# MINI-PROJECT 1

LEARNING MACHINES- 259

**Madhav Sankar Krishnakumar** madhavsankar@ucla.edu

**Parth Shettiwar** parthshettiwar@g.ucla.edu **Satvik Mashkaria** satvikm@g.ucla.edu

July 4, 2022

## Contents

# 1 Kernels

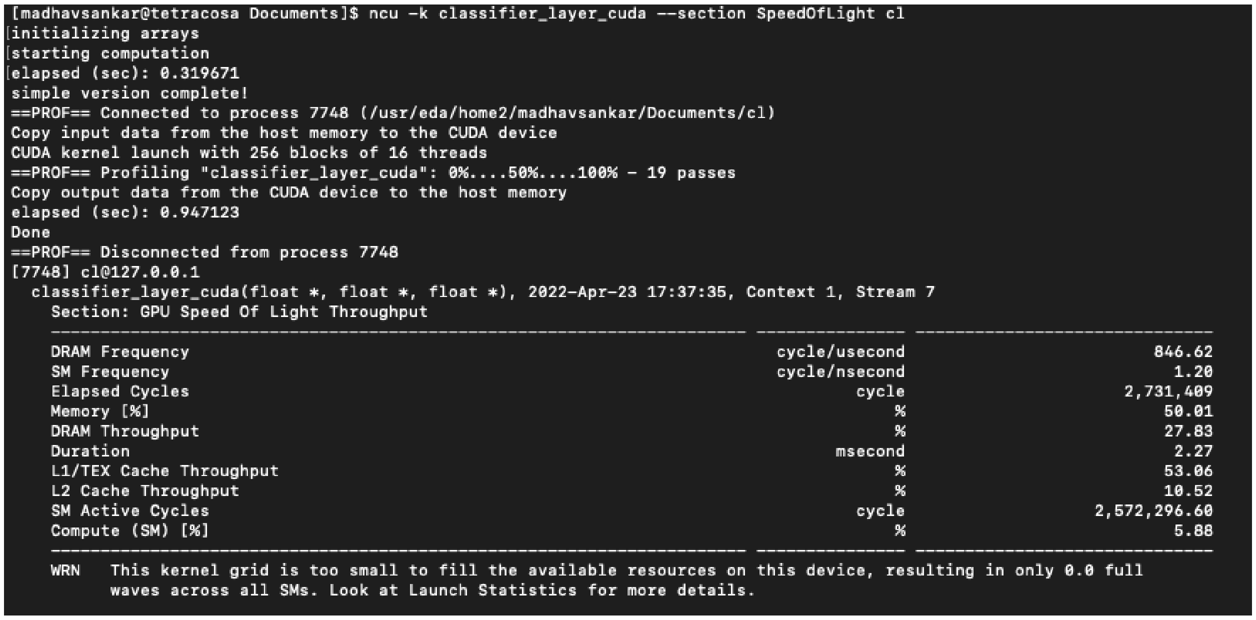
## 1.1 Classifier

Two Classifiers were used for all the models:

- Classifier 1: Ni=25088 Nn=4096
- Classifier 2: Ni=4096 Nn=1024

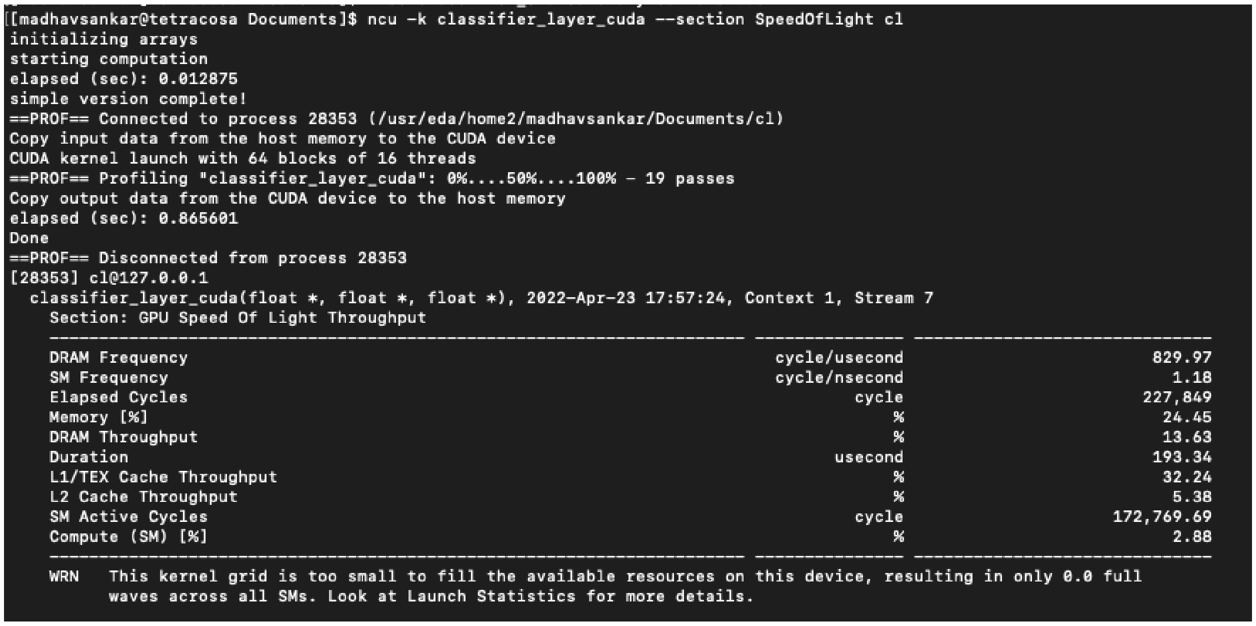
### 1.1.1 Baseline Model

The baseline model produced the following results for the 2 classifiers.



```
[madhavsankar@tetracosa Documents]$ ncu -k classifier_layer_cuda --section SpeedOfLight cl
initializing arrays
starting computation
elapsed (sec): 0.319671
simple version complete!
==PROF== Connected to process 7748 (/usr/eda/home2/madhavsankar/Documents/cl)
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 256 blocks of 16 threads
==PROF== Profiling "classifier_layer_cuda": 0%....50%....100% - 19 passes
Copy output data from the CUDA device to the host memory
elapsed (sec): 0.947123
Done
==PROF== Disconnected from process 7748
[7748] cl@127.0.0.1
  classifier_layer_cuda(float *, float *, float *), 2022-Apr-23 17:37:35, Context 1, Stream 7
    Section: GPU Speed Of Light Throughput
    ---------------------------------------------------------------------- --------------- ------------------------------
    DRAM Frequency                                                           cycle/usecond                         846.62
    SM Frequency                                                             cycle/nsecond                           1.20
    Elapsed Cycles                                                                   cycle                      2,731,409
    Memory [%]                                                                           %                          50.01
    DRAM Throughput                                                                      %                          27.83
    Duration                                                                       msecond                           2.27
    L1/TEX Cache Throughput                                                              %                          53.06
    L2 Cache Throughput                                                                  %                          10.52
    SM Active Cycles                                                                 cycle                   2,572,296.60
    Compute (SM) [%]                                                                     %                           5.88
    ---------------------------------------------------------------------- --------------- ------------------------------
    WRN   This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full
          waves across all SMs. Look at Launch Statistics for more details.
```

