MINI-PROJECT 2

LEARNING MACHINES- 259

Madhav Sankar Krishnakumar madhavsankar@ucla.edu

Parth Shettiwar parthshettiwar@g.ucla.edu

July 4, 2022

Contents

ABSTRACT

We have used the Classifer Kernel with Shared Memory and batching from Mini-Project 1 as the core model. This is followed by a number of analysis performed in the Titan V GPU. We have presented these results to understand the optimal hyperparameters for our model and possible limitations. After this we have designed an execution time approximator based on Bottleneck Analysis to find the optimal ranges of hardware parameters. These provide valuable insights on how the base hardware can be modified to extract the best performance for our model. We have summarized these results at the end.

1 Approach

Using [1],[2],[3],[4] we first noted the various Titan V hardware parameter values.

```
Boost_Clock = 1455 MHz
Streaming Multiprocessors = 80
Shared Memory Size = 96KB (equivalent to L1 cache)
Single Precision Performance = 13.8 TFLOPS
Double Precision Performance = 6.9 TFLOPS
Half Precision Performance = 27.6 TFLOPS
Integer Performance = 55.2 TFLOPS
Main memory bandwidth = 15.754 GB/sec (PCI-Express 3.0 x16 interface)
Main memory latency = 3 us
```

Now we calculate the L2 to L1 bandwidth as follows:

$$128Bline, 32ways => 4096bytes/cycle \tag{1}$$

$$4096bytes/cycle * Boostclock = 5.96 \times 10^{12} \tag{2}$$

Hence we get the L2 to L1 bandwidth. = 5.96×10^{12} We also compute the Boot time which we find out by running on tetracosa titan V, by seeing time taken other than kernel calculations (basically when no calculation is present to do in kernel, this is the minimum time required)

```
Boot_Time = 1e-6
```

Having derived the various hardware parameters, we followed the following approach to do the analysis:

- We took the classifier kernels used in Mini-project 1 and performed the roofline analysis for batched and unbatched versions. After seeing the point on roofline model, we further optimize it to shared memory model and plot it again on roofline plot, seeing it has shifted more to right, implying better utilization of resources.
- After this, we perform the analysis on this optimized model where we vary first the hyperparameters and parameters of model. In this we consider varying batch size, block size and precision levels.
- We finally consider varying various hardware parameters found above and see the optimum levels and requirements and whether the current GPU Titan V satisfies the hardware levels to achieve the optimal performance or not. We also do the analysis where we find whether each precision and kernel size is memory, buffer or compute bound. The bottleneck analysis is performed for this section. The execution time is computed as the maximum of the above three times. As part of buffer bandwidth bound time, we compute the L2-L1 time to carry the data. This depends on the L2-L1 bandwidth which we computed before. For tiling and baseline classifier, we find the effective data usage and data size to be carried.

Few of the challenges which we faced were finding the exact constants defined above. Specifically finding the latency time for main memory was tricky. Furthermore to perform the analysis over other hardware architectures was infeasible since again finding the constants was a tricky job. Formulating the bottleneck analysis was an interesting part where we tried not to make our analysis complicated but at same time cover all hardware parameters. The various insights from this analysis pave the way for future GPU models to develop domain specific architectures.

2 Kernels

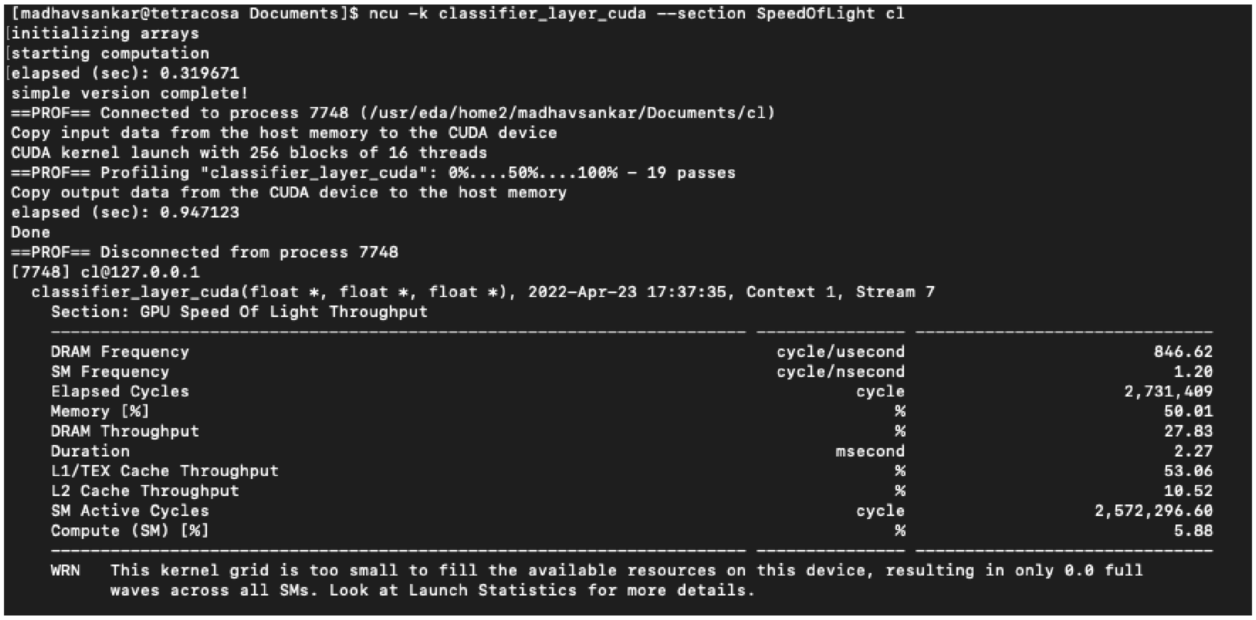
2.1 Classifier

Two Classifiers were used for all the models:

