CS 483 Project Report

Team_name: hawks
Names: Nikhil Paidipally, Daniel Niewierowski, Nicolas Nytko
Net_IDs: npaidi2, nnytko2, dniewi2
RAI_IDs: 5d97b1f588a5ec28f9cb94b0, 5d97b1f488a5ec28f9cb94ae, 5d97b1f388a5ec28f9cb94ac
Affiliation: uiuc

MILESTONE 1:

Kernels that collectively consume more than 90% of the program time:

- `[CUDA memcpy HtoD]`
- `volta_scudnn_128x64_relu_interior_nn_v1`
- `volta_gcgemm_64x32_nt`
- `fft2d_c2r_32x32`
- `volta_sgemm_128x128_tn`
- `op_generic_tensor_kernel`
- `fft2d_r2c_32x32`

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

- `cudaStreamCreateWithFlags`
- `cudaMemGetInfo`
- `cudaFree`

Difference between kernels and API calls:

CUDA kernels are effectively like regular C functions that are executed an arbitrary number of times in parallel by an arbitrary number of different CUDA threads on the gpu/device.

CUDA API are effectively regular C functions that execute on the cpu/host and they are not executed in parallel like the CUDA kernels. Some examples of CUPA APIs include cudamalloc, cudaMemCpy, cudafree, etc.

Rai running MXNet on the CPU:

(CPU C Code)

```
Loading fashion-mnist data... done
Loading model... done
New Inference
```

```
Op Time: 11.330177
Op Time: 74.542234
Correctness: 0.7653 Model: ece408
100.71user 11.16system 1:29.81elapsed 124%CPU (0avgtext+0avgdata
6044220maxresident)k
0inpu
ts+0outputs (0major+2308767minor)pagefaults 0swaps
```

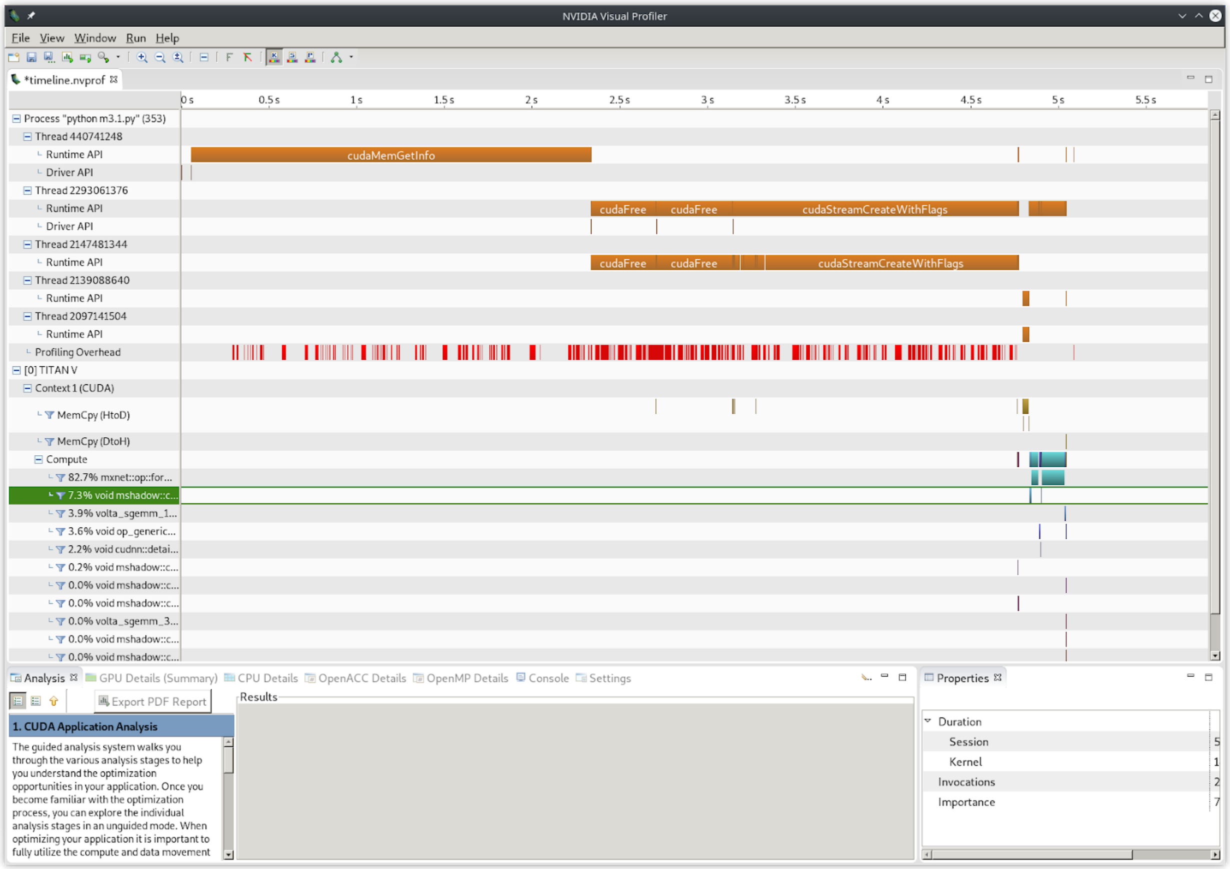
(CPU Python Code)

