# Assignment 2

## 1. Programming Problems - Matrix Transpose, Matrix Multiplication (80 pts)

### 1.1 (10 pts): Matrix Transpose

* (5 pts) Implement two parallel transpose algorithms using both PyCUDA and PyOpenCL. The input is a 2D matrix with any size, the output is the transpose of the input.
* (1 pts) Implement a serial transpose algorithm. The input is a 2D matrix with any size, the output is the transpose of the input.
* (2 pts) Choose any two integers M and N from 1 to 10. Then, randomly generate matrices with size M x N, 2M x 2N, 3M x 3N,.... Calculate the transpose of them using 3 transpose algorithms (2 parallel, 1 serial) respectively. Record the running time of each call for each of the algorithm.
* (2 pts) Plot running time vs. matrix size in one figure, **compare and analyze** the results in your report. You need to plot one for only kernel execution and another for including memory copy time (both host to device & device to host) in each of PyCUDA.py and PyOpenCL.py

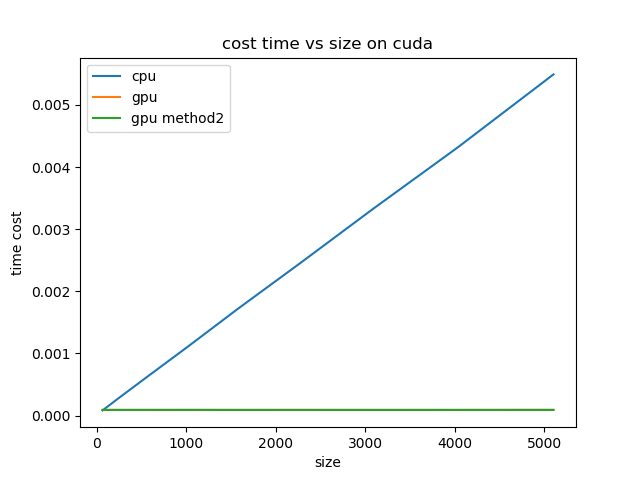
Report:

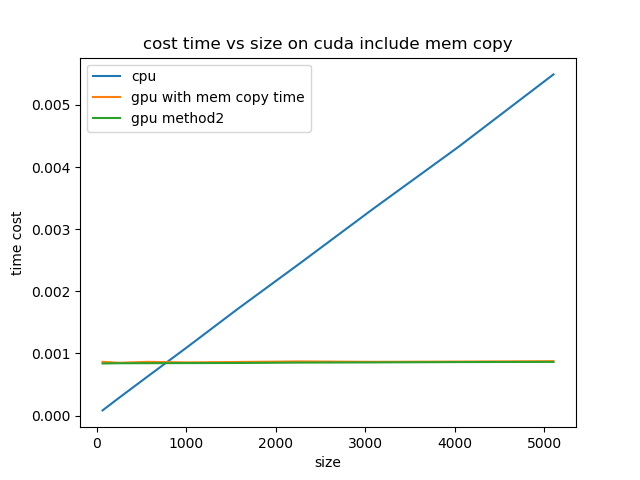
In the first task to transpose an any-size matrix, I implemented one serial and two GPU algorithm in the class Transpose. Two GPU methods are called transpose and transpose2. The serial (CPU) one is called transpose\_serial. I firstly implemented these two GPU algorithms in CUDA, then translated to OpenCL.

To help myself do record running time, I set an additional argument called flag for every method. This flag is set default as 0, this will let the method return the result matrix only. If set it to some number other than 0, these methods will also return the running time. Then I can record this. In this assignment, I record the averaged running time and record them in a CSV file.

The plot is using same method to generate just as the last assignment.

For CUDA:

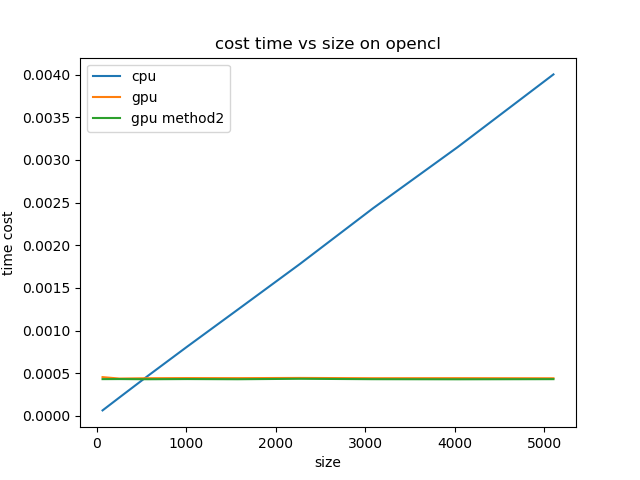


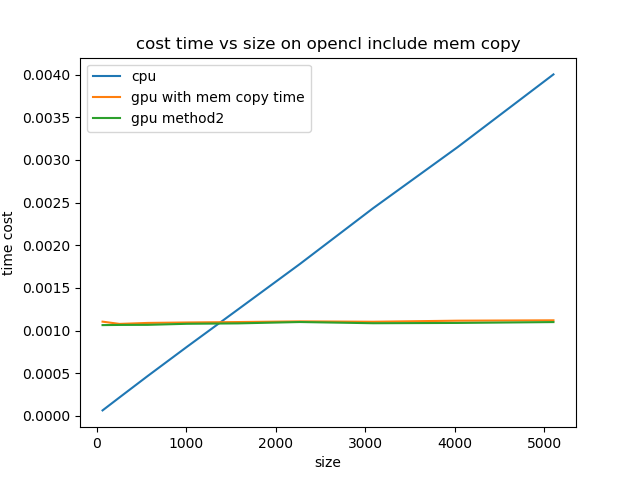


And the recorded data looks like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | time serial | time | time2 | time include mem | time2 include mem |
| 1 | 8.28E-05 | 9.03E-05 | 9.03E-05 | 0.00086403 | 0.00086403 |
| 2 | 0.00028998 | 9.12E-05 | 9.12E-05 | 0.00084901 | 0.00084901 |
| 3 | 0.00062895 | 9.08E-05 | 9.08E-05 | 0.00086403 | 0.00086403 |
| 4 | 0.00109971 | 9.11E-05 | 9.11E-05 | 0.00085521 | 0.00085521 |
| 5 | 0.00171709 | 9.11E-05 | 9.11E-05 | 0.00086224 | 0.00086224 |
| 6 | 0.00244856 | 9.13E-05 | 9.13E-05 | 0.00087102 | 0.00087102 |
| 7 | 0.00332654 | 9.08E-05 | 9.08E-05 | 0.00086445 | 0.00086445 |
| 8 | 0.00431436 | 9.08E-05 | 9.08E-05 | 0.0008685 | 0.0008685 |
| 9 | 0.00549042 | 9.18E-05 | 9.18E-05 | 0.000875 | 0.000875 |

For OpenCL:





The recorded data:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | time serial | time | time2 | time include mem | time2 include mem |
| 1 | 6.37E-05 | 0.00045371 | 0.00045371 | 0.00110404 | 0.00110404 |
| 2 | 0.00021537 | 0.00043702 | 0.00043702 | 0.00107757 | 0.00107757 |
| 3 | 0.00046635 | 0.00044092 | 0.00044092 | 0.00108902 | 0.00108902 |
| 4 | 0.00081031 | 0.00044266 | 0.00044266 | 0.00109442 | 0.00109442 |
| 5 | 0.0012455 | 0.00044203 | 0.00044203 | 0.00109935 | 0.00109935 |
| 6 | 0.00177932 | 0.00044417 | 0.00044417 | 0.0011073 | 0.0011073 |
| 7 | 0.00243354 | 0.00044099 | 0.00044099 | 0.00110292 | 0.00110292 |
| 8 | 0.00314999 | 0.00044171 | 0.00044171 | 0.00111532 | 0.00111532 |
| 9 | 0.00400241 | 0.00044036 | 0.00044036 | 0.00111969 | 0.00111969 |

We can see the CPU method’s cost time goes proportional with matrix size as expected. Two GPU methods shows no effect on time cost as matrix becoming larger. And they have very similar performance. Moreover, we can see time cost on memory operation cost a lot of time that is much more than actual kernel execution time.

### 1.2 (70 pts): Matrix Multiplications

To simplify the algorithm, the input is only **one** M x N matrix. The output is the multiplication of the input matrix and its transpose. The result should be a M x M matrix.

* (20 pts) Implement a naïve version of the matrix multiplication algorithm, in each of PyCUDA & PyOpenCL.
* (15 pts) Optimize the algorithms using shared memory.
* (15 pts) Optimize the algorithms further using any method.
* (10 pts) Choose any two integers M and N from 1 to 10. Then, randomly generate matrices with size M x N, 2M x 2N, 3M x 3N,.... Calculate the multiplication of them and their transpose using your 6 multiplication algorithms (2 naïve, 2 optimized, 2 further optimized) respectively. Record the running time of each call for each algorithm.
* (8 pts, for CUDA only) Using the Nvidia Visual Profiler (nvvp) to compare your 3 algorithms written in CUDA.
* (2 pts) Plot all running time-matrix size profile in one figure, compare and analyze the results in your report. Insert the profiling results of NVVP into your report.You need to plot one for only kernel execution and another for including memory copy time(both host to device & device to host) in each of PyCUDA.py and PyOpenCL.py

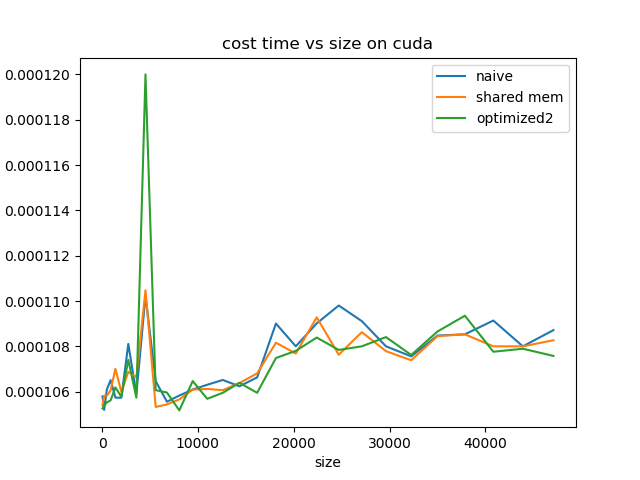
Report:

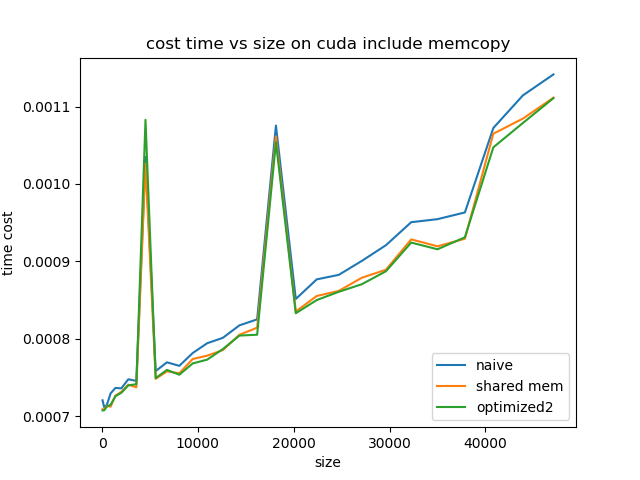
Matrix multiplication is a little complicated than transpose. There are three methods in class MatrixMultiply: matrix\_mul\_naive, matrix\_mul\_optimized1, and matrix\_mul\_optimized2.

Naïve version is quite straight forward, just use block to handle any-size matrix and take care of matrix boundary problem. Then the optimized with shared memory one use shared block memory to accelerate the algorithm just as told in the lecture. Then I use global memory coalescing accessing to accelerate tile block accessing and avoided shared memory bank conflict. This is further optimized algorithm.

I used same procedure to record time cost and produce figures.

For CUDA:



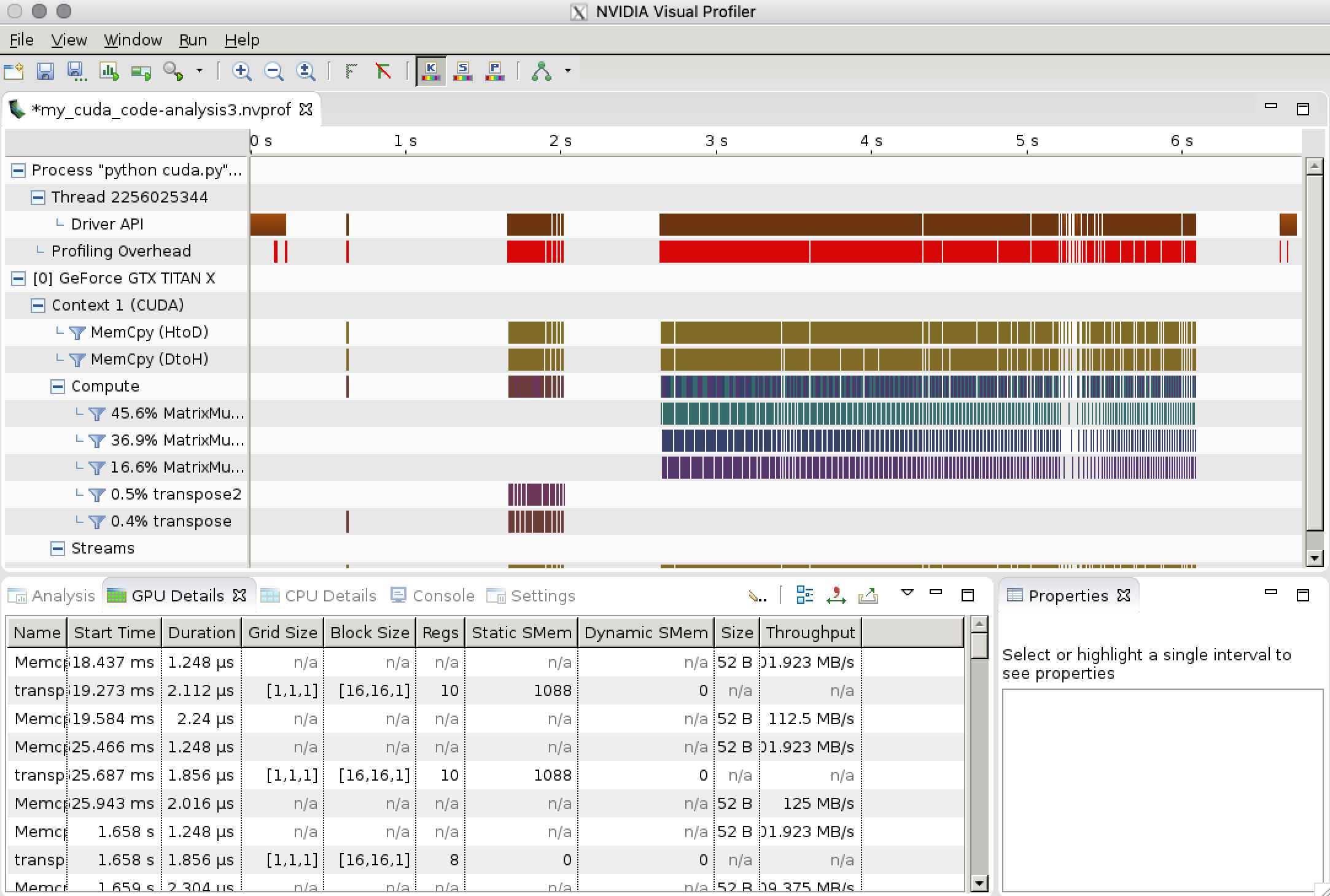


|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | time | time optm1 | time optm2 | time include mem | time optm1 include mem | time optm2 include mem |
| 1 | 0.0001058 | 0.00010541 | 0.00010526 | 0.00072046 | 0.000709 | 0.00070744 |
| 2 | 0.0001052 | 0.00010583 | 0.00010538 | 0.00071179 | 0.0007084 | 0.00070749 |
| 3 | 0.0001061 | 0.00010583 | 0.00010553 | 0.00071446 | 0.00071332 | 0.00071239 |
| 4 | 0.00010651 | 0.00010606 | 0.00010562 | 0.0007294 | 0.00071219 | 0.0007147 |
| 5 | 0.00010574 | 0.00010701 | 0.00010619 | 0.00073647 | 0.00072654 | 0.00072575 |
| 6 | 0.00010574 | 0.00010599 | 0.00010578 | 0.00073591 | 0.00073131 | 0.00073017 |
| 7 | 0.00010811 | 0.00010688 | 0.00010741 | 0.00074762 | 0.0007407 | 0.00073979 |
| 8 | 0.00010589 | 0.00010664 | 0.00010574 | 0.00074553 | 0.00073719 | 0.00074128 |
| 9 | 0.00011024 | 0.00011048 | 0.00012 | 0.00103489 | 0.00102609 | 0.00108282 |
| 10 | 0.00010647 | 0.00010533 | 0.00010607 | 0.00075825 | 0.00074824 | 0.00074914 |
| 11 | 0.00010556 | 0.00010544 | 0.00010596 | 0.00076954 | 0.00075785 | 0.00075974 |
| 12 | 0.00010583 | 0.00010566 | 0.00010518 | 0.000765 | 0.00075534 | 0.00075352 |
| 13 | 0.0001061 | 0.0001061 | 0.00010647 | 0.00078157 | 0.00077385 | 0.00076813 |
| 14 | 0.00010631 | 0.00010613 | 0.00010569 | 0.00079423 | 0.00077813 | 0.00077302 |
| 15 | 0.00010652 | 0.00010607 | 0.00010595 | 0.00080112 | 0.00078535 | 0.00078658 |
| 16 | 0.00010624 | 0.00010638 | 0.00010639 | 0.0008173 | 0.00080511 | 0.00080408 |
| 17 | 0.00010663 | 0.00010681 | 0.00010595 | 0.00082502 | 0.0008143 | 0.00080519 |
| 18 | 0.00010901 | 0.00010816 | 0.00010749 | 0.00107535 | 0.00106149 | 0.00105378 |
| 19 | 0.000108 | 0.00010769 | 0.0001078 | 0.00085137 | 0.00083521 | 0.00083293 |
| 20 | 0.00010902 | 0.00010929 | 0.00010839 | 0.0008767 | 0.00085523 | 0.00084996 |
| 21 | 0.00010981 | 0.00010763 | 0.00010784 | 0.00088252 | 0.00086178 | 0.00086063 |
| 22 | 0.00010912 | 0.00010863 | 0.000108 | 0.00090054 | 0.00087878 | 0.00087044 |
| 23 | 0.000108 | 0.00010779 | 0.00010841 | 0.00092086 | 0.00088918 | 0.00088722 |
| 24 | 0.00010756 | 0.00010739 | 0.00010762 | 0.00095055 | 0.00092835 | 0.00092421 |
| 25 | 0.00010848 | 0.00010845 | 0.00010866 | 0.00095448 | 0.00091941 | 0.00091556 |
| 26 | 0.00010853 | 0.00010853 | 0.00010935 | 0.00096314 | 0.00092916 | 0.00093102 |
| 27 | 0.00010914 | 0.000108 | 0.00010777 | 0.00107222 | 0.00106487 | 0.00104724 |
| 28 | 0.000108 | 0.000108 | 0.0001079 | 0.00111431 | 0.00108433 | 0.00107869 |
| 29 | 0.00010872 | 0.00010827 | 0.00010758 | 0.00114147 | 0.00111139 | 0.00111079 |

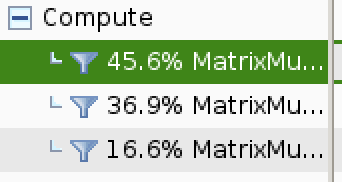
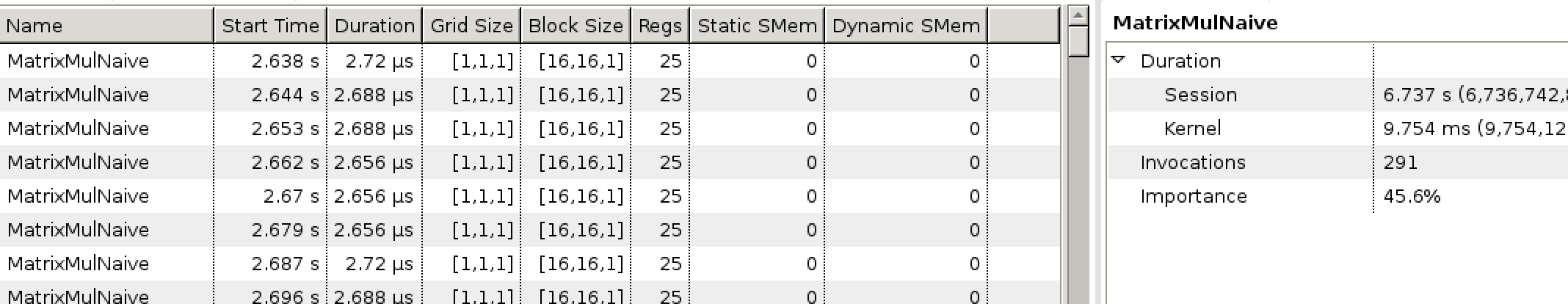
We can see the time cost is as our expectation. The performance of further optimized one is better than shared memory one and naïve version.