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

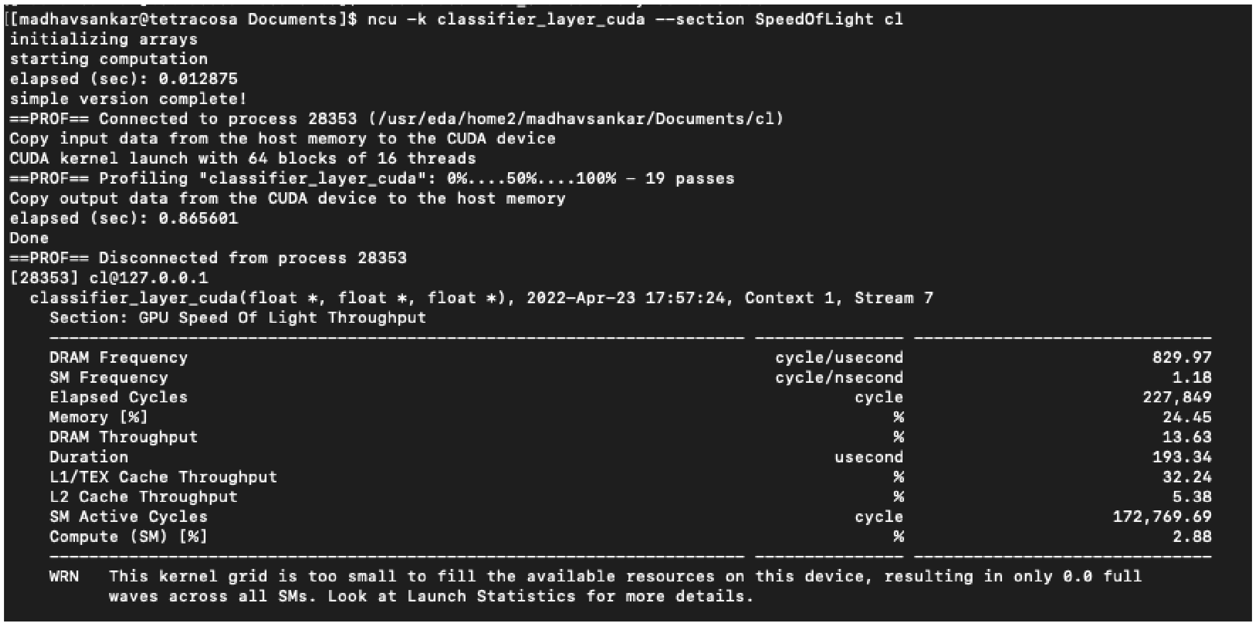
2.1.1 Baseline Model

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.
```

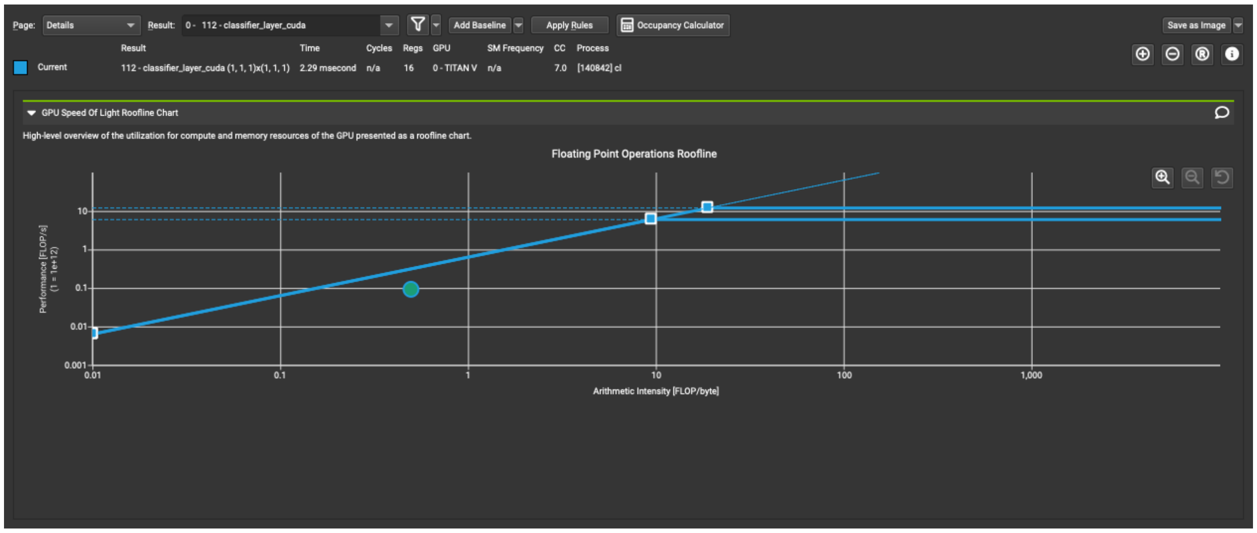
Figure 1: Baseline Model for Classifier 1



```
[[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:



	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

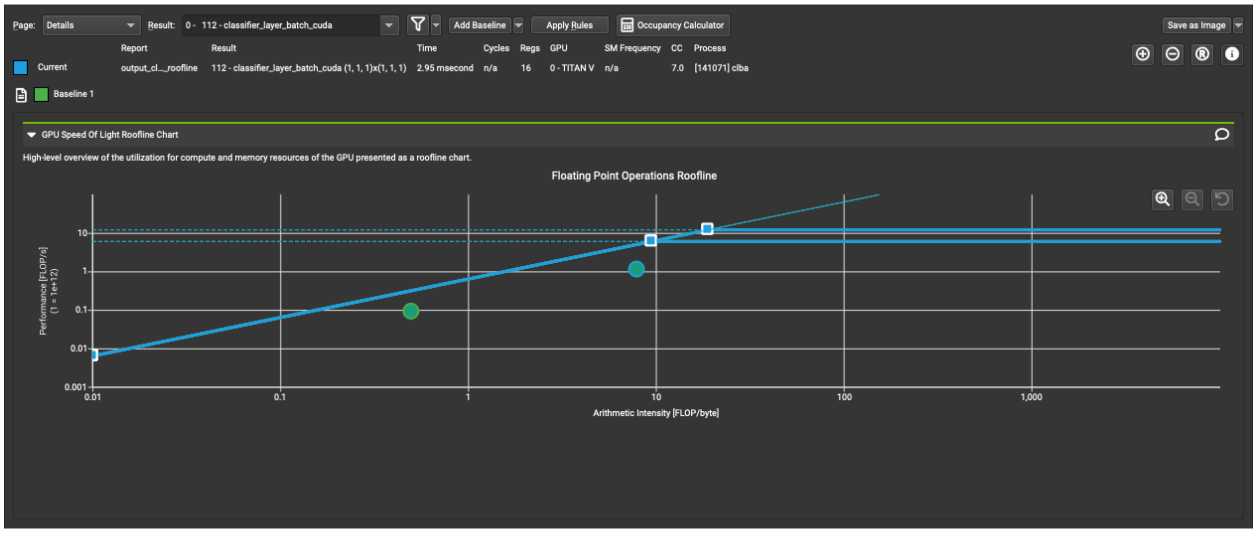
High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

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.

2.1.2 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.

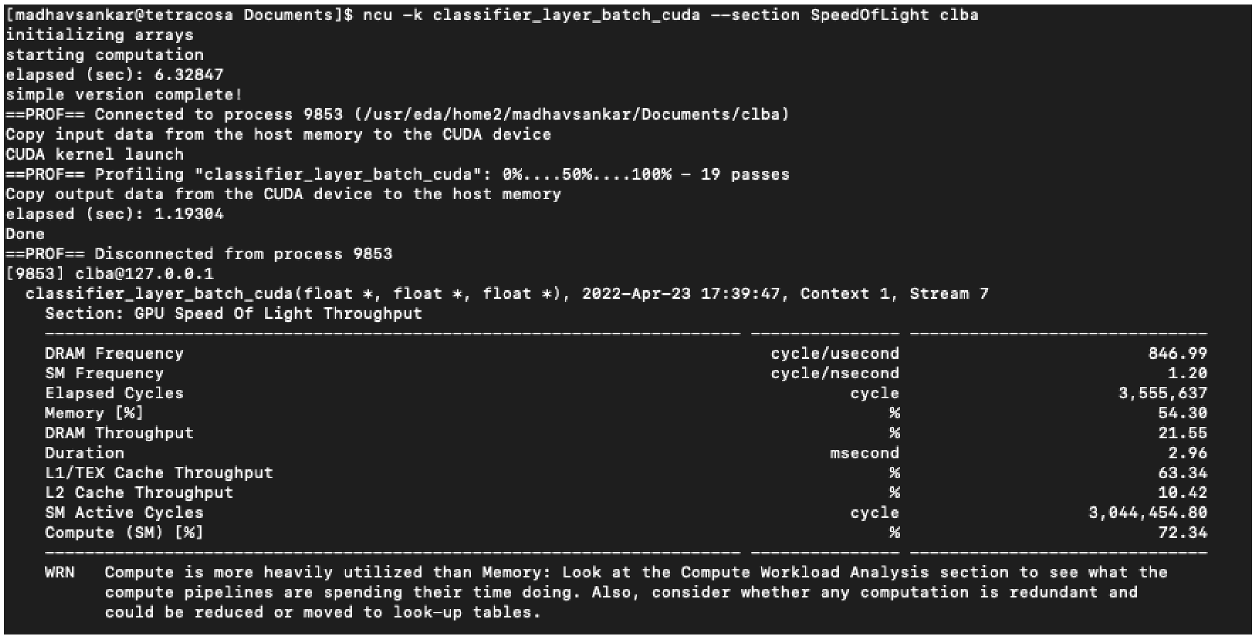


	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	output_cl..._roofline	112 - classifier_layer_batch_cuda (1, 1, 1)x(1, 1, 1)	2.95 msecond	n/a	16	0 - TITAN V	n/a	7.0	[141071] clba
🟩 Baseline 1									