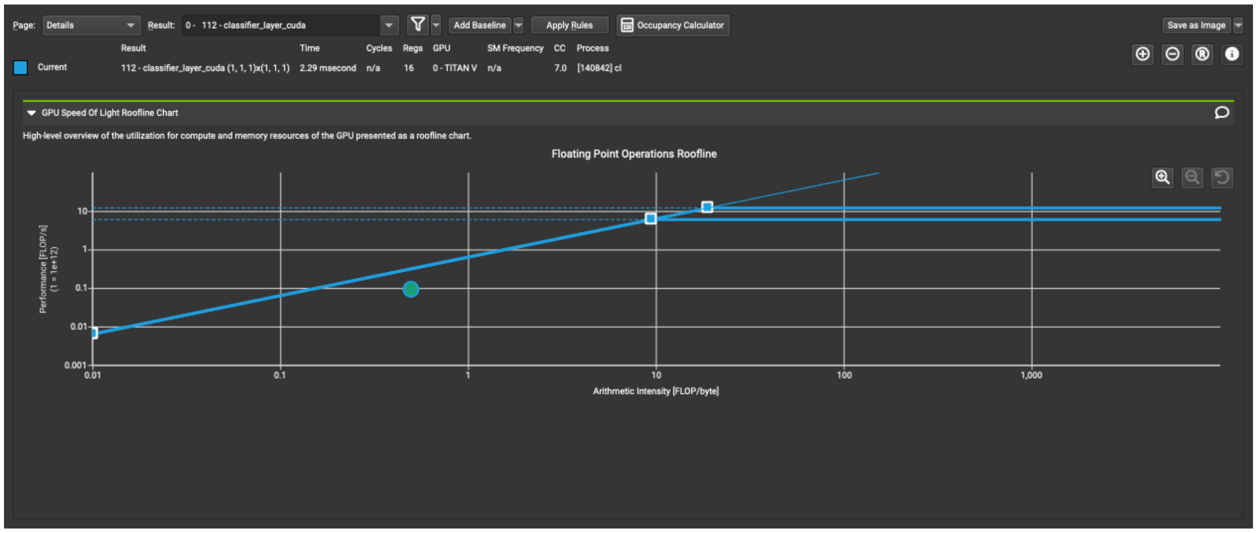
Figure 1: Baseline Model for Classifier 1



```
[[madhavsankar@tetracosa Documents]$ ncu -k classifier_layer_cuda --section SpeedOfLight cl
initializing arrays
starting computation
elapsed (sec): 0.012875
simple version complete!
==PROF== Connected to process 28353 (/usr/eda/home2/madhavsankar/Documents/cl)
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 64 blocks of 16 threads
==PROF== Profiling "classifier_layer_cuda": 0%....50%....100% - 19 passes
Copy output data from the CUDA device to the host memory
elapsed (sec): 0.865601
Done
==PROF== Disconnected from process 28353
[28353] cl@127.0.0.1
  classifier_layer_cuda(float *, float *, float *), 2022-Apr-23 17:57:24, Context 1, Stream 7
    Section: GPU Speed Of Light Throughput
    ---------------------------------------------------------------------- --------------- ------------------------------
    DRAM Frequency                                                           cycle/usecond                         829.97
    SM Frequency                                                             cycle/nsecond                           1.18
    Elapsed Cycles                                                                   cycle                        227,849
    Memory [%]                                                                           %                          24.45
    DRAM Throughput                                                                      %                          13.63
    Duration                                                                       usecond                         193.34
    L1/TEX Cache Throughput                                                              %                          32.24
    L2 Cache Throughput                                                                  %                           5.38
    SM Active Cycles                                                                 cycle                     172,769.69
    Compute (SM) [%]                                                                     %                           2.88
    ---------------------------------------------------------------------- --------------- ------------------------------
    WRN   This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full
          waves across all SMs. Look at Launch Statistics for more details.
```

Figure 2: Baseline Model for Classifier 2

As can be seen, the larger dataset of Classifier 1 would prove as a better measure to compare various optimizations as the various metrics are large enough to observe the differences. We plotted the Roofline Chart for Baseline Classifier 1 and observed the following:



```
Page: Details   Result: 0 -  112 - classifier_layer_cuda

         Result                                      Time           Cycles  Regs  GPU          SM Frequency  CC   Process
Current  112 - classifier_layer_cuda (1, 1, 1)x(1, 1, 1)  2.29 msecond  n/a     16    0 - TITAN V  n/a           7.0  [140842] cl

GPU Speed Of Light Roofline Chart
High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
```

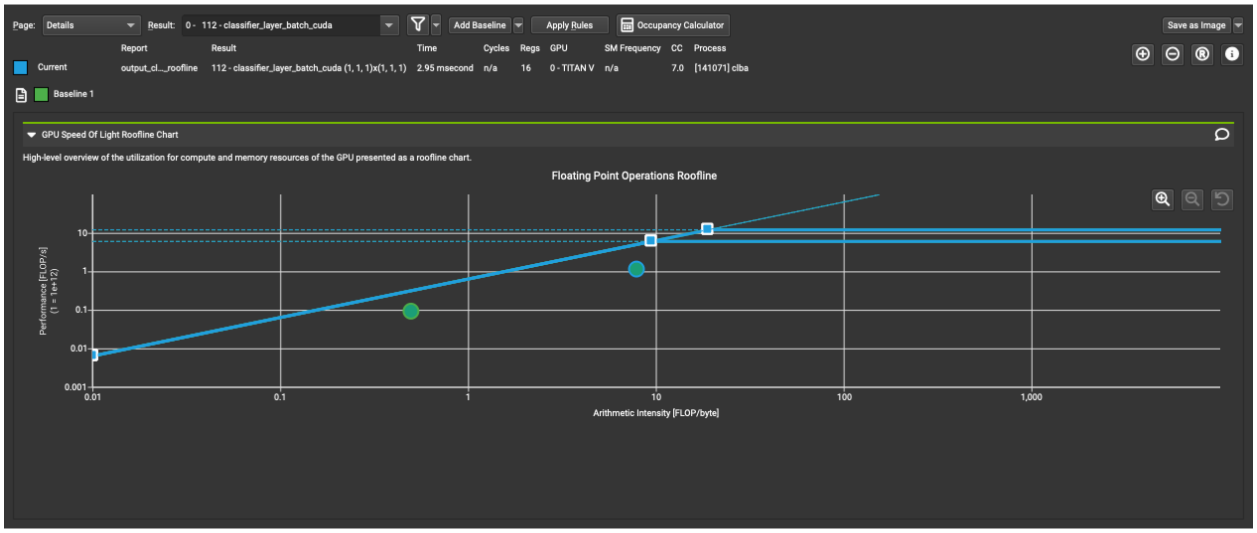
Add Baseline
Apply Rules
Occupancy Calculator
Save as Image
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1e+12)
10
1
0.1
0.01
0.001
0.01
0.1
1
10
100
1,000
Arithmetic Intensity [FLOP/byte]

Figure 3: Roofline Chart for Baseline Model

The initial roofline analysis in the figure shows that the arithmetic intensity of the kernel is just low enough to fall under the sloped memory-bound roofline in the chart. The achieved arithmetic intensity is under FLOP/byte, but the machine balance point for the GPU in double-precision is an arithmetic intensity of 10. This is the point where the kernel doing enough work to become compute-bound. We therefore need to increase the arithmetic intensity enough to fall under one of the horizontal compute-bound ceilings instead. That gives a better chance of maximizing the compute performance of this kernel.

### 1.1.2 Batching

As we observed in the previous section, the roofline analysis proved that the kernel is in the memory-bound section. We try to move it to the compute-bound region by batching the inputs together using a batch size of 16. This way the compute has to increase. We observe the following roofline analysis. The baseline is retained for comparison.



| | Report | Result | Time | Cycles | Regs | GPU | SM Frequency | CC | Process |
|---|---|---|---|---|---|---|---|---|---|
| 🟦 Current | output_cl..._roofline | 112 - classifier_layer_batch_cuda (1, 1, 1)x(1, 1, 1) | 2.95 msecond | n/a | 16 | 0 - TITAN V | n/a | 7.0 | [141071] clba |
| 🟩 Baseline 1 | | | | | | | | | |

High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Figure 4: Roofline Chart for Batching as compared to the baseline model.

Now, we observe that the kernel is close to the compute bound region. Let us also show the results for both classifiers. We can see that the duration is much lesser than running the baseline model 16 times.

