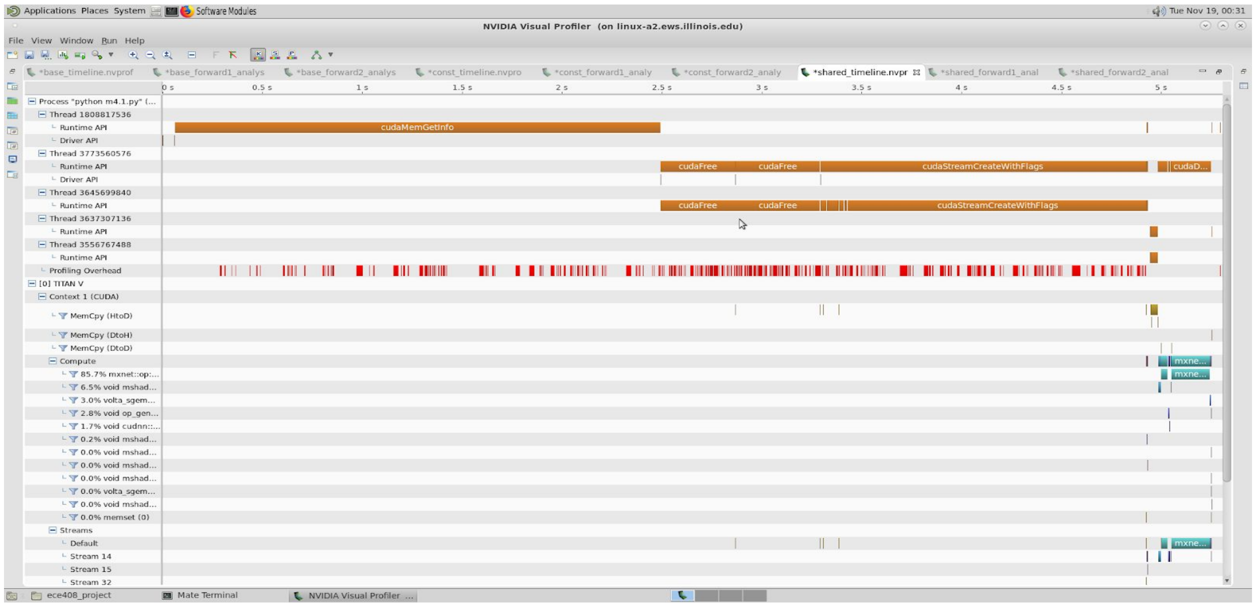
Const_forward1_analysis.nvprof - NVVP


NVIDIA Visual Profiler (on linux-a2.ews.illinois.edu)
Tue Nov 19, 00:17
*base_timeline.nvprof
*base_forward1_analysis.nvprof
*base_forward2_analyss.nvprof
*const_timeline.nvprof
*const_forward1_analysis.nvprof
*const_forward2_analyss.nvprof
Process "python m4.1.py" (...
Profiling Overhead
[0] TITAN V
Context 1 (CUDA)
Compute
100.0% mxnet::o...
Streams
Default
ece408_project
Mate Terminal
NVIDIA Visual Profiler ...