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

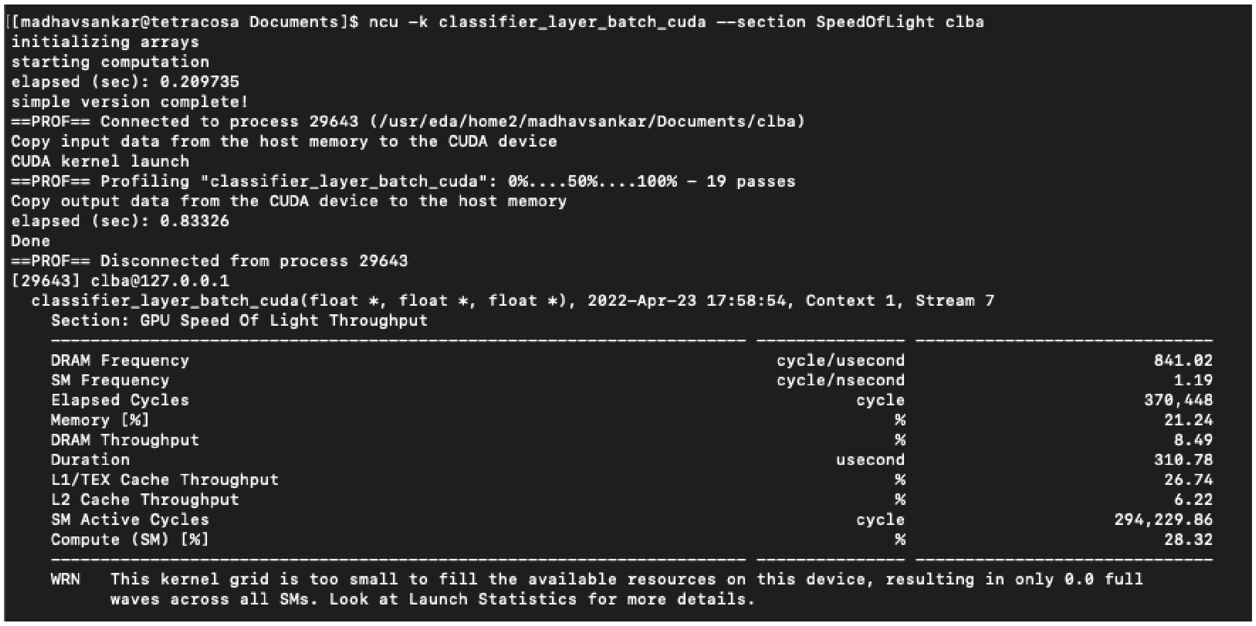
Figure 4: Roofline Chart for Batching as compared to the baseline model.