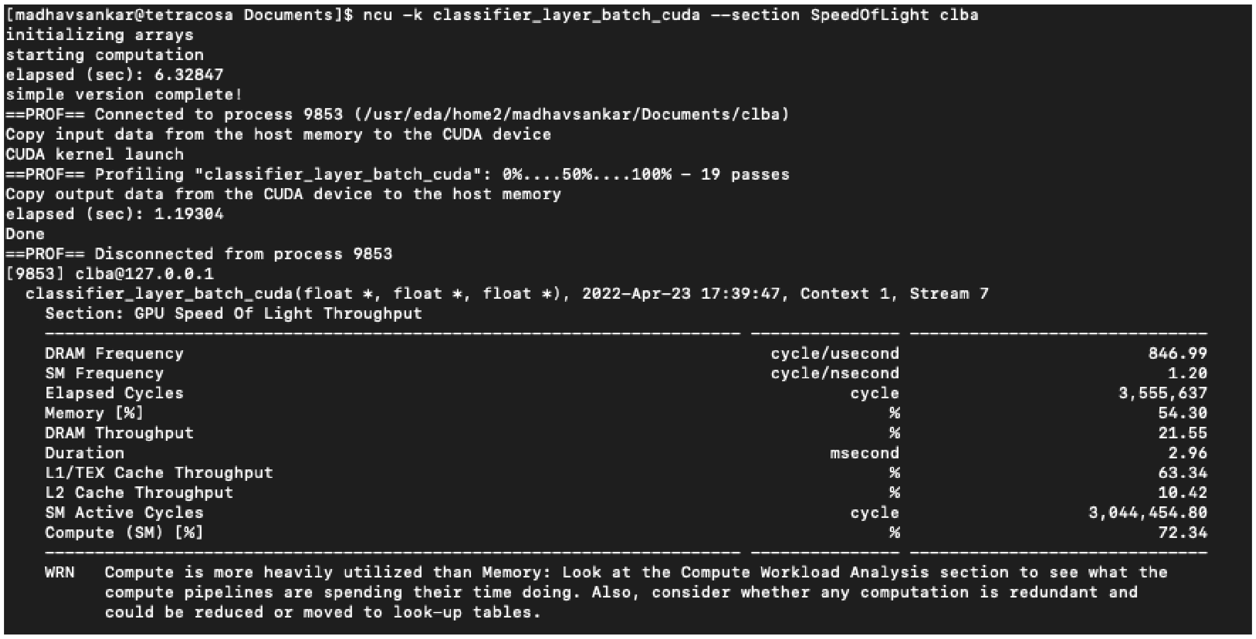
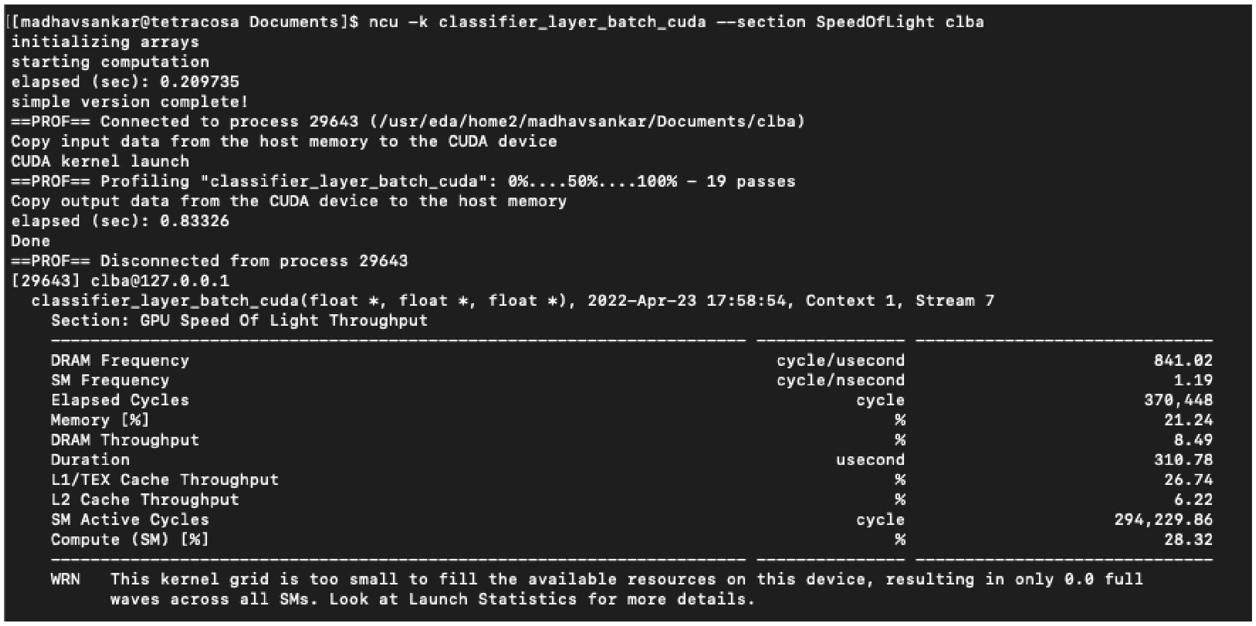

```
[madhavsankar@tetracosa Documents]$ ncu -k classifier_layer_batch_cuda --section SpeedOfLight clba
initializing arrays
starting computation
elapsed (sec): 6.32847
simple version complete!
==PROF== Connected to process 9853 (/usr/eda/home2/madhavsankar/Documents/clba)
Copy input data from the host memory to the CUDA device
CUDA kernel launch
==PROF== Profiling "classifier_layer_batch_cuda": 0%....50%....100% - 19 passes
Copy output data from the CUDA device to the host memory
elapsed (sec): 1.19304
Done
==PROF== Disconnected from process 9853
[9853] clba@127.0.0.1
  classifier_layer_batch_cuda(float *, float *, float *), 2022-Apr-23 17:39:47, Context 1, Stream 7
    Section: GPU Speed Of Light Throughput
    ---------------------------------------------------------------------- --------------- ------------------------------
    DRAM Frequency                                                          cycle/usecond                          846.99
    SM Frequency                                                            cycle/nsecond                            1.20
    Elapsed Cycles                                                                  cycle                       3,555,637
    Memory [%]                                                                          %                           54.30
    DRAM Throughput                                                                     %                           21.55
    Duration                                                                      msecond                            2.96
    L1/TEX Cache Throughput                                                             %                           63.34
    L2 Cache Throughput                                                                 %                           10.42
    SM Active Cycles                                                                cycle                    3,044,454.80
    Compute (SM) [%]                                                                    %                           72.34
    ---------------------------------------------------------------------- --------------- ------------------------------
    WRN   Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis section to see what the
          compute pipelines are spending their time doing. Also, consider whether any computation is redundant and
          could be reduced or moved to look-up tables.
```

Figure 5: Batching Model for Classifier 1

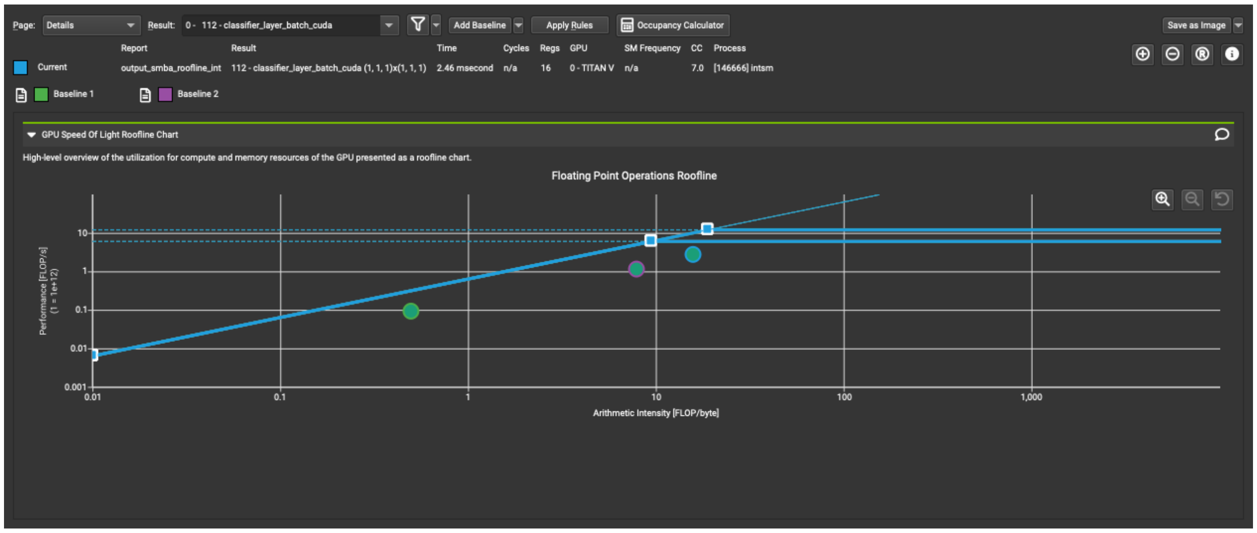


```
[[madhavsankar@tetracosa Documents]$ ncu -k classifier_layer_batch_cuda --section SpeedOfLight clba
initializing arrays
starting computation
elapsed (sec): 0.209735
simple version complete!
==PROF== Connected to process 29643 (/usr/eda/home2/madhavsankar/Documents/clba)
Copy input data from the host memory to the CUDA device
CUDA kernel launch
==PROF== Profiling "classifier_layer_batch_cuda": 0%....50%....100% - 19 passes
Copy output data from the CUDA device to the host memory
elapsed (sec): 0.83326
Done
==PROF== Disconnected from process 29643
[29643] clba@127.0.0.1
  classifier_layer_batch_cuda(float *, float *, float *), 2022-Apr-23 17:58:54, Context 1, Stream 7
    Section: GPU Speed Of Light Throughput
    ---------------------------------------------------------------------- --------------- ------------------------------
    DRAM Frequency                                                           cycle/usecond                         841.02
    SM Frequency                                                             cycle/nsecond                           1.19
    Elapsed Cycles                                                                   cycle                        370,448
    Memory [%]                                                                           %                          21.24
    DRAM Throughput                                                                      %                           8.49
    Duration                                                                       usecond                         310.78
    L1/TEX Cache Throughput                                                              %                          26.74
    L2 Cache Throughput                                                                  %                           6.22
    SM Active Cycles                                                                 cycle                     294,229.86
    Compute (SM) [%]                                                                     %                          28.32
    ---------------------------------------------------------------------- --------------- ------------------------------
    WRN   This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full
          waves across all SMs. Look at Launch Statistics for more details.
```

Figure 6: Batching Model for Classifier 2

### 1.1.3 Shared Memory (with tiling)

Considering that we managed to use batching to move the model towards being compute-bound, we can now apply shared memory to help improve the model further. Having shared memory can greatly decrease the memory bandwidth and improve the performance.