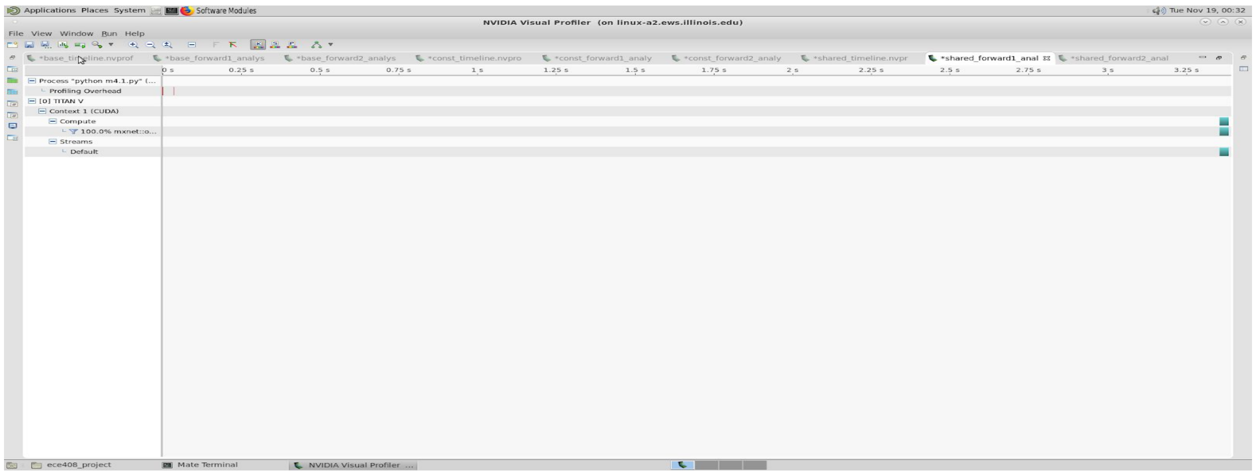
Const_forward2_analysis.nvprof - NVVP



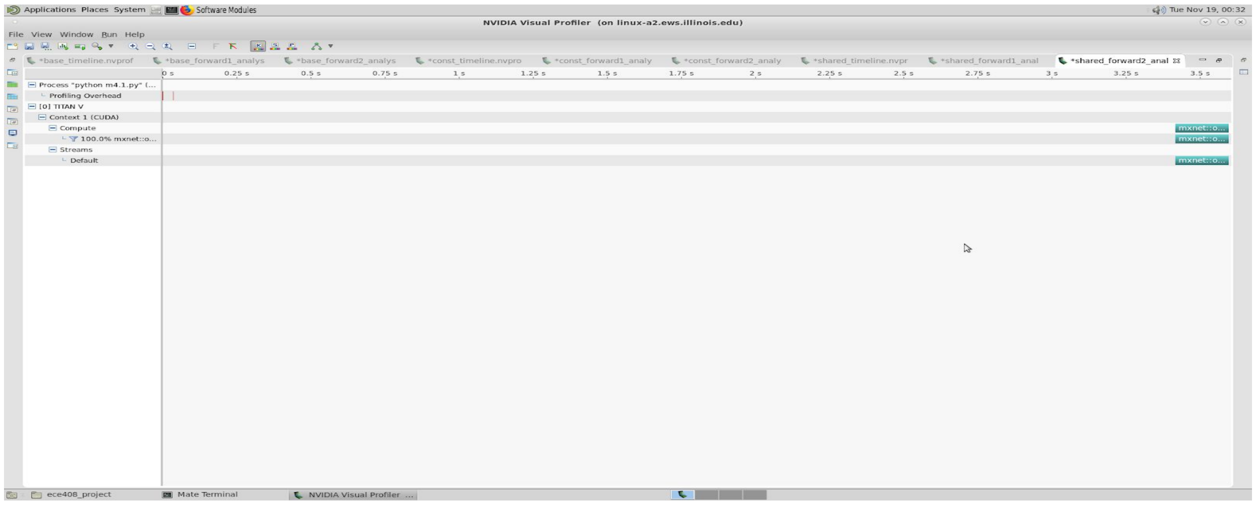
Shared_timeline.nvprof - NVVP



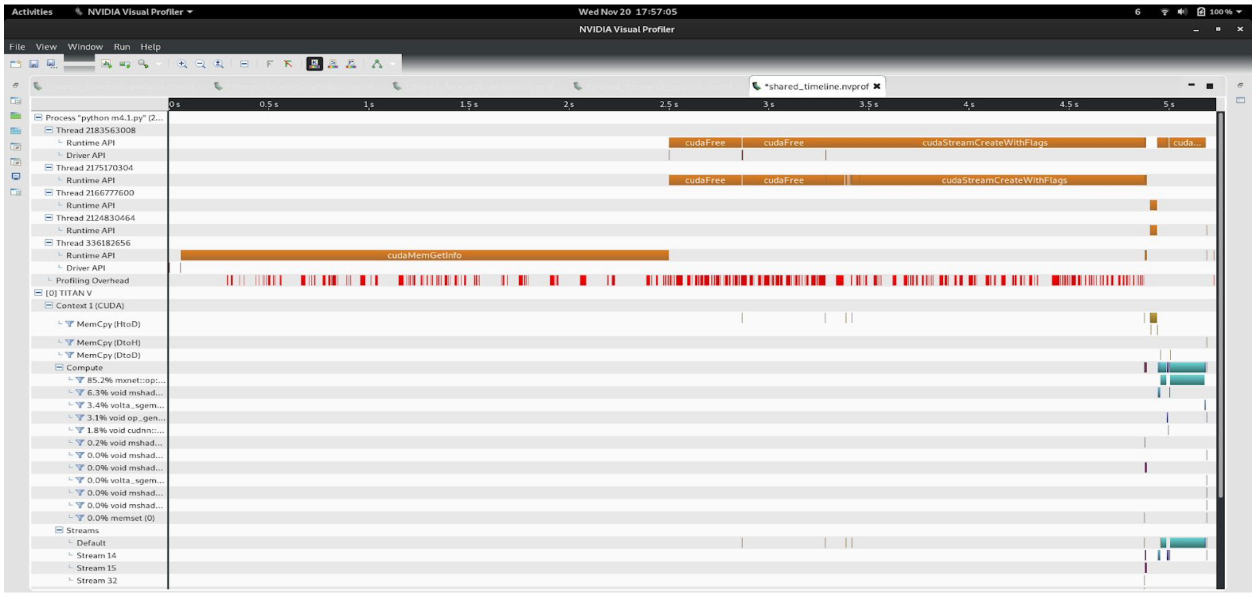
Shared_forward1_analysis.nvprof - NVVP



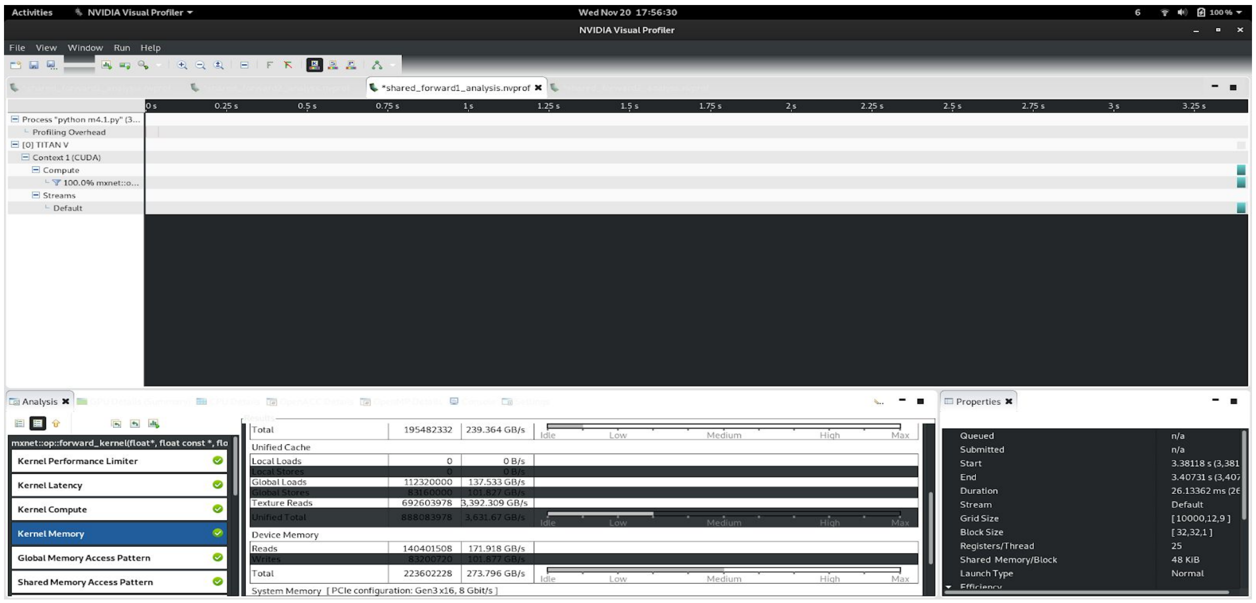
Shared_forward2_analysis - NVVP



Unroll_timeline.nvprof - NVVP



Unroll_forward1.nvprof - NVVP


Wed Nov 20 17:56:30

*shared_forward1_analysis.nvprof

```
Process "python m4.1.py" (3...
  Profiling Overhead
[0] TITAN V
  Context 1 (CUDA)
    Compute
      100.0% mxnet::o...
    Streams
      Default
```

mxnet::op::forward_kernel(float*, float const *, flo

- Kernel Performance Limiter ✓
- Kernel Latency ✓
- Kernel Compute ✓
- Kernel Memory ✓
- Global Memory Access Pattern ✓
- Shared Memory Access Pattern ✓