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

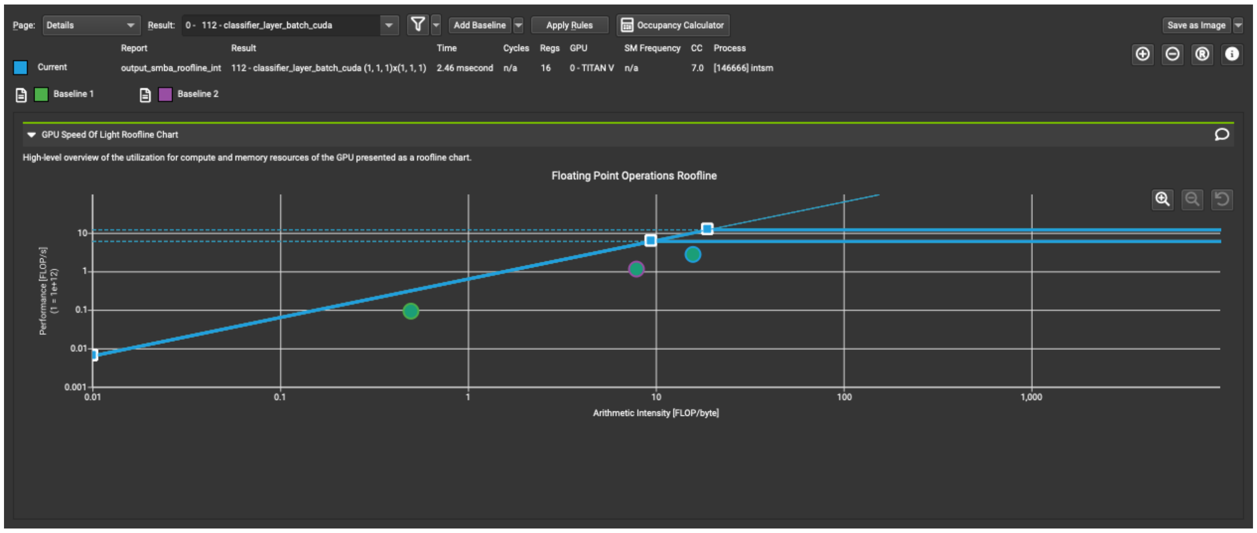


```
[[madhavsankar@tetracosa Documents]$ ncu -k classifier_layer_batch_cuda --section SpeedOfLight clba
initializing arrays
starting computation
elapsed (sec): 0.209735
simple version complete!
==PROF== Connected to process 29643 (/usr/eda/home2/madhavsankar/Documents/clba)
Copy input data from the host 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

2.1.3 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.



Page: Details Result: 0 - 112 - classifier_layer_batch_cuda

	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

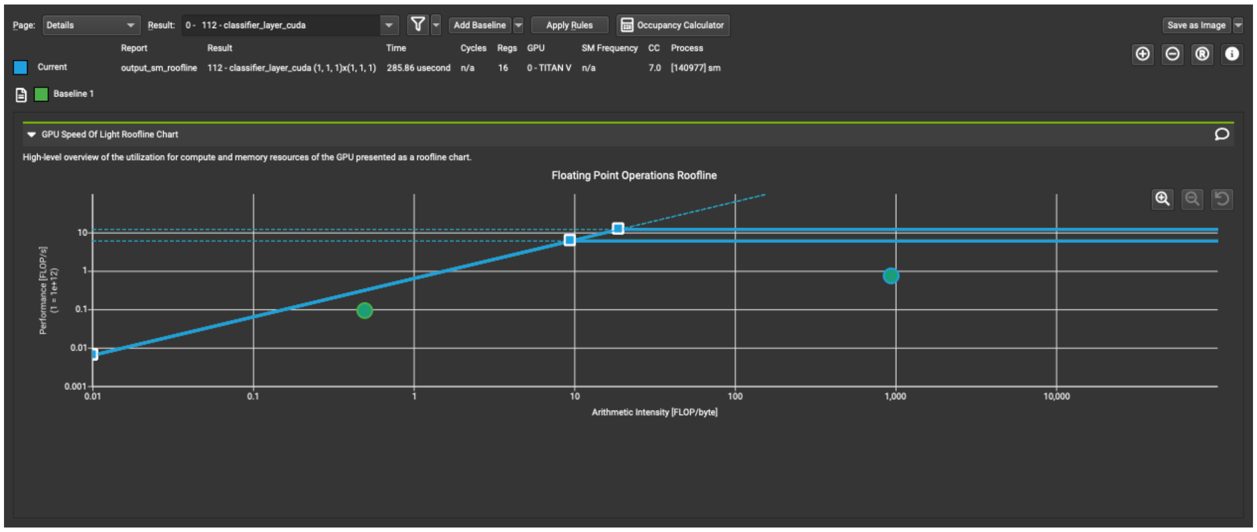
🟩 Baseline 1 🟪 Baseline 2

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
GPU Speed Of Light Roofline Chart
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 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.

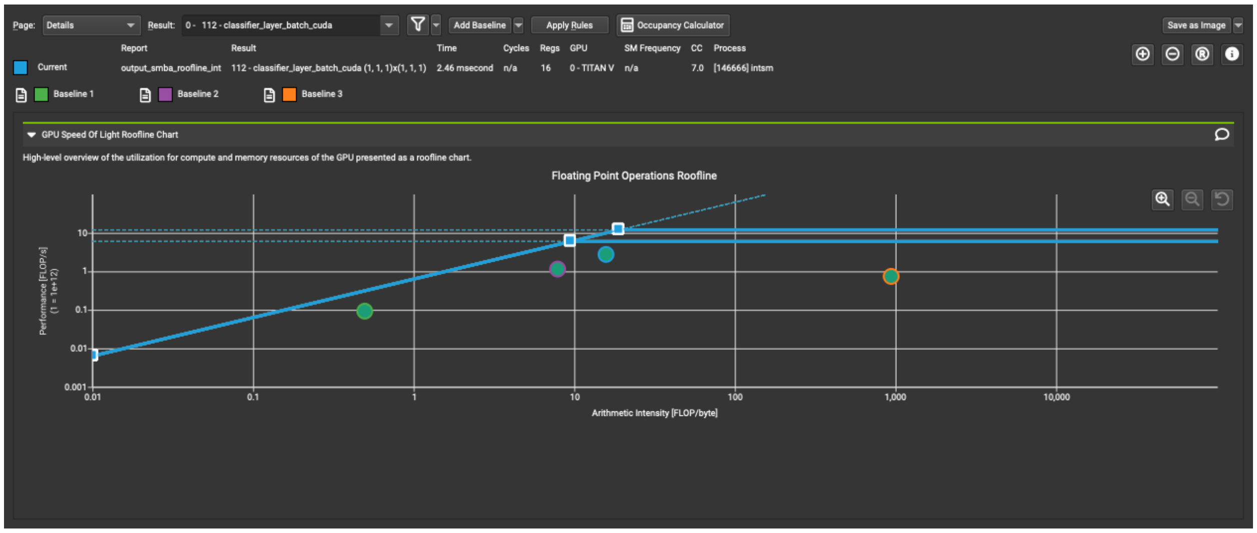


	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	output_sm_roofline	112 - classifier_layer_cuda (1, 1, 1)x(1, 1, 1)	285.86 usecond	n/a	16	0 - TITAN V	n/a	7.0	[140977] sm
🟩 Baseline 1									

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]

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]

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:

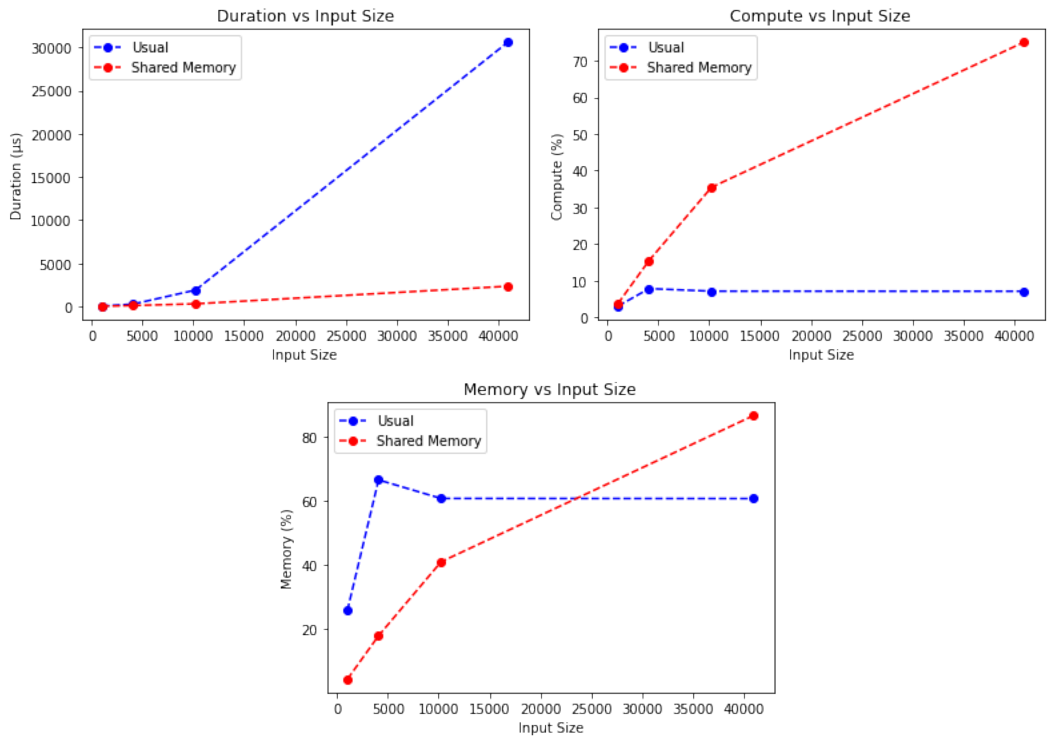


Figure 10: Metrics as a function of Input Size.

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:

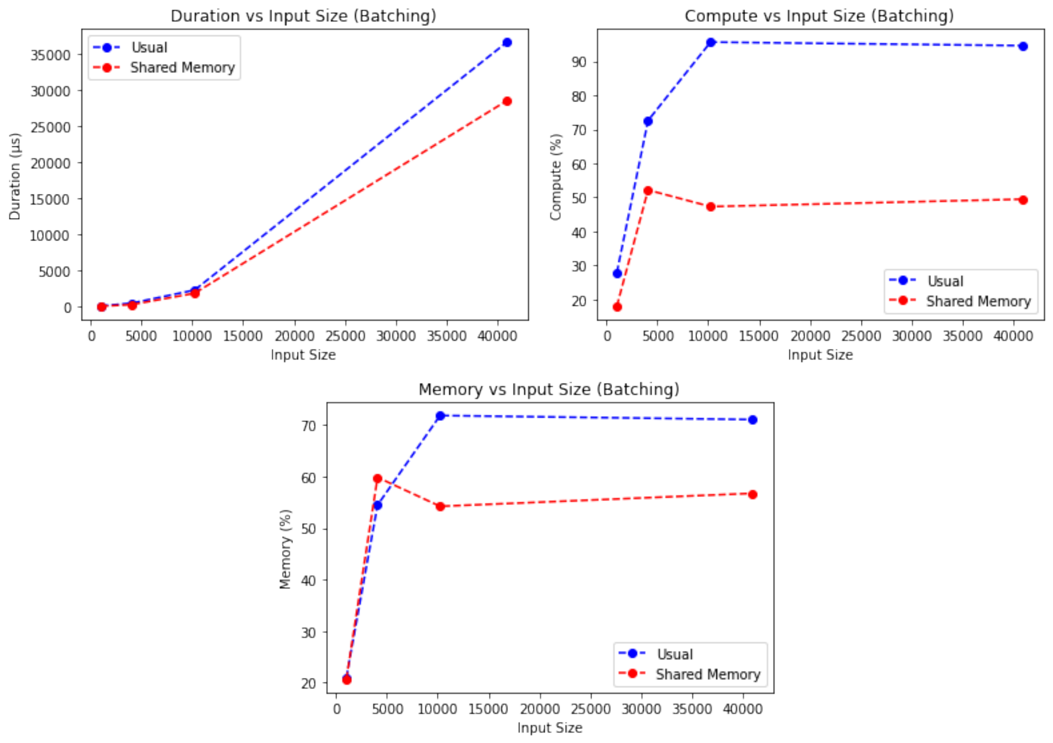


Figure 11: Metrics as a function of Input Size.

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:

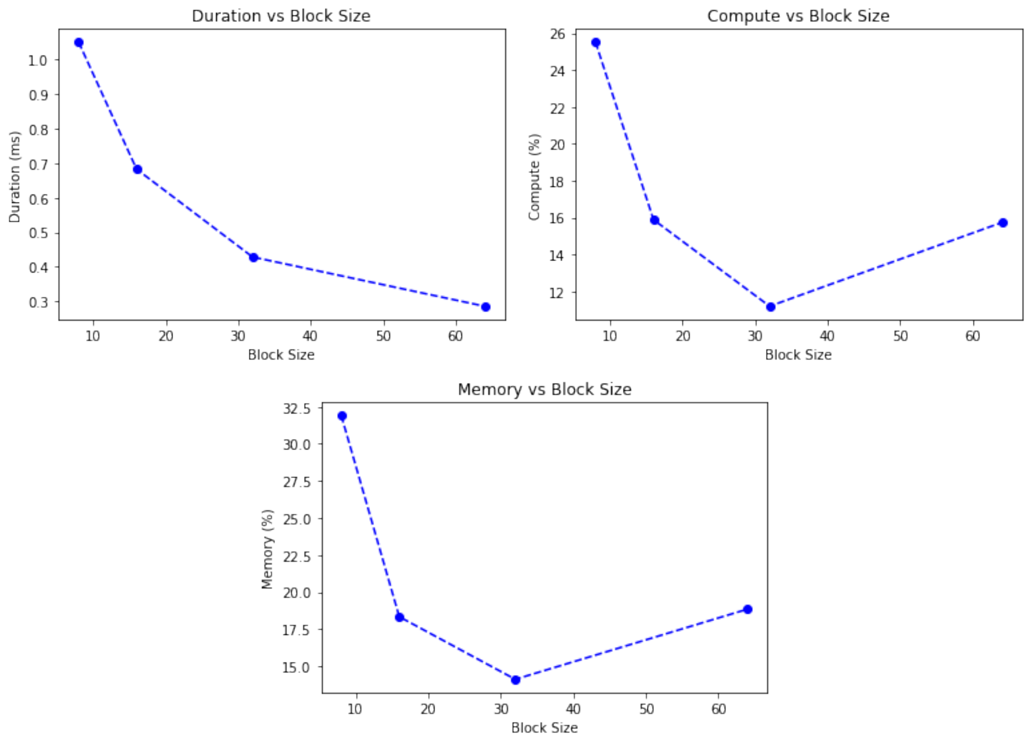


Figure 12: Metrics as a function of Block Size.

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:

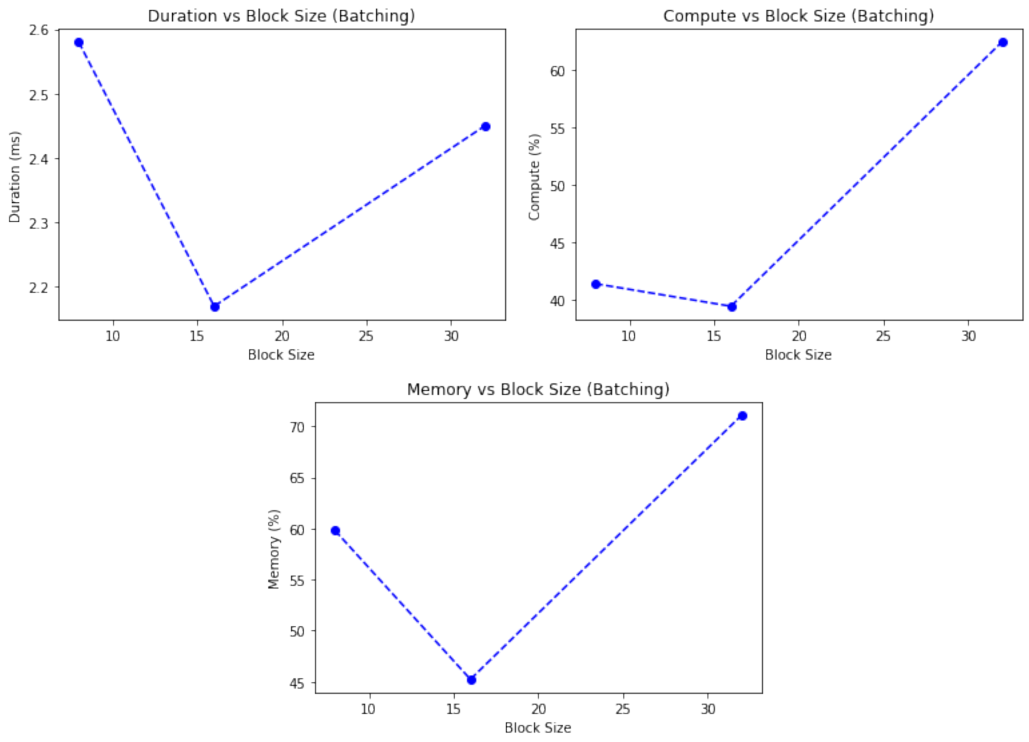


Figure 13: Metrics as a function of Block Size.

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

3 Bottleneck Analysis:

3.1 Varying input sizes:

We have performed the Bottleneck Analysis on a number of different input sizes and have tabulated the expected execution time and whether it is memory-bound (M), compute-bound(C) or buffer-bound(B).