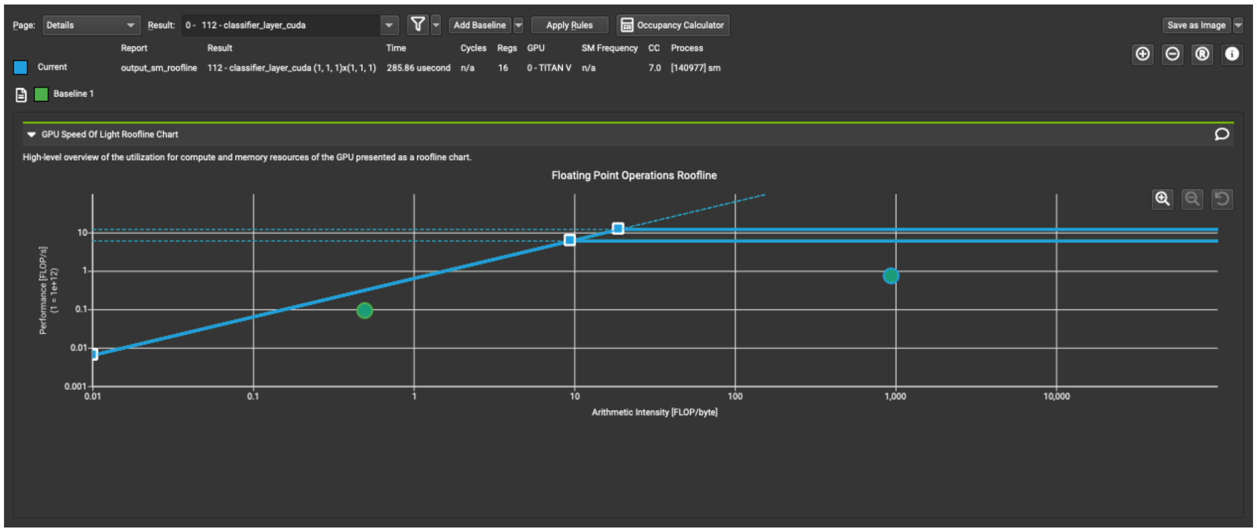


| | Report | Result | Time | Cycles | Regs | GPU | SM Frequency | CC | Process |
|---|---|---|---|---|---|---|---|---|---|
| 🟦 Current | output_smba_roofline_int | 112 - classifier_layer_batch_cuda (1, 1, 1)x(1, 1, 1) | 2.46 msecond | n/a | 16 | 0 - TITAN V | n/a | 7.0 | [146666] intsm |

High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Figure 7: Roofline Chart for Shared Memory + Batching as compared to the baseline model and just batching models.

Let us also produce the results for applying shared memory without batching. We can observe that the kernel has improved both in performance and Arithmetic Intensity and is well in the compute-bound region.

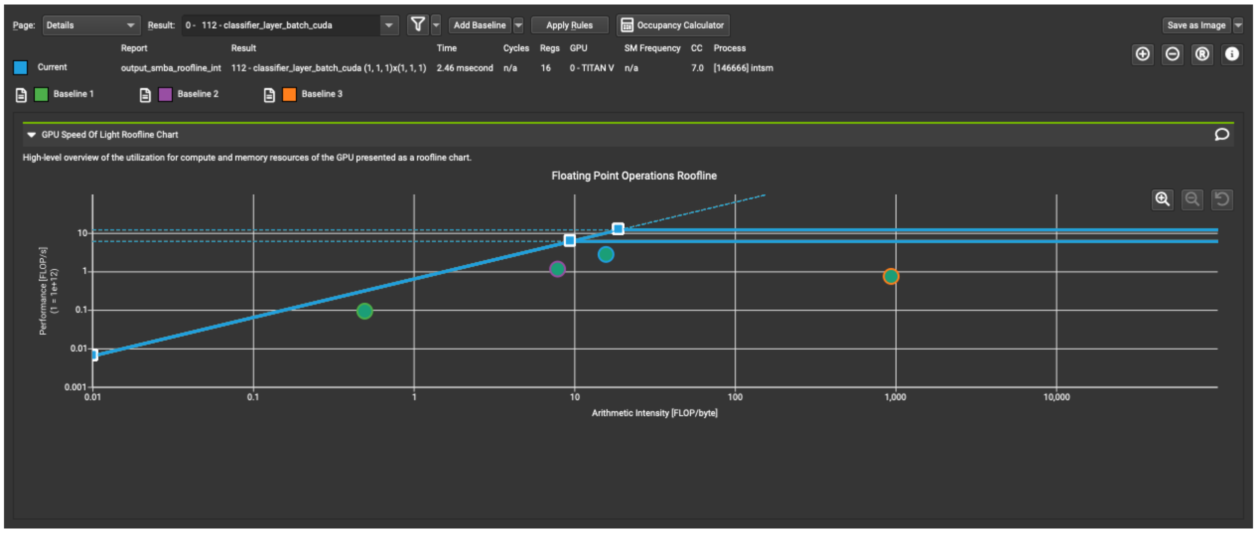


| | Report | Result | Time | Cycles | Regs | GPU | SM Frequency | CC | Process |
|---|---|---|---|---|---|---|---|---|---|
| 🟦 Current | output_sm_roofline | 112 - classifier_layer_cuda (1, 1, 1)x(1, 1, 1) | 285.86 usecond | n/a | 16 | 0 - TITAN V | n/a | 7.0 | [140977] sm |
| 🟩 Baseline 1 | | | | | | | | | |

Page: Details
Result: 0 - 112 - classifier_layer_cuda
Add Baseline
Apply Rules
Occupancy Calculator
Save as Image
GPU Speed Of Light Roofline Chart
High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1e+12)
Arithmetic Intensity [FLOP/byte]
0.001
0.01
0.1
1
10
100
1,000
10,000

Figure 8: Roofline Chart for Shared Memory as compared to the baseline model.

We can summarize all the roofline plot till now as:



| | Report | Result | Time | Cycles | Regs | GPU | SM Frequency | CC | Process |
|---|---|---|---|---|---|---|---|---|---|
| 🟦 Current | output_smba_roofline_int | 112 - classifier_layer_batch_cuda (1, 1, 1)x(1, 1, 1) | 2.46 msecond | n/a | 16 | 0 - TITAN V | n/a | 7.0 | [146666] intsm |
| 🟩 Baseline 1 | | | | | | | | | |
| 🟪 Baseline 2 | | | | | | | | | |
| 🟧 Baseline 3 | | | | | | | | | |

Page: Details
Result: 0 - 112 - classifier_layer_batch_cuda
Add Baseline
Apply Rules
Occupancy Calculator
Save as Image
GPU Speed Of Light Roofline Chart
High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1e+12)
Arithmetic Intensity [FLOP/byte]
0.001
0.01
0.1
1
10
100
1,000
10,000

Figure 9: Roofline Chart for Shared Memory as compared to the baseline model.