Total	195482332	239.364 GB/s

Unified Cache

Local Loads	0	0 B/s
Local Stores	0	0 B/s
Global Loads	112320000	137.533 GB/s
Global Stores	83160000	101.827 GB/s
Texture Reads	692603978	3,392.309 GB/s
Unified Total	888083978	3,631.67 GB/s

Device Memory

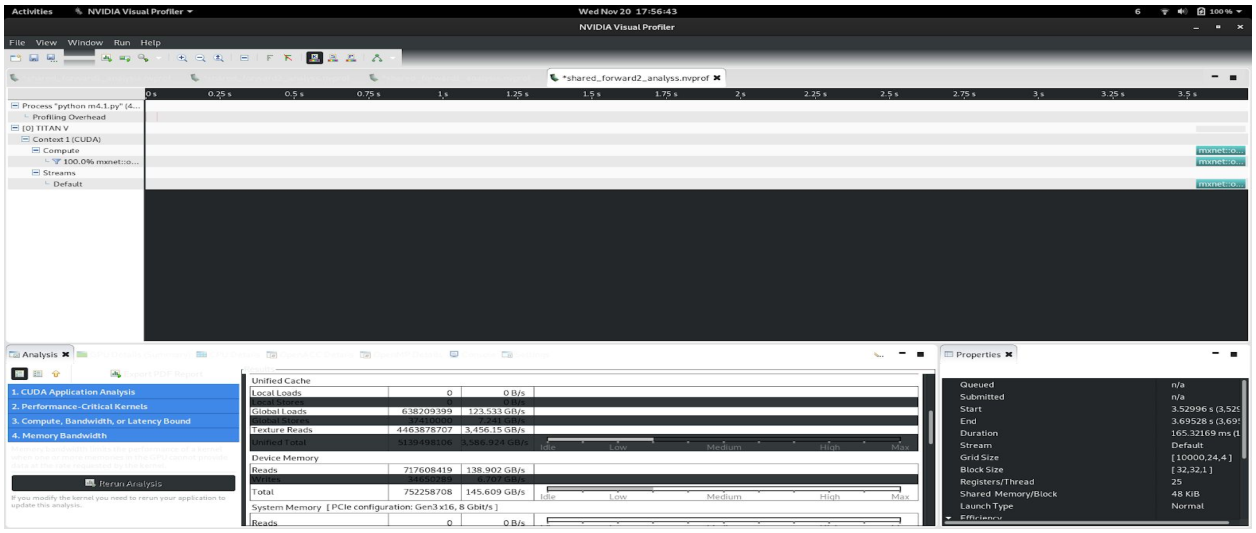
Reads	140401508	171.918 GB/s
Writes	83200720	101.877 GB/s
Total	223602228	273.796 GB/s

System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]

Properties

Queued	n/a
Submitted	n/a
Start	3.38118 s (3,381
End	3.40731 s (3,407
Duration	26.13362 ms (26
Stream	Default
Grid Size	[10000,12,9]
Block Size	[32,32,1]
Registers/Thread	25
Shared Memory/Block	48 KiB
Launch Type	Normal

Unroll_forward2.nvprof - NVVP


Wed Nov 20 17:56:43

*shared_forward2_analyss.nvprof

```
Process "python m4.1.py" (4...
  Profiling Overhead
[0] TITAN V
  Context 1 (CUDA)
    Compute
      100.0% mxnet::o...
    Streams
      Default
```

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Memory Bandwidth

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Unified Cache

Local Loads	0	0 B/s
Local Stores	0	0 B/s
Global Loads	638209399	123.533 GB/s
Global Stores	37410000	7.241 GB/s
Texture Reads	4463878707	3,456.15 GB/s
Unified Total	5139498106	3,586.924 GB/s

Device Memory

Reads	717608419	138.902 GB/s
Writes	34650289	6.707 GB/s
Total	752258708	145.609 GB/s

System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]