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

17.90user 4.36system 0:09.78elapsed 227%CPU (0avgtext+0avgdata
6045072maxresident)k
0inputs+2824outputs (
0major+1603748minor)pagefaults 0swaps
```

Rai running MXNet on the GPU:

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

4.96user 3.07system 0:04.63elapsed 173%CPU (0avgtext+0av
gdata 2996488maxresident)k
0inputs+1712outputs (0major+734016minor)pagefaults 0swaps
```

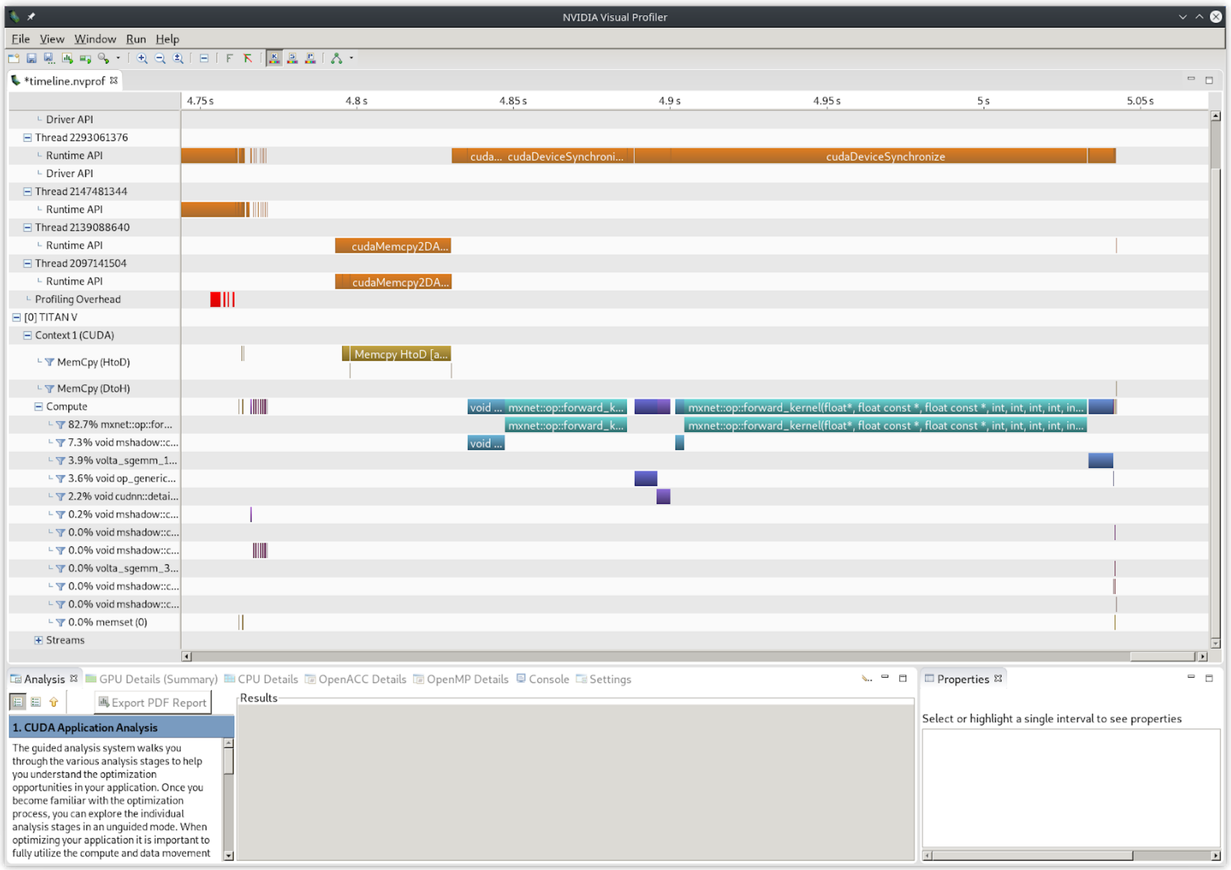
<u>MILESTONE 3:</u>



1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement

(View of overall CUDA execution)



The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement

Select or highlight a single interval to see properties

(Zoomed view of the forward-prop convolution kernel)

MILESTONE 4:

To establish a baseline, we first ran our convolution without any optimisations several times and recorded the average running time for both operations, detailed below:

```
Op Time: 0.045837
Op Time: 0.134574
Correctness: 0.755 Model: ece408
```

This was our “time to beat” so to speak.

OPTIMISATION 1:

The first optimisation we performed was to tune the kernel with the restrict keyword unroll the innermost loop. The __restrict__ keyword was added to the arguments y, x, and k to signal to the compiler that the memory regions are non-overlapping and thus the pointers do not alias. In some cases this can provide significant speed-ups. The innermost convolution loop was also unrolled, since the mask size is fixed at 5 elements in each direction.