Now let us try to understand how the model works for different input matrix sizes. For simplicity, we shall assume that Nn = Ni. We shall compare 3 metrics for the comparison:

- Duration
- Compute (%)
- Memory (%)

**1. Impact of input size (without batching):**
Let us first try to understand the impact of input size for the shared memory model without batching:

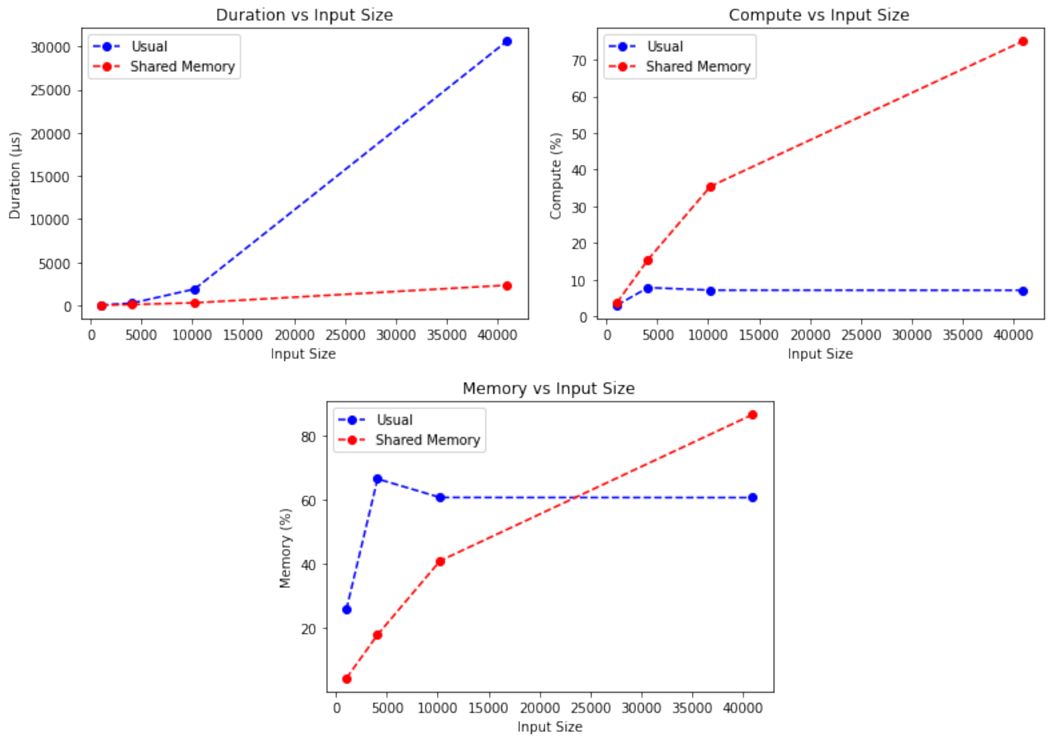


Figure 10: Metrics as a function of Input Size.

### 1.1. Observations:

- The duration increases exponentially for the baseline model with increase in input size. By adding shared memory, the rate of increase is brought down drastically.
- The compute seems to have increased by quite some margin while using shared memory. This explains the shift towards compute-bound region of the roofline chart.

### 2. Impact of input size (with batching):

Let us first try to understand the impact of input size for the shared memory model with batching:

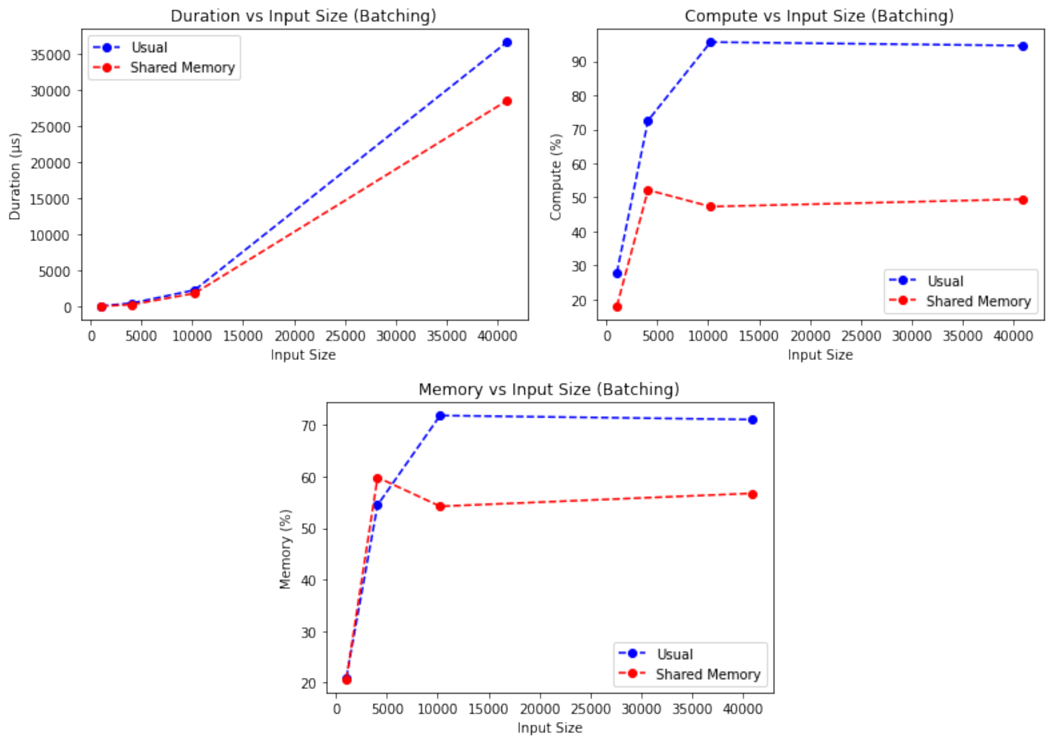


Figure 11: Metrics as a function of Input Size.

### 2.1 Observations:

- The compute seems to have decreased here unlike the previous scenario. This explains the shift towards compute-bound region of the roofline chart being much lesser when with batching. This could probably be because batching is already compute-intensive and hence the compute load added by shared memory is negligible.

## 3. Hyperparameter tuning:

Shared memory has a hyperparameter in the form of Block Size, which determines the shared memory size and the size of matrices that will be multiplied as a chunk. Let us try various block sizes like 8,16 etc. and determine the best value. We shall run it for both with and without batching.

### 3.1 Without Batching:

The various metric values as a function of Block Size is given below:

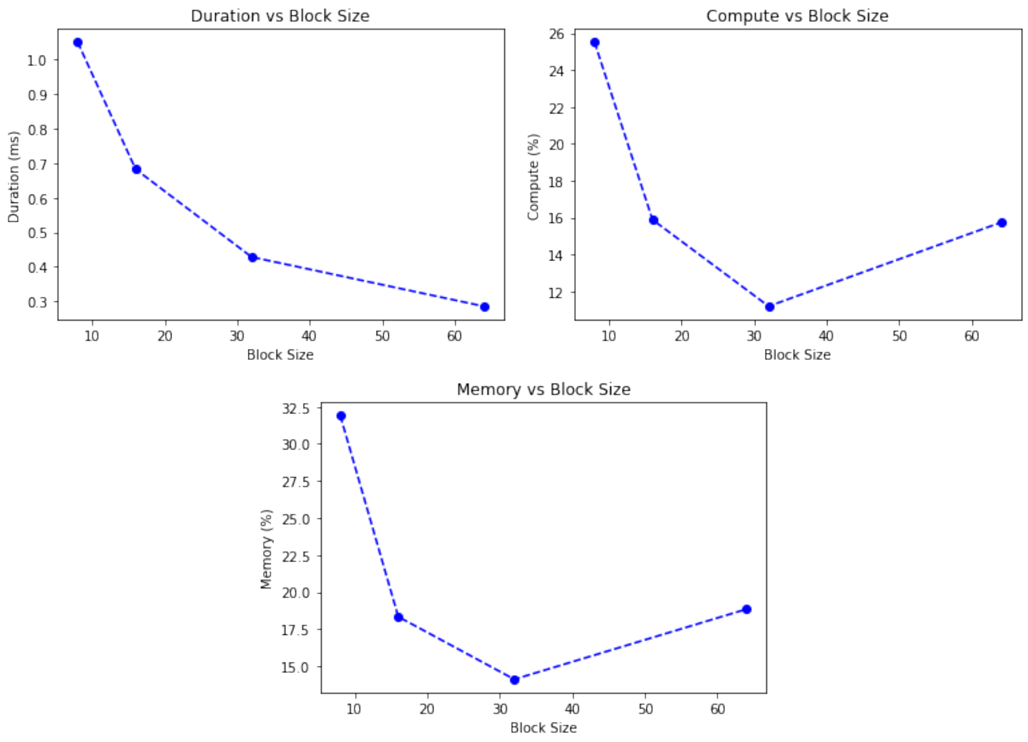


Figure 12: Metrics as a function of Block Size.

Duration seems to continuously drop with increasing block size but memory and compute seems to be least at block size of 32.