Reads	0	0 B/s

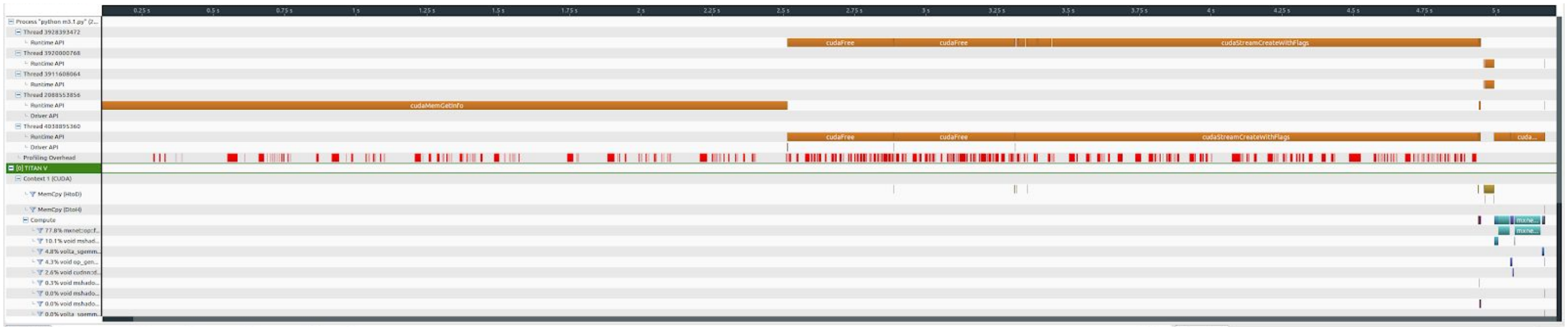
Properties

Queued	n/a
Submitted	n/a
Start	3.52996 s (3,529
End	3.69528 s (3,695
Duration	165.32169 ms (1
Stream	Default
Grid Size	[10000,24,4]
Block Size	[32,32,1]
Registers/Thread	25
Shared Memory/Block	48 KiB
Launch Type	Normal