$Precision$	m	k	n	$BoundedTime$	Bound
FP64	25088	4096	1	0.006528050731269841	M
FP64	25088	4096	2	0.006529903683650793	M
FP64	25088	4096	4	0.006533609588412698	M
FP64	25088	4096	8	0.006541021397936508	M
FP64	25088	4096	16	0.006555845016984127	M
FP64	25088	4096	32	0.006585492255079365	M
FP64	25088	4096	64	0.009122243779865771	B
FP64	25088	4096	128	0.01824327155973154	B
FP64	25088	4096	256	0.03648532711946308	B
FP64	25088	4096	512	0.07296943823892617	B
FP64	4096	1024	1	0.00026835498523809526	M
FP64	4096	1024	2	0.0002686800646031746	M
FP64	4096	1024	4	0.00026933022333333335	M
FP64	4096	1024	8	0.0002706305407936508	M
FP64	4096	1024	16	0.0002732311757142857	M
FP64	4096	1024	32	0.00027843244555555556	M
FP64	4096	1024	64	0.000375650877852349	B
FP64	4096	1024	128	0.000750085755704698	B
FP64	4096	1024	256	0.0014989555114093961	B
FP64	4096	1024	512	0.002996695022818792	B

Table 1: Varying input sizes for FP64.

$Precision$	m	k	n	$BoundedTime$	Bound
FP32	25088	4096	1	0.003264887810634921	M
FP32	25088	4096	2	0.003265814286825397	M
FP32	25088	4096	4	0.0032676672392063493	M
FP32	25088	4096	8	0.003271373143968254	M
FP32	25088	4096	16	0.003278784953492064	M
FP32	25088	4096	32	0.0032936085725396827	M
FP32	25088	4096	64	0.003323255810634921	M
FP32	25088	4096	128	0.004561729889932886	B
FP32	25088	4096	256	0.009122243779865771	B
FP32	25088	4096	512	0.01824327155973154	B
FP32	4096	1024	1	0.00013503993761904764	M
FP32	4096	1024	2	0.00013520247730158732	M
FP32	4096	1024	4	0.00013552755666666668	M
FP32	4096	1024	8	0.0001361777153968254	M
FP32	4096	1024	16	0.00013747803285714286	M
FP32	4096	1024	32	0.00014007866777777779	M
FP32	4096	1024	64	0.00014527993761904762	M
FP32	4096	1024	128	0.00018843343892617452	B
FP32	4096	1024	256	0.000375650877852349	B
FP32	4096	1024	512	0.000750085755704698	B

Table 2: Varying input sizes for FP32.

$Precision$	m	k	n	$BoundedTime$	Bound
FP16	25088	4096	1	0.0016333063503174605	M
FP16	25088	4096	2	0.0016337695884126985	M
FP16	25088	4096	4	0.0016346960646031747	M
FP16	25088	4096	8	0.0016365490169841272	M
FP16	25088	4096	16	0.001640254921746032	M
FP16	25088	4096	32	0.0016476667312698414	M