**3.1 With Batching:**

The various metric values as a function of Block Size is given below:

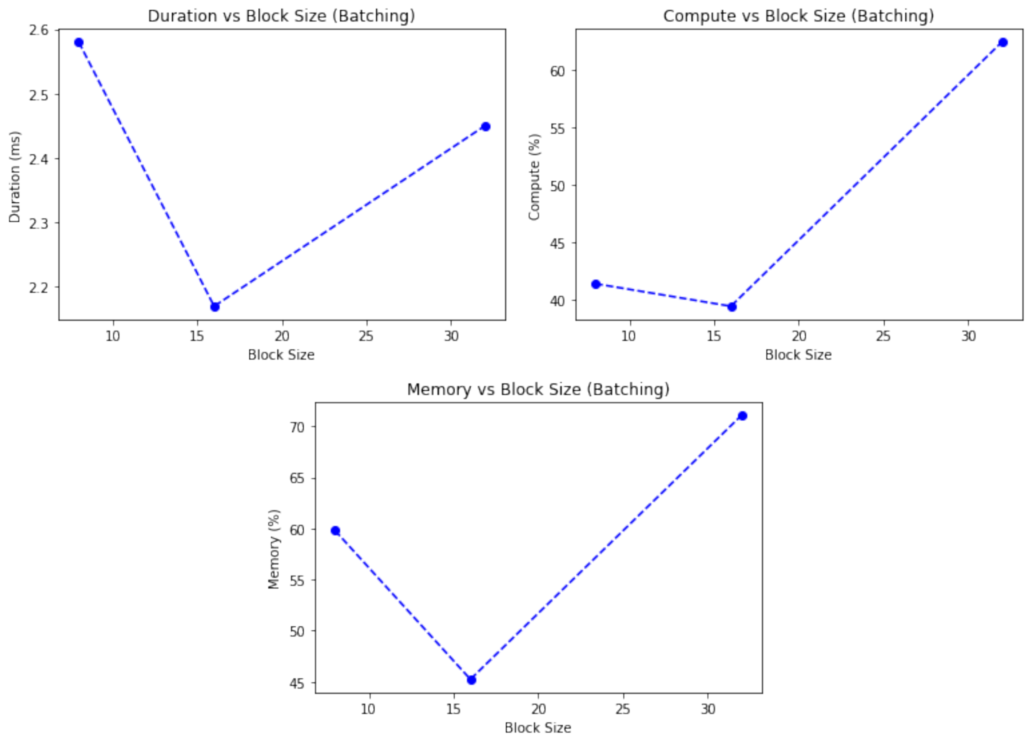


Figure 13: Metrics as a function of Block Size.

All the metrics seem to be least at block size of 16.

#### 1.1.4 Tiling

Square tiles were used for this part. A shared memory (private to each block) of size equal to the tile was created while computing the matrix multiplication. Batching was used in the experiments. Batched neuron of size $1024 \times 1024$ was multipled with synapse of size $1024 \times 1024$ in these experiments.

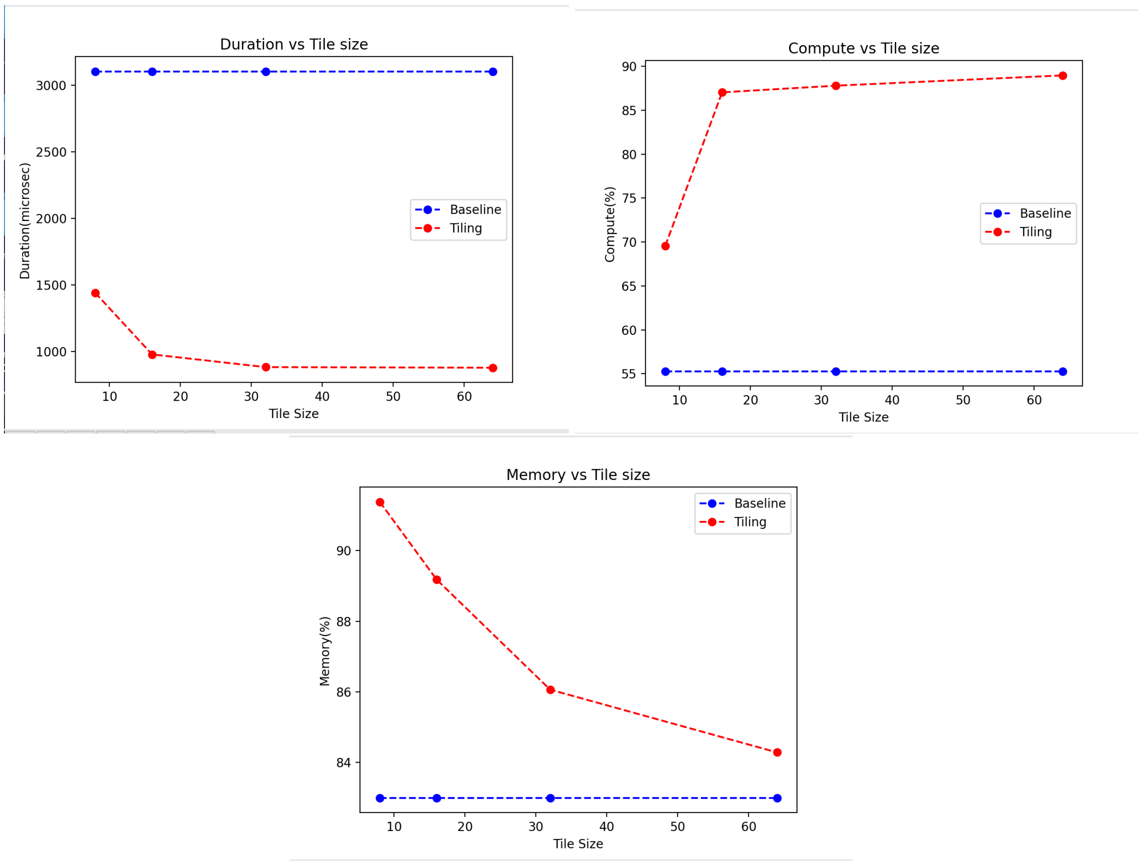


Figure 14: Metrics as a function of Tile size.

### 1.1.5 Tensor Cores

In tensor core, we know that we tradeoff precision with speed. We convert all the arrays to half precision = float 16. This precision loss leads to some off values from correct values and in code we have calculated the average error in each iteration over all values of output. A tiling based implementation was done for this part. As we explored the docs, we observed that only matrix matrix multiplication is supported if we wanted to use the tensor cores. Furthermore the tiling factor used is 16 which has to be kept fixed and cant be changed. This is because tensor cores only support 16*16 matrix multiplication currently. As a result for all the experiments, we kept batch size = 16 or multiple of it. We considered thee following metrics for comparison:

- Duration
- Compute (%)
- Memory (%)

The implementation was such that block dimension across batch size was kept as multiple of warp size which is equal to 32. The grid dimension was accordingly set such that grid dimension $\times$ block dimension = batch size. The second grid and block dimensions were used for Nn channel of output. The following experiments were performed to evaluate the performance in this part:

- Impact of input size: In this experiment we observe the impact of input size. For simplicity Ni = Nn for our case.
- Impact of Batch size In this experiment we observe the impact of Batch size.
- Impact of Block dimension

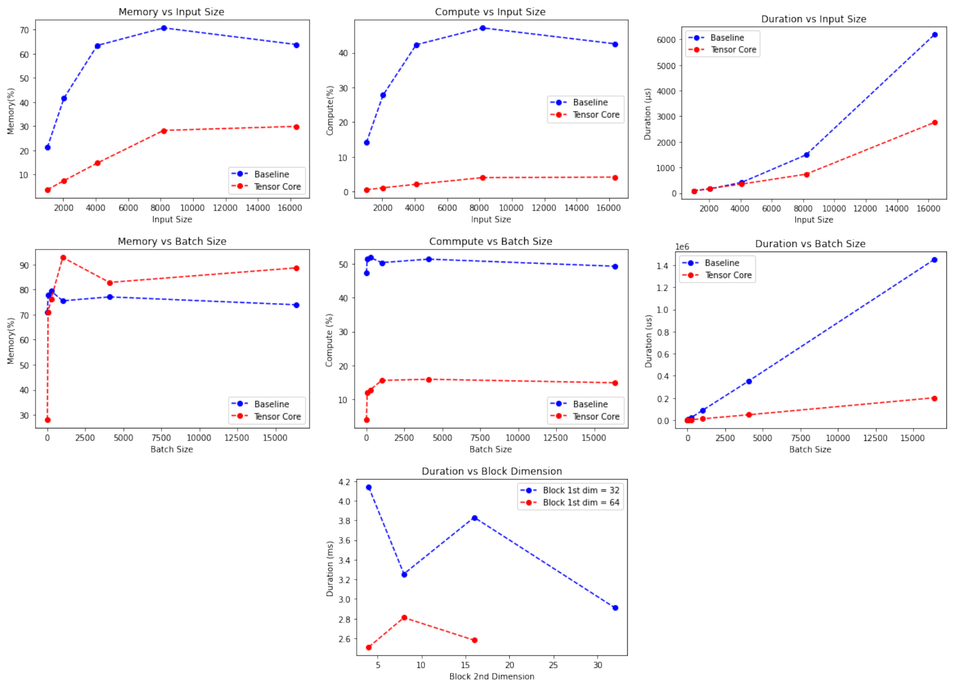


Figure 15: Ablation experiments of duration, memory and compute vs input and batch size when using tensor cores and baseline

The following observations were made:

- With a higher block dimensions gave better results (less duration), this is because, if block dimension is more, then grid dimension is less and hence number of blocks less, leading to less block writing overhead.