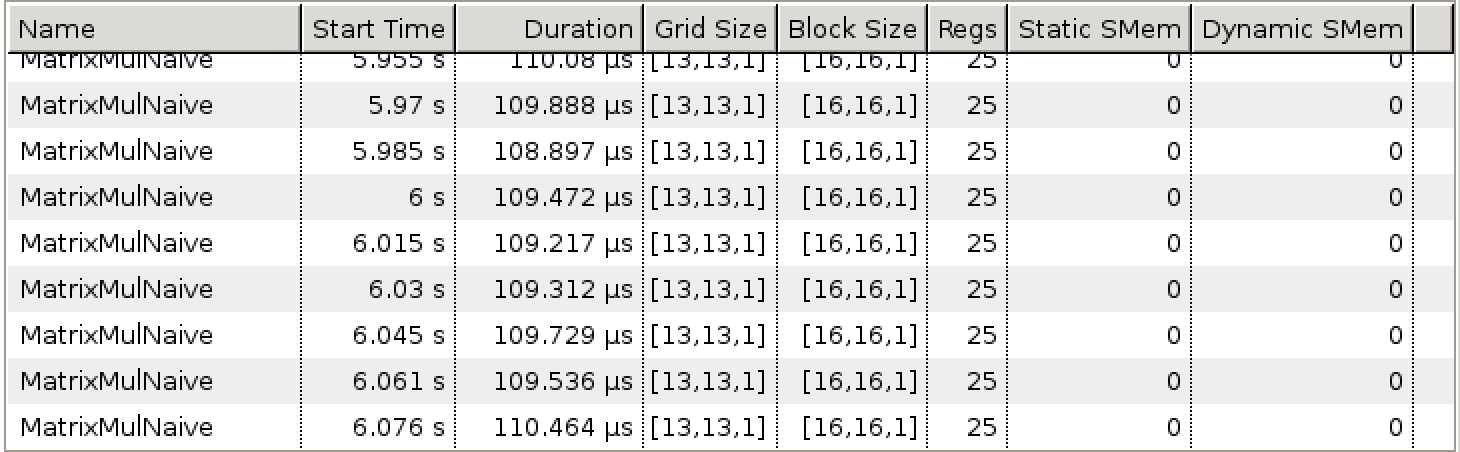
An interesting fact is that kernel execution time doesn’t grow as much as matrix size grows. While the including-memory-copy time grows almost proportionally to the size.

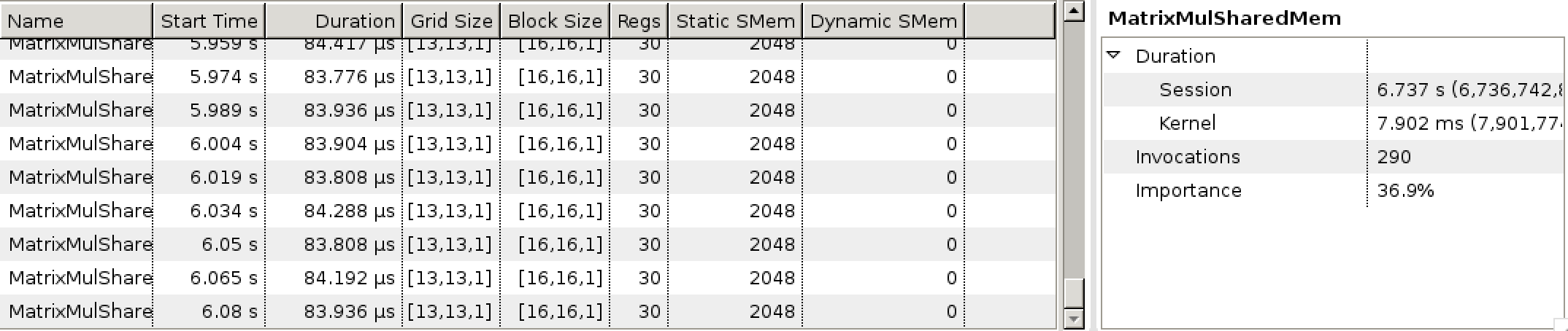
The NVIDIA visual profile looks like this:

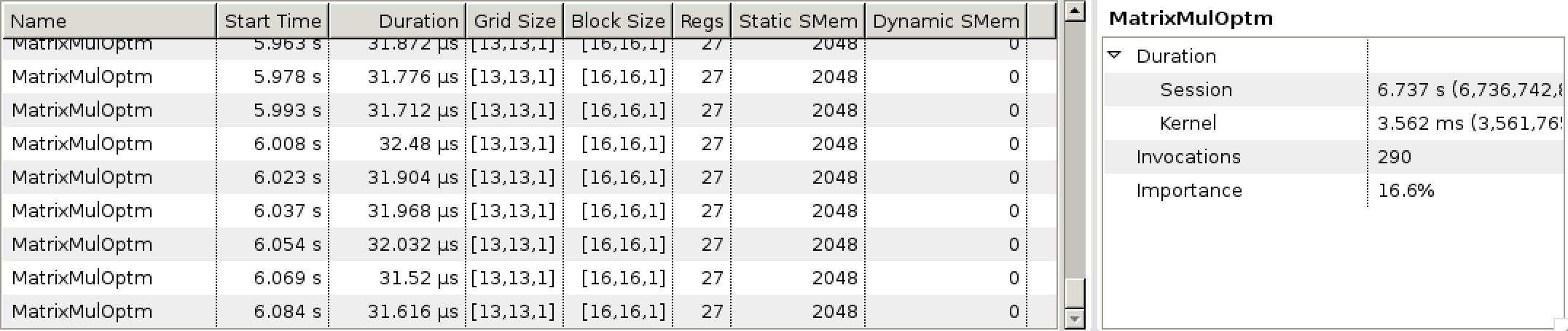


It shows I called each method for about 290 times. Just as showed in graph, naïve version cost most time. Shared memory does progress on optimization. Further optimized one has best performance.