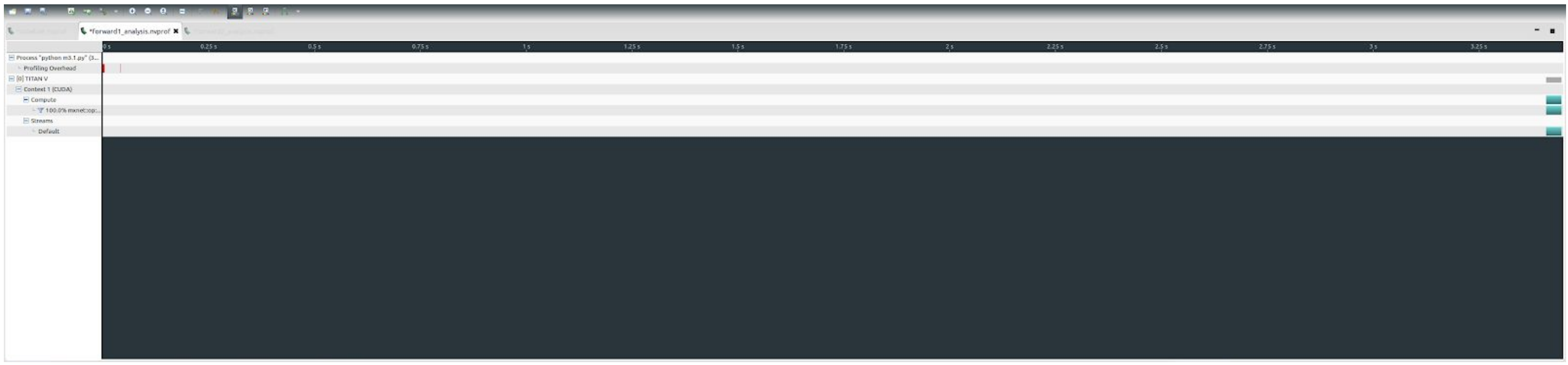
<u>Milestone 3</u>

Report: demonstrate nvprof profiling the execution

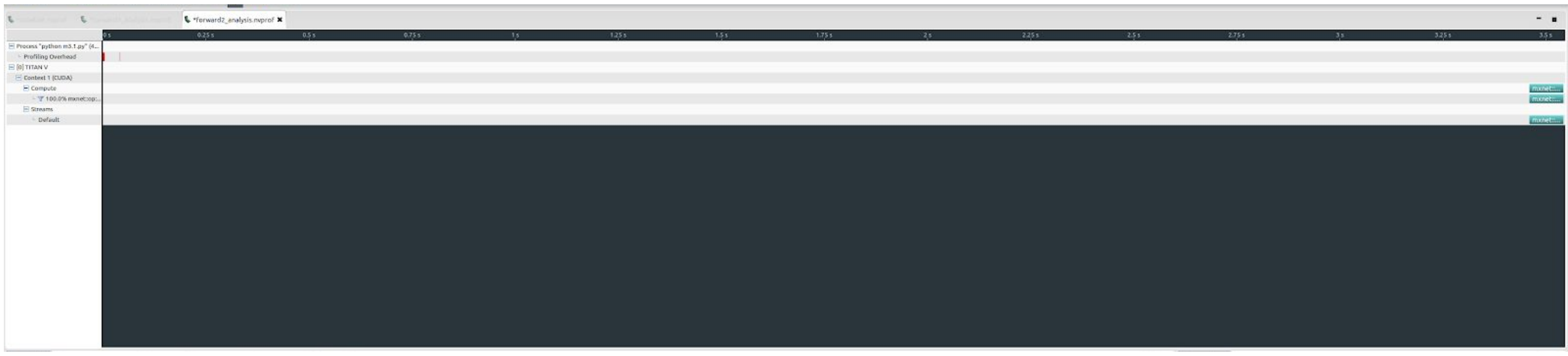
timeline.nvprof



forward1_analysis.nvprof



forward2_analysis.nvprof



According to NVVP we have low compute utilization meaning many of the SM were idling. Looking at the timeline trace it seems like computation on the GPU is taking very little time relative to the CUDA API calls on host code. Further, within our kernel, we also see a low global memory load efficiency. This metric is the number of bytes requested divided by the number of bytes transferred from device memory. The actual value for this metric was 62.5 percent which indicates that 37.5 percent of the memory coming out of the DRAM bursts was not requested and likely not used. Since memory is typically the slowest piece in the architecture, increasing this value certainly would be a viable optimization method in the future. Finally, we also noticed a fairly high divergence rate with 22.9 percent of threads diverging on the following line.

```
if (h >= H_out || w >= W_out) {
        return;
}
```

This source of this issue is the dimensions of the Grid/Blocks which obviously should not really be an issue for this milestone as we are prioritizing functionality, but again, in the future, this is certainly an area where we can improve. Still, this check is more or less required in some form, as we do not want to be writing in a thread that "falls out" of the bounds of the array.

Milestone 2

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

Time (%)	**Name**
31.97%	**[CUDA memcpy HtoD]**
17.85%	**volta_scudnn_128x64_relu_interior_nn_v1**
17.16%	**volta_gcgemm_64x32_nt**
8.56%	**fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=0, 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)**
7.79%	**volta_sgemm_128x128_tn**
6.50%	**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)**
5.70%	**void fft2d_r2c_32x32<float, bool=0, unsigned int=0, bool=0>(float2*, float**

	const *, int, int, int, int, int, int, int, int, int, cudnn::reduced_divisor, bool, int2, int, int)

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

Time (%)	**Name**
42.14%	**cudaStreamCreateWithFlags**
33.26%	**cudaMemGetInfo**
21.03%	**cudaFree**

Difference between kernels and API calls

Both kernels and API calls potentially interact with the device/GPU in the system; however, the way in which they do so is the key distinction.

Essentially, an API call is a function provided in the driver (run on the host) that can interact with the device, some examples of which are allocating and freeing device memory (cudaMalloc and cudaFree, respectively). The key note here is that while the code interacts with the device's hardware, the code itself is running on the host, and there are no threads spawning/computations being done on the GPU device (i.e. the majority of the code/functionality is occurring on the host, rather than the device).

This is in contrast to a kernel, where the kernel is called from the host, but directly spawns threads on the GPU device, which all then execute the kernel code on the device, in parallel.

Report: Show output of rai running MXNet on the CPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
```

Report: List program run time

```
17.25user 4.45system 0:08.95elapsed 242%CPU (0avgtext+0avgdata 6045864maxres
ident)k
0inputs+2824outputs (0major+1600460minor)pagefaults 0swaps
```

So, 8.95 wall-clock seconds for a run on the CPU.

Report: Show output of rai running MXNet on the GPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
```

Same accuracy makes sense and is worth noting.

Report: List program run time

```
5.20user 3.19system 0:04.63elapsed 181%CPU (0avgtext+0avgdata 2975644max
resident)k
```

So, 4.63 wall-clock seconds for a run on the GPU. Also makes sense this would run faster.

CPU whole program execution time

```
86.44user 7.99system 1:16.76elapsed
```

So, 1:16.76 wall-clock minutes for a run on the CPU.

CPU Op Times

```
Op Time: 11.020163
Op Time: 62.090408
```