- In all cases, tensor core takes less duration over baseline as expected since tensor core is much faster than cuda cores.
- The tensor cores, is highly affected by batch size as we see the memory is highly consumed if bath size increased. This is because, as discussed above, we have parallelized across bath size across tensor cores and hence as batch size increases, the number of blocks consumed also increase. Similarly compute also increases and become saturated after one point.

Since NSIGHT doesnt support roofline charts to be =created for tensor cores, Using the reference mentioned in (1), we calculated the points for x and y axis. Following is the data: For classifier 1:

$$Time = 2281571.264 \times 10^9 \tag{1}$$

$$FLOPs = 6422528 * 512 = 3288334336 \tag{2}$$

DRAM = 206.66 MB, L2 = , l1 = 277.10 MB, 592.79 MB. Hence total data movement:

$$Data - movement = 1076.55MB \tag{3}$$

$$Arithmeticintensity = FLOPs/datamovement = 3.054 \tag{4}$$

$$Performance = FLOPs/time = 1.441TFLOP/sec \tag{5}$$

Hence this point plooted on roofline would lead to point very close to the line, hence it has much more scope to go right and above and performance is not optimal yet. The data reuse is not much, due to various restrictions on tiling factor and hence leads to supoptimal arithmetic intensity, however FLOPs/ sec is very good since it almost touches the slanted line, which is the max achievable FLOP/sec for that arithmetic intensity. Hence implementation has DRAM being limiting factor. All the results are compiled for classifier 1 and batch size = 16.

(1) https://www.nersc.gov/assets/Uploads/RooflineHack-2020-mechanism-v2.pdf

## 1.2 Convolution (normal and with batching)

The convolution operation was implemented such that parallelization was done across three dimensions. The exact parallelization strategy is attached in Q1 of next section. This is a baseline implementation for both batch versions (1 and 16). We conducted the following 4 experiments in this and for each we noted Compute, Memory and Duration. With the undestanding that, a convolution operation can be done by converting it into matrix-matrix multiplication (using the Toeplitz matrix formulation), we can leverage the optimizations implementations discussed before. Hence in this section, we only report the baseline metrics where computation has been done in a traditional manner.

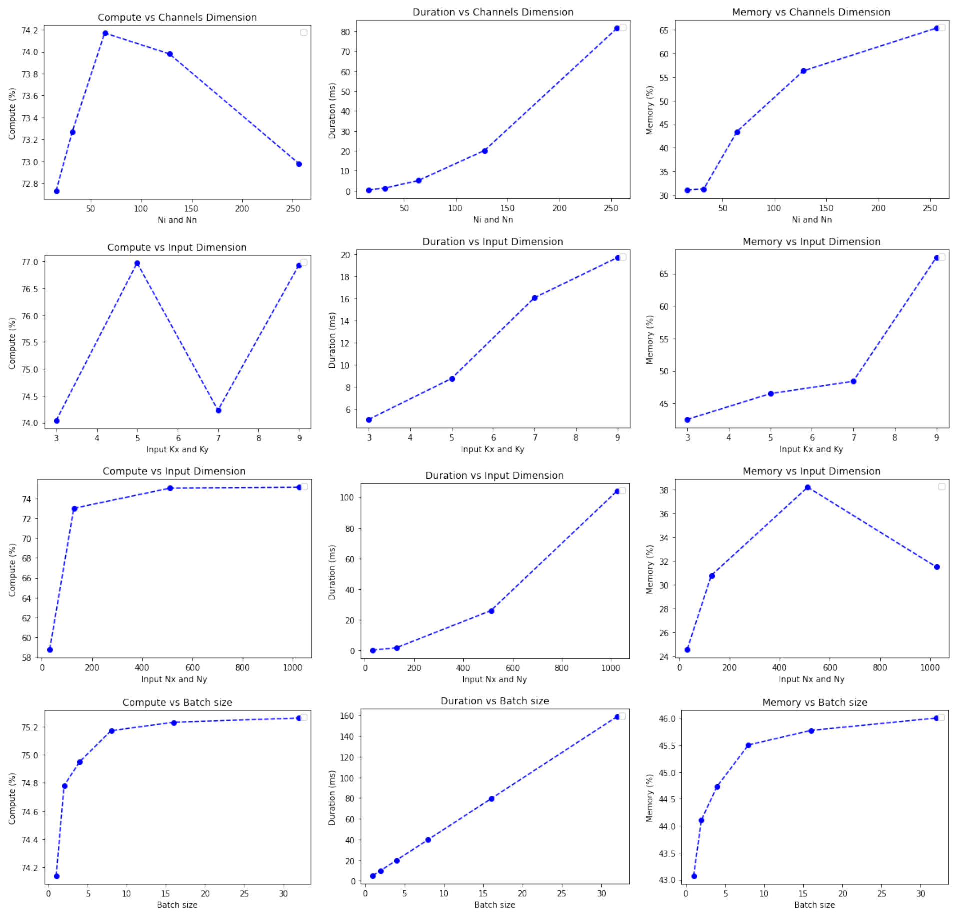


Figure 16: Ablation study for convolution implementation

For each of the experiments, the parameters written at bottom of graph are changed simultaneously and are equal at any point. The following observations are noted:

- If Nn and Ni channels are increased, then compute remains more or less constant, duration as expected is increased and memory percent is increased as work in each thread is increased as number of channels increased (see the parallelization strategy in next section)
- As Filter size is increased, compute again remains more or less constant, duration is increased as expected and memory also increased due to similar reasons as above. Compute remains constant as it has reached its max capacity
- Now as Nx and Ny are increased, across which parallelization is done, we see as expected compute is increased and duration is increased too due to simialr reasons. Memory more or less remains constant as number of blocks and threads only increase with Nx and Ny.

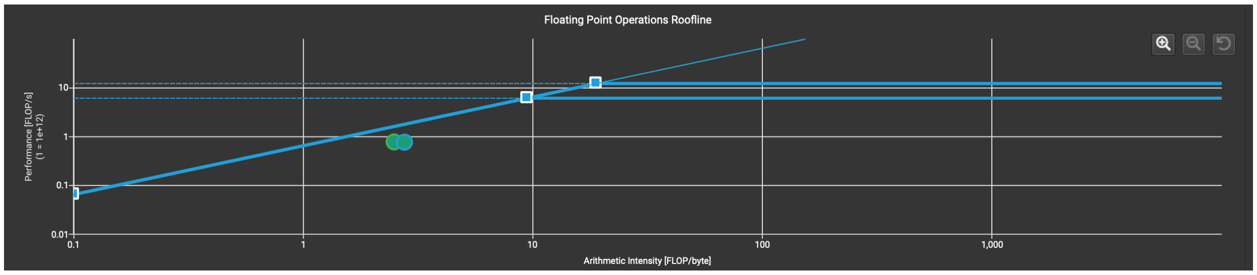


Figure 17: Roofline chart for convolution kernel. Left point is for the batching case = 16, and right point is for batch size = 1

We clearly see that the performnce is sub-optimal and points can be pushed towards right and above. Since this is the baseline implementation, where we have looped over many variables, there is huge data movement. Similarly, very less reuse is these. Furthermore, due to many for loops, the performance is also reduced due to more time taken by each thread. Since this is to left hand side, the implementation is such that DRAM is the limiting factor.