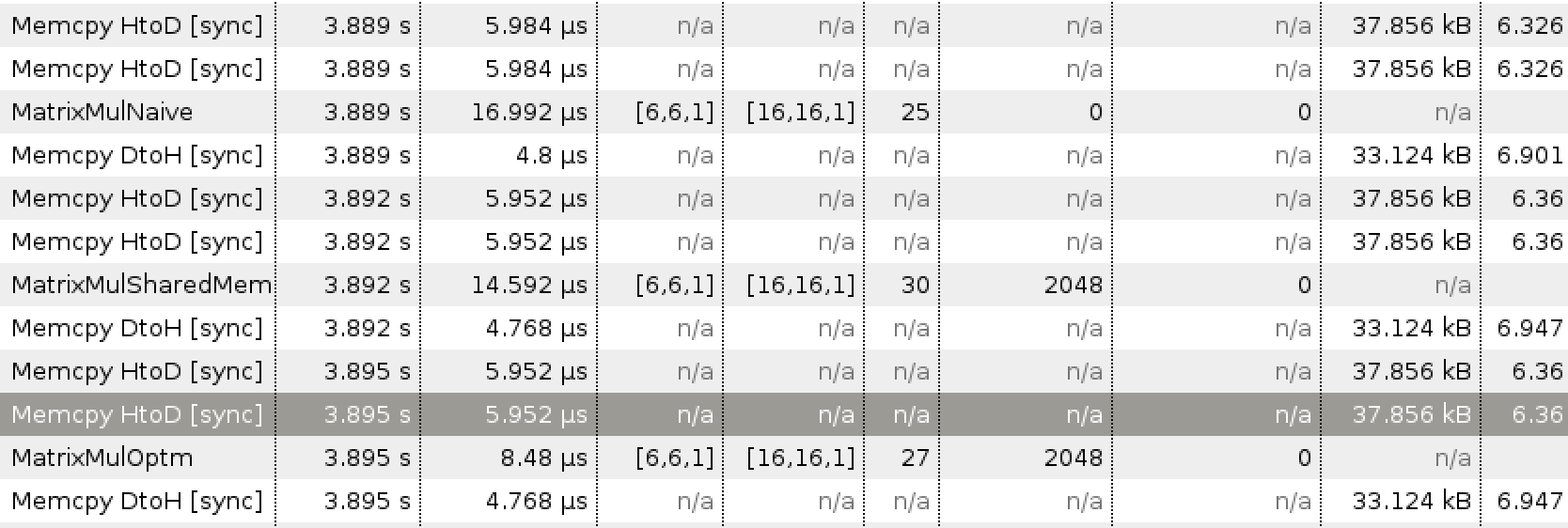
 



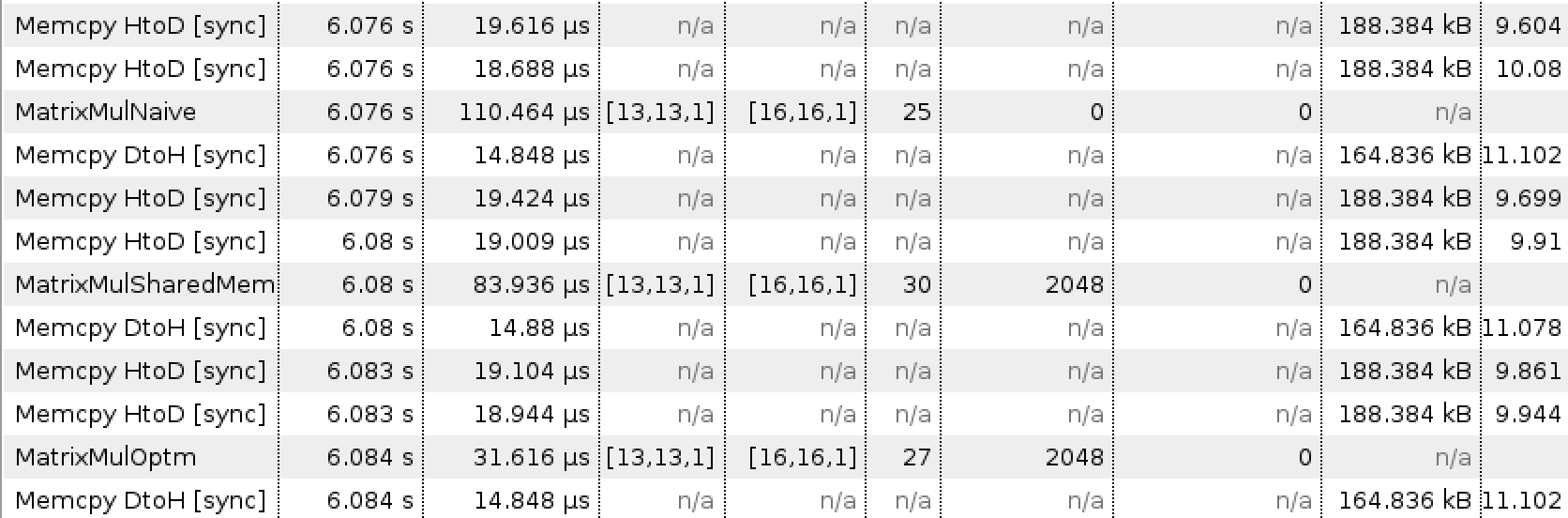




For kernel time, naïve one may cost around 109 μs, the shared memory one costs around 83.7 μs and the further optimized one costs around 32 μs.

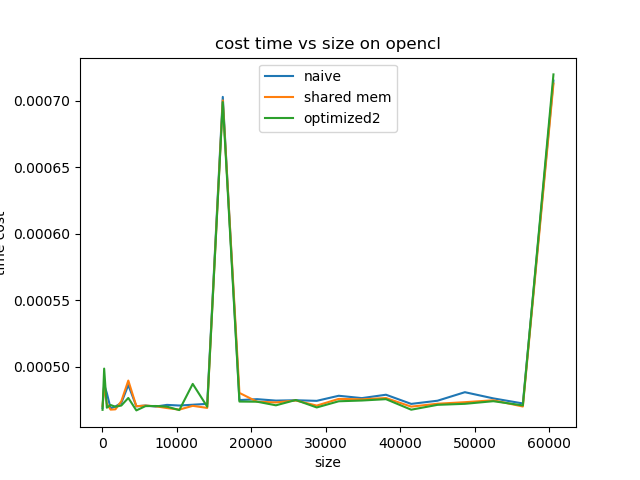


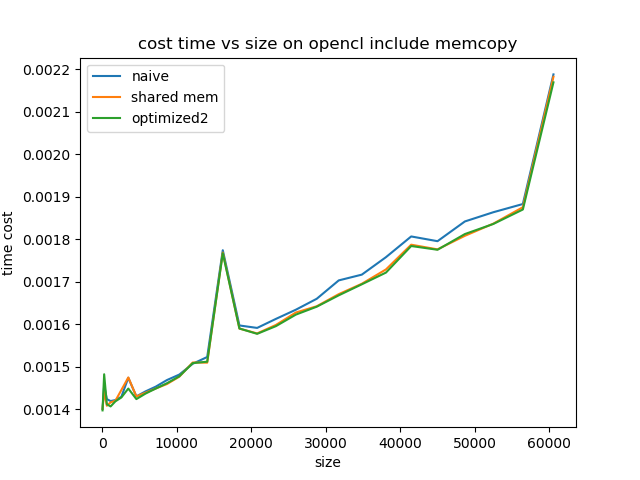
For example, when grid size is [6,6,1], memory copy time can cost 12 μs for host to device and 5 μs from device to host. Memory copy time is much more than execution time.



When grid size grows to [13,13,1] (matrix size is around 200\*230), memory copy time may be not so significant according to the execution time of naïve version.

For OpenCL:





Performances for three algorithms on OpenCL do not show much difference. To use shared memory (\_\_local in OpenCL), I used LocalMemory class in PyOpenCL. One confusing issue I got is naïve algorithm has much less time cost than optimized one. To solve this, I tried to also input local memory objects in to kernel function of naïve version (of course not used in kernel), then the result comes out as expectation. So, I keep this in my code. My guess is that the LocalMemory operation in PyOpenCL costs much time, so I need to balance this by adding same procedure to naïve version. This is not like CUDA, shared memory is directly declared in the kernel, OpenCL requires you to treat it as a input of kernel function.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | time include mem | time optm1 include mem | time optm2 include mem | time | time optm1 | time optm2 |
| 1 | 0.00140035 | 0.00140035 | 0.00139686 | 0.00046915 | 0.00047067 | 0.00046775 |
| 2 | 0.00145602 | 0.00144139 | 0.00148201 | 0.00048664 | 0.00048572 | 0.00049874 |
| 3 | 0.00142342 | 0.00140792 | 0.00141147 | 0.00048029 | 0.00047231 | 0.00046936 |
| 4 | 0.00141889 | 0.00141573 | 0.00140619 | 0.00046888 | 0.00046793 | 0.00047138 |
| 5 | 0.00142121 | 0.00141975 | 0.00141826 | 0.00047028 | 0.00046812 | 0.00047007 |
| 6 | 0.00142878 | 0.00144476 | 0.00142726 | 0.00047296 | 0.00047424 | 0.000471 |
| 7 | 0.00147364 | 0.00147444 | 0.00144833 | 0.00048637 | 0.00048974 | 0.00047669 |
| 8 | 0.00142986 | 0.0014298 | 0.00142348 | 0.0004704 | 0.00047046 | 0.00046733 |
| 9 | 0.0014416 | 0.00143844 | 0.00143641 | 0.000471 | 0.00047097 | 0.00047055 |
| 10 | 0.00145271 | 0.00144836 | 0.00144824 | 0.00047021 | 0.00047043 | 0.00047064 |
| 11 | 0.00146833 | 0.00145897 | 0.00146085 | 0.00047147 | 0.00046909 | 0.00047013 |
| 12 | 0.00148124 | 0.00147626 | 0.00147715 | 0.000471 | 0.00046793 | 0.00046757 |
| 13 | 0.00150672 | 0.00150973 | 0.00150821 | 0.00047159 | 0.00047076 | 0.00048724 |
| 14 | 0.0015226 | 0.00150929 | 0.00151157 | 0.00047225 | 0.00046921 | 0.00047013 |
| 15 | 0.00177419 | 0.00176865 | 0.00176704 | 0.00070289 | 0.00070027 | 0.00069898 |
| 16 | 0.0015969 | 0.00158954 | 0.00158975 | 0.00047508 | 0.00048047 | 0.00047398 |
| 17 | 0.00159112 | 0.00157815 | 0.0015772 | 0.00047576 | 0.00047383 | 0.00047386 |
| 18 | 0.0016124 | 0.00159812 | 0.00159532 | 0.00047469 | 0.00047332 | 0.00047115 |
| 19 | 0.00163392 | 0.00162733 | 0.00162238 | 0.00047496 | 0.00047478 | 0.00047514 |
| 20 | 0.00165975 | 0.00164214 | 0.00164109 | 0.00047451 | 0.00047094 | 0.00046955 |
| 21 | 0.00170302 | 0.00167063 | 0.00166786 | 0.00047833 | 0.00047597 | 0.00047418 |
| 22 | 0.00171661 | 0.00169513 | 0.00169386 | 0.00047656 | 0.00047565 | 0.00047475 |
| 23 | 0.00175777 | 0.00172889 | 0.00172145 | 0.0004791 | 0.00047663 | 0.00047576 |
| 24 | 0.00180656 | 0.00178713 | 0.001784 | 0.00047222 | 0.00047016 | 0.00046786 |
| 25 | 0.00179538 | 0.00177613 | 0.00177521 | 0.0004746 | 0.00047234 | 0.0004715 |
| 26 | 0.00184187 | 0.00180805 | 0.00181219 | 0.00048101 | 0.0004735 | 0.00047227 |
| 27 | 0.0018636 | 0.00183672 | 0.00183597 | 0.00047636 | 0.00047493 | 0.00047424 |
| 28 | 0.00188294 | 0.00187543 | 0.00186998 | 0.00047255 | 0.00047025 | 0.00047109 |
| 29 | 0.00218821 | 0.0021826 | 0.00216949 | 0.00071514 | 0.00071284 | 0.00071979 |

## 2. Theory Problems (20 pts, 4 points each)

**2.1** If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index: (A) i=threadIdx.x + threadIdx.y; (B) i=blockIdx.x + threadIdx.x; (C) i=blockIdx.x\*blockDim.x + threadIdx.x; (D) i=blockIdx.x \* threadIdx.x.

Answer:

(C) should be the correct one.

**2.2** We want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?(A) i=blockIdx.xblockDim.x + threadIdx.x +2; (B) i=blockIdx.xthreadIdx.x2; (C) i=(blockIdx.xblockDim.x + threadIdx.x)2; (D) i=blockIdx.xblockDim.x\*2 + threadIdx.x.

Answer:

(C) should be the correct one.

**2.3** Assume that a kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel? a. 1; ;b. 1,000; c. 512; d. 512,000.

Answer:

(b) should be the correct one.

**2.4** For our tiled matrix-matrix multiplication kernel, if we use a 32X32 tile, what is the reduction of memory bandwidth usage for input matrices A and B? a. 1/8 of the original usage; b. 1/16 of the original usage; c. 1/32 of the original usage; d. 1/64 of the original usage.

Answer:

(c) should be the correct one.

**2.5** For the tiled single-precision matrix multiplication kernel as shown in one of the lectures, assume that the tile size is 32X32 and the system has a DRAM burst size of 128 bytes. How many DRAM bursts will be delivered to the processor as a result of loading one A-matrix tile by a thread block?a. 16; b. 32; c. 64; d. 128.

Answer:

There are sizeof(float)\*32\*32=2048 bytes in a tile. Burst size is 128 bytes. There are 2048/128=16 bursts in total.

(a) should be the correct one.