FP16	25088	4096	64	0.0016624903503174606	M
FP16	25088	4096	128	0.0016921375884126986	M
FP16	25088	4096	256	0.002281472944966443	B
FP16	25088	4096	512	0.004561729889932886	B
FP16	4096	1024	1	6.838241380952381e-05	M
FP16	4096	1024	2	6.846368365079365e-05	M
FP16	4096	1024	4	6.862622333333333e-05	M
FP16	4096	1024	8	6.895130269841269e-05	M
FP16	4096	1024	16	6.960146142857142e-05	M
FP16	4096	1024	32	7.090177888888888e-05	M
FP16	4096	1024	64	7.35024138095238e-05	M
FP16	4096	1024	128	7.870368365079365e-05	M
FP16	4096	1024	256	9.482471946308725e-05	B
FP16	4096	1024	512	0.00018843343892617452	B

Table 3: Varying input sizes for FP16.

$Precision$	m	k	n	$BoundedTime$	Bound
INT8	25088	4096	1	0.0008175156201587301	M
INT8	25088	4096	2	0.0008177472392063491	M
INT8	25088	4096	4	0.0008182104773015872	M
INT8	25088	4096	8	0.0008191369534920634	M
INT8	25088	4096	16	0.0008209899058730159	M
INT8	25088	4096	32	0.0008246958106349206	M
INT8	25088	4096	64	0.0008321076201587301	M
INT8	25088	4096	128	0.0008469312392063492	M
INT8	25088	4096	256	0.0009542400371014493	C
INT8	25088	4096	512	0.0019072640742028987	C
INT8	4096	1024	1	3.505365190476191e-05	M
INT8	4096	1024	2	3.509428682539683e-05	M
INT8	4096	1024	4	3.517555666666667e-05	M
INT8	4096	1024	8	3.533809634920635e-05	M
INT8	4096	1024	16	3.566317571428571e-05	M
INT8	4096	1024	32	3.6313334444444445e-05	M
INT8	4096	1024	64	3.76136519047619e-05	M
INT8	4096	1024	128	4.0214286825396826e-05	M
INT8	4096	1024	256	4.5415556666666665e-05	M
INT8	4096	1024	512	7.898538666666667e-05	C

Table 4: Varying input sizes for INT8.

3.1.1 Observations:

- The duration increases proportionally as we increase precision.

- Most of the moodels are memory bound. Only as we increase batching to larger sizes, we observe that the compute or buffer times exceeding the memory time.
- For INT, as each data item is 8 bits, the memory and buffer times decrease. This is why we observe compute bound models only in this case.

3.2 Precision:

We have performed the Bottleneck Analysis on a number of different precision types. We have used four classifier models for the same. Classifier 1 has m = 25088, k = 4096 and batch size of 16. Classifier 2 has m = 4096, k = 1024 and batch size of 16. Classifier 3 has m = 25088, k = 4096 and batch size of 512. Classifier 4 has m = 4096, k = 1024 and batch size of 512.

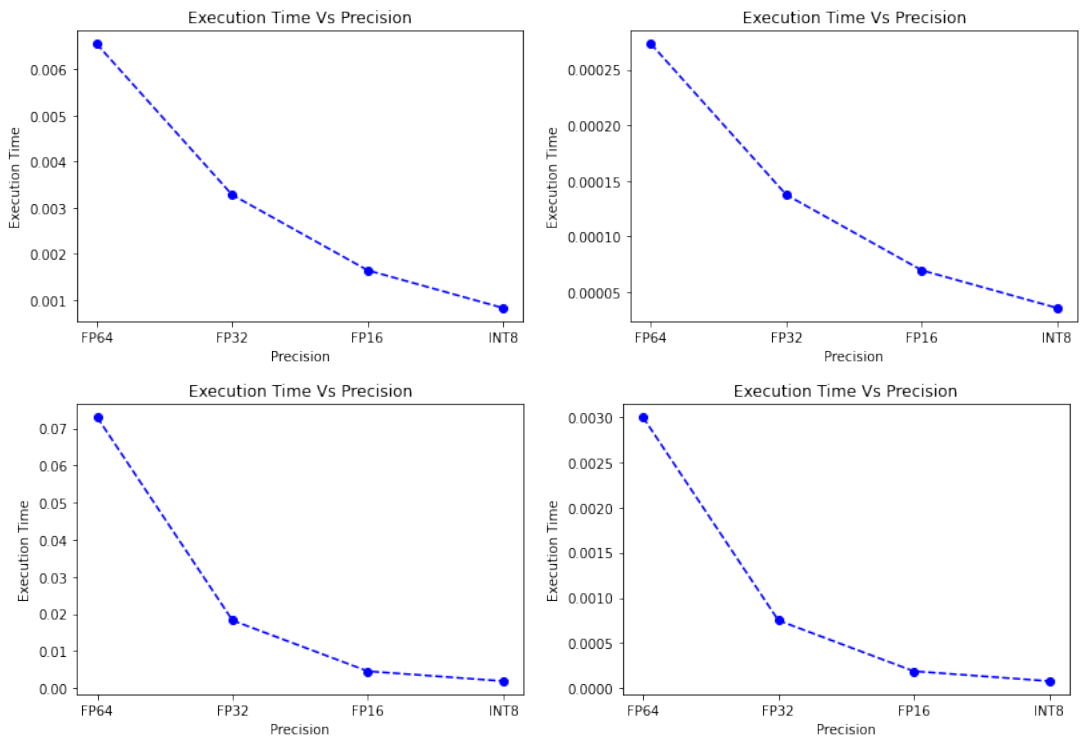


Figure 14: Execution Time as a function of Precision.

We obtained the corresponding error for the various precisions by running the evaluations on Titan V. The results are as follows.

$Precision$	m	k	n	$RMSError$
FP64	25088	4096	1	6.64575e-06
FP32	25088	4096	1	3.75105e-05
FP16	25088	4096	1	0.0042997
INT8	25088	4096	1	13.3623
FP64	25088	4096	16	6.53115e-06
FP32	25088	4096	16	3.76165e-05
FP16	25088	4096	16	0.00435934
INT8	25088	4096	16	13.1814
FP64	4096	1024	1	1.75076e-06
FP32	4096	1024	1	6.10597e-06
FP16	4096	1024	1	0.00176083
INT8	4096	1024	1	5.4058
FP64	4096	1024	16	1.7096e-06
FP32	4096	1024	16	6.03281e-06
FP16	4096	1024	16	0.00176695
INT8	4096	1024	16	5.35969

Table 5: Accuracy Comparison

3.2.1 Observations:

- For the first two models we observe a proportional decrease in time. This is because both these models were memory bound. Therefore as we decrease precision from 32 to 16 bits, time halves.
- For the other two models, we see a more steep fall as these are buffer bound. So a decrease in the precision by half makes the execution time one-fourth.
- Combining the execution times and accuracies, **FP16 seems to provide the best balance between high accuracy and low execution times.**

3.3 Precision Performance:

We have performed the Bottleneck Analysis on a number of different precision performance levels. We have used these classifier models. Classifier 1: m = 25088, k = 4096 and batch size of 512 [Buffer-Bound]. Classifier 2: m = 25088, k = 4096 and batch size of 16 [Memory-Bound]. We tried across a few models and results are fairly consistent across models.

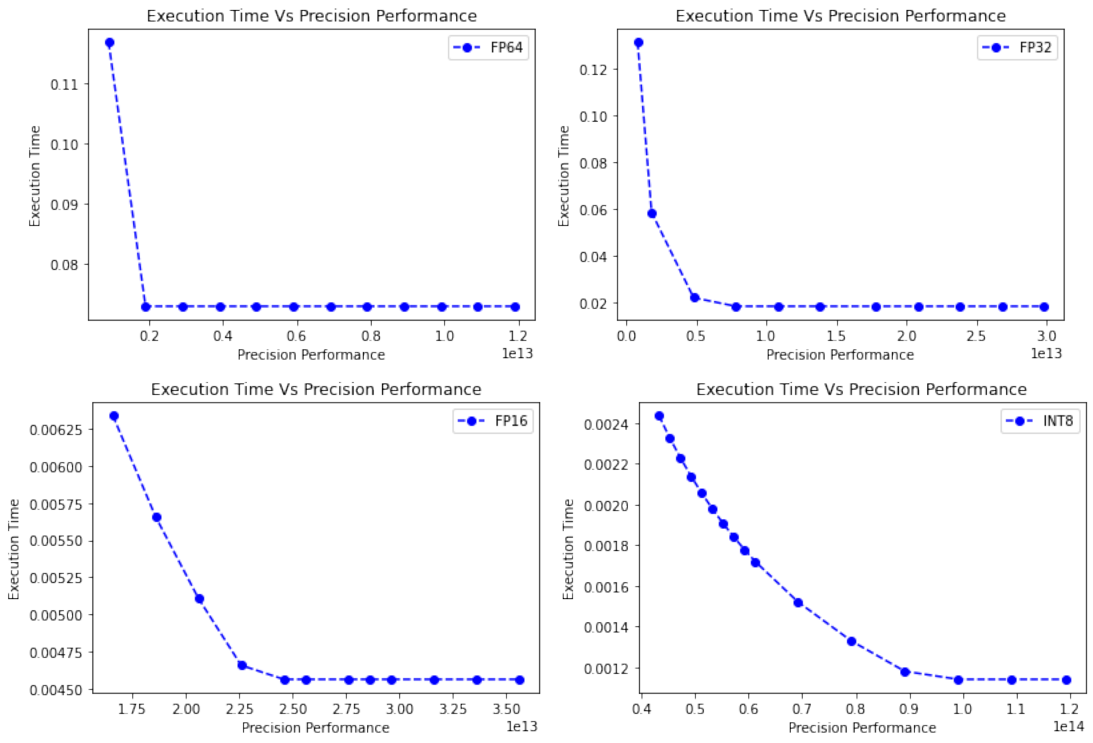


Figure 15: Execution Time as a function of Precision Performance (Buffer-bound except INT8 which is Compute Bound).

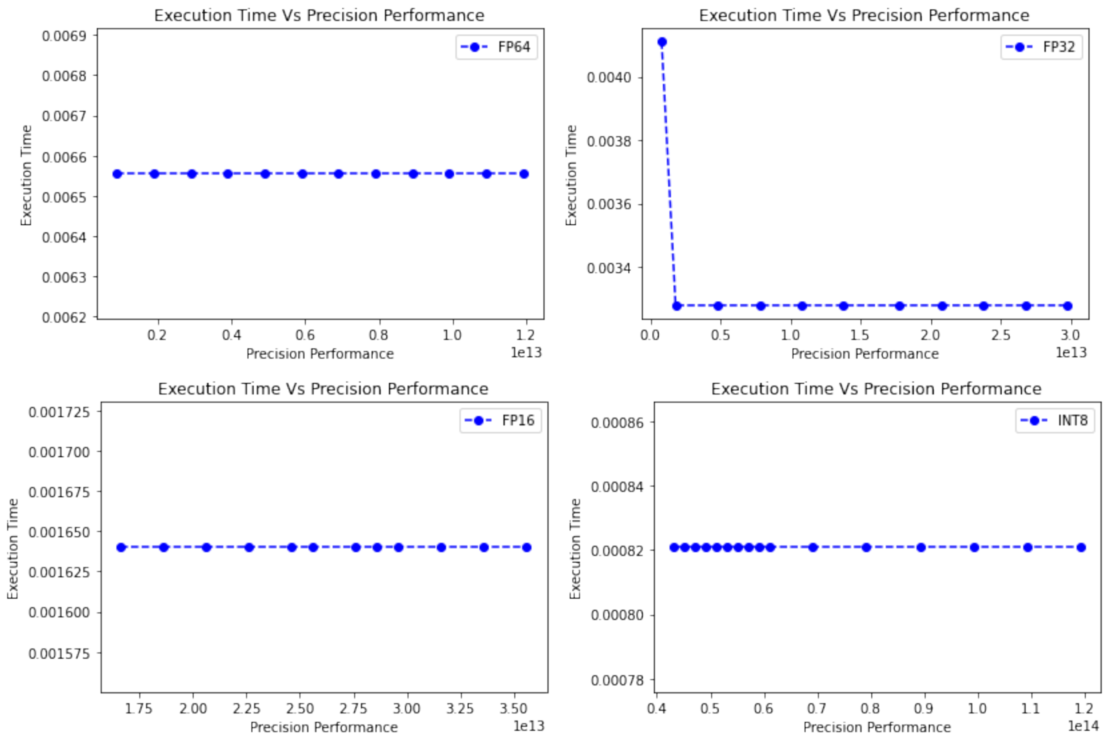


Figure 16: Execution Time as a function of Precision Performance (Memory-bound).

The values of Titan V are:

```
Double_Precision_Performance =  6900000000000    # 6.9 TFLOPS
Single_Precision_Performance = 13800000000000    # 13.8 TFLOPS
Half_Precision_Performance =   27600000000000    # 27.6 TFLOPS
Integer_INT8_Performance =     55200000000000    # 55.2 TOPS
```

3.3.1 Observations:

- For buffer-bound models, we observe an exponential drop in execution time with increasing precision performance. But after a point, the difference is negligible as buffer is no longer the bottleneck.
- We observe that for all the floating point precisions, the precision performance of Titan V falls under the optimum region. But this is not the case for INT8 as this is Compute Bound. For Compute-Bound, we observe that the execution time can be dropped to a much larger extent, even halved. Therefore, one of the insights we obtain is that **Increasing the Integer Performance, will lead to much greater performance.**
- For memory-bound models, unless the precision performance is extremely low, we do not see much of an impact on the execution times.

3.4 Shared Memory Size:

We have performed the Bottleneck Analysis on a number of different shared memory sizes. We tried across a few models and results are fairly consistent across models. We have decided the tiling size to be the maximum tile size that can be accommodated in the shared memory.

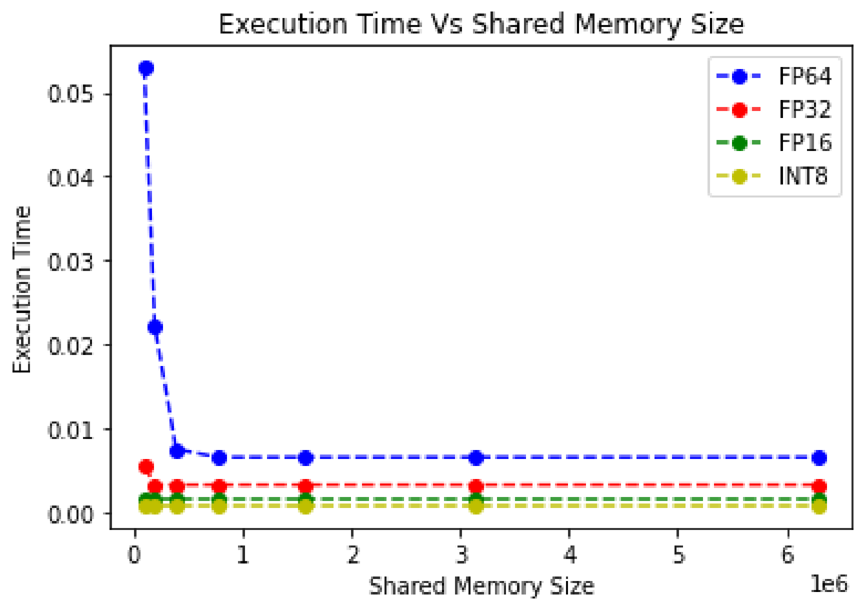


Figure 17: Execution Time as a function of Shared Memory Size

3.4.1 Observations:

- As the shared memory size, increases we do observe a decrease in execution time.
- For smaller sizes, as we increase, we see a drastic drop in execution time but the improvement decreases with increasing size. So for larger sizes, we do not see much performance change.

- We observe that by Shared Memory size of 384KB, the execution time converges. Titan V has a Shared Memory Size of 96KB, so we get the insight that **Increasing the Shared Memory Size by a factor of 4, decreases execution time by a factor of 5.**

3.5 Clock Rate:

We have performed the Bottleneck Analysis on a number of different Clock Rate. We tried across a few models and results are fairly consistent across models.

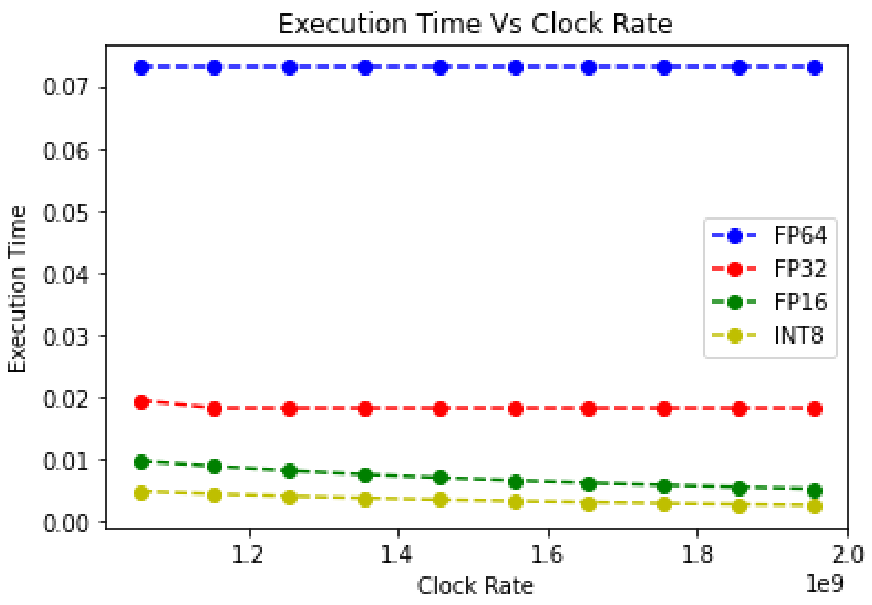


Figure 18: Execution Time as a function of Clock Rate

3.5.1 Observations:

- The effect of clock rate on Execution Time is fairly low as most of our models were not compute-bound and even if they were, they became memory or stream bound when compute execution time dropped.
- **This implies that the Titan V's 1455MHz Clock Rate is sufficient for our kernel.**

3.6 Number of Streaming Processors:

We have performed the Bottleneck Analysis on a number of different number of Streaming Processors. We tried across a few models and results are fairly consistent across models.

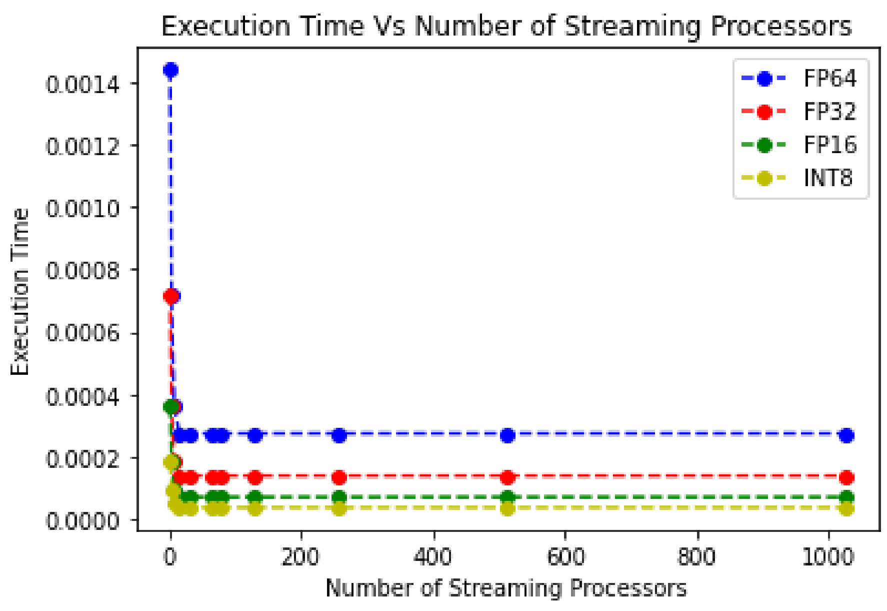


Figure 19: Execution Time as a function of Number of SPs

3.6.1 Observations:

- For smaller numbers, as we increase, we see a drastic drop in execution time but the improvement decreases with increasing numbers. So for larger SPs, we do not see much performance change.
- **This implies that the Titan V's 80 SPs are sufficient for our kernel.**

3.7 L2-L1 Bandwidth

In this experiment, we did the bottleneck analysis as we vary the L2-L1 bandwidth. The variation was in increasing factors of 2, starting with $\frac{TitanV bandwidth}{16}$

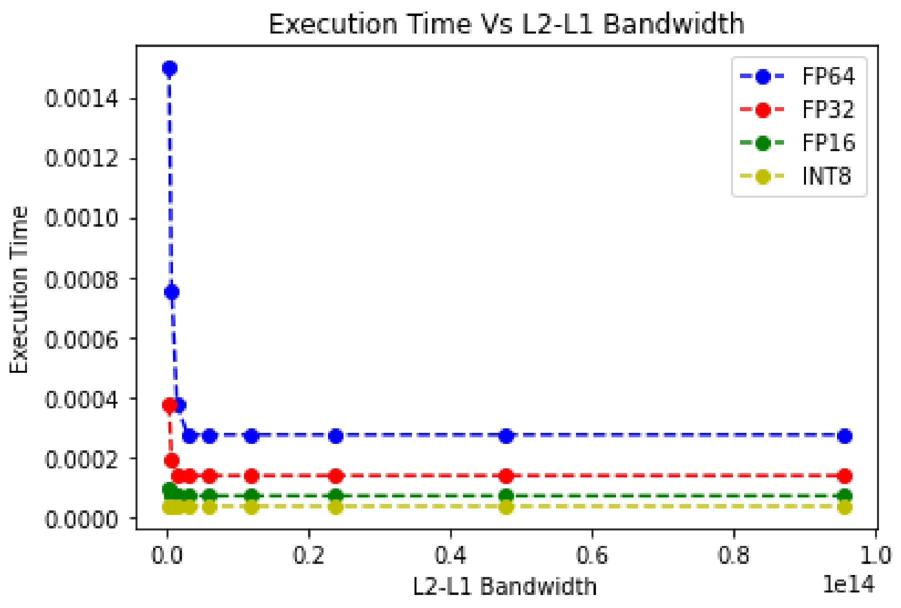


Figure 20: Execution time as a function of L2-L1 Bandwidth

3.7.1 Observations:

- For smaller numbers, as we increase, we see a drastic drop in execution time but the improvement decreases with increasing numbers. So for larger bandwidths, we do not see much performance change.
- **This implies that the Titan V's L2-L1 bandwidth of 5.96 TB/sec is sufficient for our kernel.**

3.8 Main Memory Bandwidth

In this experiment, we did the bottleneck analysis as we vary the main memory bandwidth, starting with $\frac{TitanV memorybandwidth}{8}$ and increasing by factors of 2.

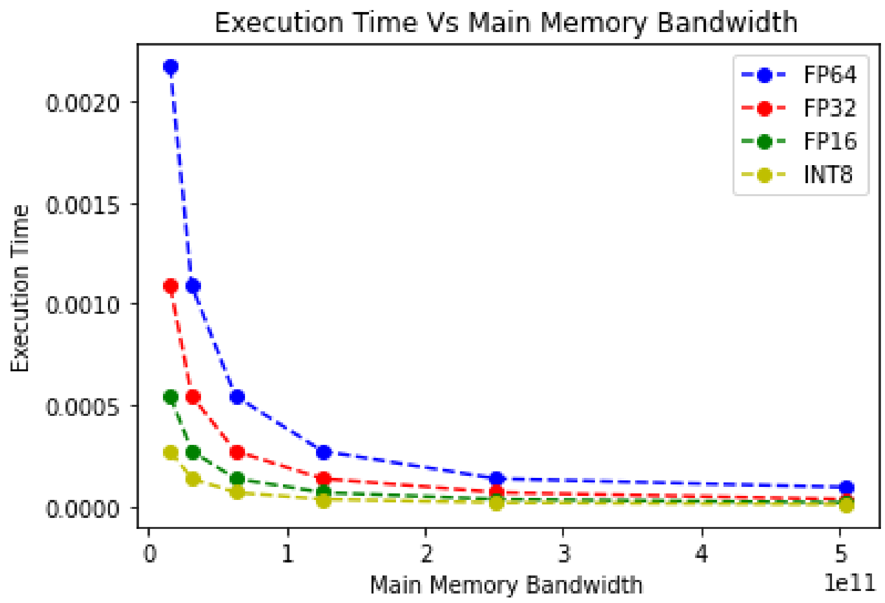


Figure 21: Execution Time as a function of Main Memory bandwidth

3.8.1 Observations:

- Again, we see an exponential drop in execution time at start, but saturates at higher memory bandwidths.
- **Titan V's 80 main memory bandwidth of 15.754 is not sufficient an if increased by 4x times, then optimal execution time will be reached (an increase of 4x in Main memory bandiwdth results in 4x reduction in execution time approximately)**

4 Summary of Analysis

4.1 Hyperparameters:

- Batching helps to bring the model more towards compute-intensive. This enables to perform more classifications of inputs at almost the same time and improves throughput.
- Shared Memory Block Size of 16 provides the least execution time.
- FP16 seems to provide the best balance between high accuracy and low execution times.

4.2 Hardware Parameters:

- Increasing the Integer Performance will lead to much greater performance.
- Increasing the Shared Memory Size by a factor of 4, decreases execution time by a factor of 5.
- Titan V's Clock Rate is optimum for our model.
- Titan V's 80 Streaming Processors are sufficient for our kernel.
- Titan V's L2-L1 bandwidth is sufficient for our kernel.
- Increase of 4x in Main memory bandwidth results in 4x reduction in execution time approximately.

5 How to run:

- $classifier\_batch\_sm.cu$: implements the shared memory classifier with batching code. While running, edit the Ni, Nn, BLOCK_SIZE and BatchSize variables according to the required value.
- $Bottleneck\_analysis$: This is a python notebook that can be run directly. The classifier parameters can be modified by changing the parameters passed while creating the object of type classifierLayer. Similarly hardware parameters can be changed in the GPUHardware object creation. Tile Dimension can be passed as a parameter for Bottleneck_tiling.

References

[1] `https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive`.

[2] `https://hothardware.com/reviews/nvidia-titan-v-volta-gv100-gpu-review`.

[3] `https://www.techpowerup.com/gpu-specs/titan-v.c3051`.

[4] `https://www.pugetsystems.com/labs/hpc/PCIe-X16-vs-X8-with-4-x-Titan-V-GPUs-for-Machine-Learning-1167`.