## 2 Questions Answers

- **What was your basic parallelization strategy? Are there limitations in scaling this strategy, or consequences for throughput or latency?**
  If not mentioned, the data is stored in row major format. **DATA LAYOUT:**

  *Baseline:*

```
Synapse: Nn * Ni
Neuron_i: Ni * 1
Neuron_n: Nn * 1
threadsPerBlock: 16
blocksPerGrid: (Nn + threadsPerBlock - 1) / threadsPerBlock
Each thread computes one output element of Neuron_n, i.e. Ni products.
```

  *Batching:*

```
Synapse: Nn * Ni
Neuron_i: Ni * BatchSize
Neuron_n: Nn * BatchSize
threadsPerBlock: 16
Grid Dimension:
    grid_rows: (Nn + threadsPerBlock - 1) / threadsPerBlock
    grid_cols: (BatchSize + threadsPerBlock - 1) / threadsPerBlock
Block Dimension: threadsPerBlock * threadsPerBlock
Each thread computes one output element of Neuron_n, i.e. Ni products.
```

  *Shared Memory (with tiling):*

```
Synapse: Nn * Ni
Neuron_i: Ni * 1
Neuron_n: Nn * 1
threadsPerBlock: BLOCK_SIZE
blocksPerGrid: (Nn + threadsPerBlock - 1) / threadsPerBlock
Each thread computes one output element of Neuron_n, i.e. Ni products.
```

```
Shared Memory: BLOCK_SIZE elements of Synapse and Neuron_i loaded into memory.
Each thread of the block loads one element of each Synapse row and Neuron_i column
and writes one element of Neuron_n.
```

*Shared Memory with Batching:*

```
Synapse: Nn * Ni
Neuron_i: Ni * BatchSize
Neuron_n: Nn * BatchSize
threadsPerBlock: BLOCK_SIZE
Grid Dimension:
    grid_rows: (Nn + threadsPerBlock - 1) / threadsPerBlock
    grid_cols: (BatchSize + threadsPerBlock - 1) / threadsPerBlock
Block Dimension: BLOCK_SIZE * BLOCK_SIZE
Each thread computes one output element of Neuron_n, i.e. Ni products.
Shared Memory: BLOCK_SIZE * BLOCK_SIZE elements of Synapse and Neuron_i loaded into memory
Each thread of the block loads one element of each Synapse row and Neuron_i column
and writes one element of Neuron_n.
```

*Tensor Core (Batching):*

```
Synapse: Ni * Nn
Neuron_i:  BatchSize * Ni
Neuron_n: BatchSize * Nn
threadsPerBlock: (32,32)
Grid Dimension:
    grid.x : (BatchSize + ((tensor_batch * blockDim.x / warpS) - 1))
    / (tensor_batch * blockDim.x / warpS);
    grid.y: (Nn + (tensor_out_dim * blockDim.y) - 1) / (tensor_out_dim * blockDim.y);
All matrices stored as column major format
Here tensor_out_dim and tensor_batch are constants and equal to 16 as this is
the interface which is provided by NVIDIA for using tensor cores
(tiling factor is fixed to 16).
Each thread computes a 16*16 tile of output and store it
in output matrix (The internal implementation is such that warp scheduler
ensures each thread does 4*4 matrix multiplication job)
```

*Convolution (Baseline):*

```
Synapse: Nn * Ni * Ky * Kx
Neuron_i:  BatchSize * Ni * Ny * Nx
Neuron_n: BatchSize * Nn * Ny * Nx
threadsPerBlock: Nx/Sx
Grid Dimension:
    gridDim.x = Ny/Sy;
    gridDim.y = Nn;
    gridDim.z = BatchSize;
Each thread computes one pixel of output. Total threads. = (Nx*Ny*Nn*BatchSize)/(Sx*Sy)
```

In general, in row major data layout, a thread requiring data by travelling row wise rather than travelling columns is beneficial, this is because a warp will schedule many threads at a time and this will access the L1 cache more frequently since all threads will require data column wise leading to more spatial reuse (if all row data are accessed by a single thread, then next thread will access data which is from second row and that will be number of columns length apart, leading to less L1 cache usage).

Apart from this, number of threads per block were decided based on Nx in convolution and if we want to scale, this parallelization needs to be avoided since maximum threads allowed per block = 512. Furthermore, if our parallelization strategy is such that number of blocks increase with input dimensions, then as input increase, a block writing and read overhead will also come specially in shared memory, leading to more latency.

- **What is the execution time of each kernel? (and throughput if you use batching)**

| $Kernel$ | $ExecutionTime(ms)$ |
|---|---|
| Baseline | 2.27 |
| Shared Memory (SM) | 0.682 |
| Batching (16) | 2.96 |
| Batching (16) + SM | 2.17 |
| Batching (16) + Tensor cores | 2.29 |

Table 1: Execution time of Classifier 1.

| $Kernel$ | $ExecutionTime(\mu s)$ |
|---|---|
| Baseline | 193.34 |
| Shared Memory (SM) | 114.18 |
| Batching (16) | 310.78 |
| Batching (16) + SM | 191.36 |
| Batching (16) + Tensor cores | 352.45 |

Table 2: Execution time of Classifier 2.

| $Kernel$ | $ExecutionTime(ms)$ |
|---|---|
| Baseline | 4.9 |
| Baseline (Batching 16) | 77.75 |

Table 3: Execution time of Convolution 1.

| $Kernel$ | $ExecutionTime(ms)$ |
|---|---|
| Baseline | 2.82 |
| Baseline (Batching 16) | 40.68 |

Table 4: Execution time of Convolution 2.

| $Kernel$ | $TFLOP/s$ |
|---|---|
| Baseline | 0.089 |
| Batching (16) | 1.11 |

Table 5: Throughput of Classifier 1.

As we can observe, batching kernel performs approximately 12.5x more operations every second.

| $Kernel$ | $TFLOP/s$ |
|---|---|
| Baseline | 0.043 |
| Batching (16) | 0.428 |

Table 6: Throughput of Classifier 2.

As we can observe, batching kernel performs approximately 10x more operations every second.

- **What do you suspect is the limiting factor for performance of each kernel (compute,dram,scratchpad)? Where does the implementation fall on the roofline model for your particular GPU?**
  The limiting factor along with the roofline model has been presented along with the corresponding kernel in the sections above.

- **How does the implementation compare with CUDNN? (use same batch size in CUDNN)**
  Here SM is shared memory. We report results for our best implementation optinmization which is shared memory.

| $Kernel$ | $m$ | $k$ | $n$ | $Duration(\mu s)$ | $Ops(mill)$ |
|---|---|---|---|---|---|
| CUDNN | 25088 | 4096 | 1 | 746 | 102.76 |
| SM | 25088 | 4096 | 1 | 682 | 490 |
| CUDNN | 25088 | 4096 | 16 | 751 | 1,644.17 |
| SM | 25088 | 4096 | 16 | 2170 | 3291 |
| CUDNN | 4096 | 1024 | 1 | 40 | 4.19 |
| SM | 4096 | 1024 | 1 | 114.18 | 8.344 |
| CUDNN | 4096 | 1024 | 16 | 41 | 67.11 |
| SM | 4096 | 1024 | 16 | 191.36 | 134.6 |

Table 7: CUDNN vs Our Classifier

- **What optimizations did you find most useful/least useful?**
  The better optimization depended on the size of inputs. For very small inputs, especially without batching, we observed that shared memory decreases the duration significantly. However, for very very large inputs, tensor core beats shared memory, this is beacause, tensor core gives correct output with some less precision and thats the tradeoff. Hence both of our optimizations having tiling in them, lead to better performance as compared to baseline and depending on input size, we find different optimizations useful.

- **Turn in the cuda kernel as an attachment in the submission. Don't zip everything together though, make sure source and PDF are turned in separately.**

  How to run:

  - $classifier.cu$: implements the baseline classifier code. While running, edit the Ni, Nn variables according to the required value.
  - $classifier\_SharedMemory.cu$: implements the shared memory classifier code. While running, edit the Ni, Nn and BLOCK_SIZE variables according to the required value.
  - $classifier\_batch.cu$: implements the classifier with batching code. While running, edit the Ni, Nn, BatchSize variables according to the required value.
  - $classifier\_batch\_sm.cu$: implements the shared memory classifier with batching code. While running, edit the Ni, Nn, BLOCK_SIZE and BatchSize variables according to the required value.
  - $tensor - batch.cu$: implements tensor core, can run by following command:

```
nvcc -arch=sm_70 tensor_batch.cu
```

while running the executable, you can pass inputs, specifically Ni, Nn, Batchsize and if bool flag if you want to run the cpu version of our code. For any other file, you can skip arch flag.

- $conv - baseline.cu$: implements baseline of convolution. In this also you can pass inputs to run executable, specifically Ni, Nn, Kx, Ky, Nx, Ny and Batchsize and bool flag if you want to run the cpu version of code. If you dont pass any values then it will run with default values specificed in file.