Implementing the above improved our times slightly, but it was hard to judge overall effectiveness due to the randomness of the timing values. Perhaps the loop control does not affect the running time too heavily.

```
Op Time: 0.036734
Op Time: 0.111906
Correctness: 0.755 Model: ece408
```

<u>OPTIMISATION 2:</u>

Our second optimisation was to improve the kernel itself by using shared memory to cache tiled sections of the image. For each image channel, a block of threads will first load in a tiled section together, then perform the convolution on the tiled section. This approach was modeled after the “strategy 2” tiled convolution proposed in lecture.

Additionally, we were able to catch a bound error and increase our correctness slightly.

```
Op Time: 0.026975
Op Time: 0.076901
Correctness: 0.7643 Model: ece408
```

<u>OPTIMISATION 3:</u>

For our third optimisation, we performed a sweep of the block values in order to determine the best (local) running time. Since the maximum block size is defined as 1024 threads in CUDA, our domain of block values can range from 5 to 32.

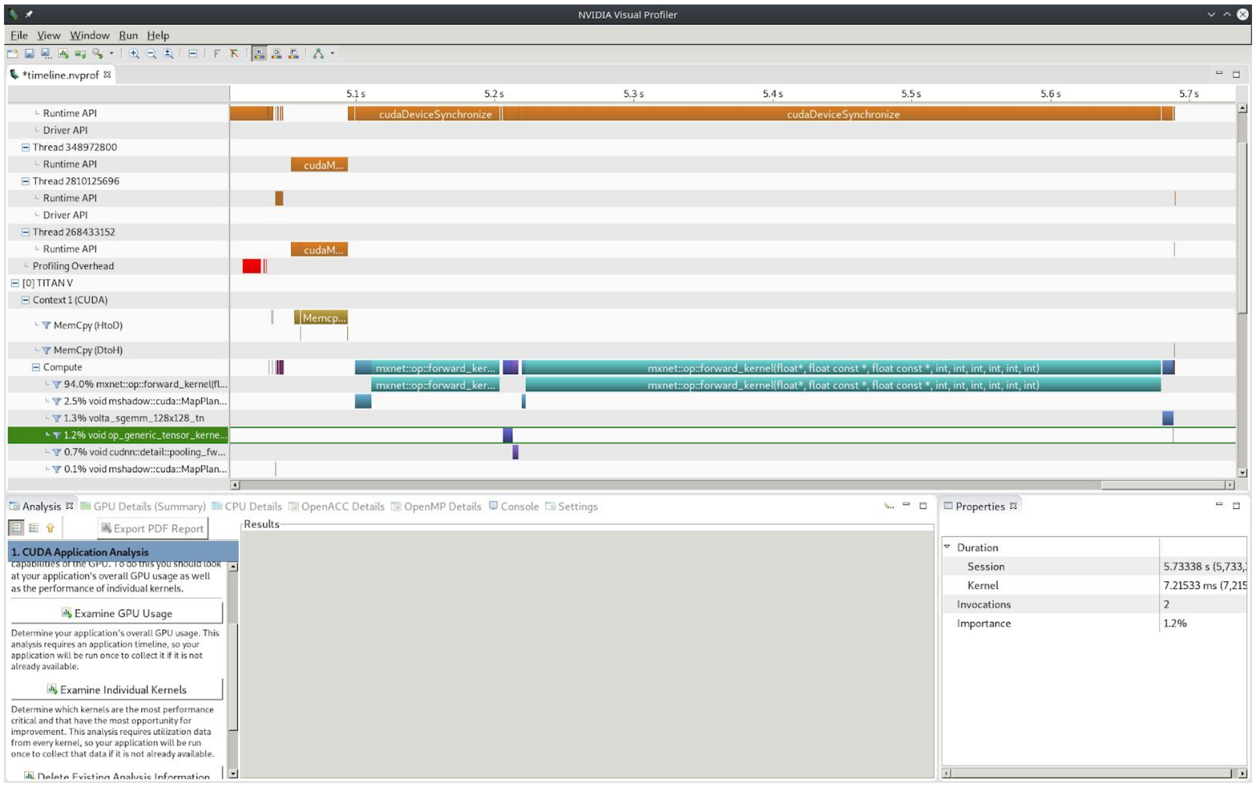
We modeled our sweep after the golden section search, a method for optimising 1-dimensional functions.

Block Size	**Op1 Time (s)**	**Op2 Time (s)**
15	0.09702	0.533354
21	0.08562	0.456664
25	0.123988	0.681743
19	0.109751	0.39073
20	0.131026	0.437012

22	0.9062	0.507769

The local minimum was found experimentally to be a block size of 21.

<u>PROFILER VIEW:</u>



1. CUDA Application Analysis

capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.

Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.

Duration	
Session	5.73338 s (5,733,
Kernel	7.21533 ms (7,215
Invocations	2
Importance	1.2%

(View of the optimised kernel in *NVVP*)