Lab (2)

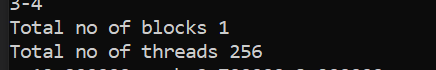
|  |  |  |
| --- | --- | --- |
| Name | Sec | BN |
| Basma Hatem Elhoseny | 1 | 16 |
| Sarah Mohamed Hossam Hassan | 1 | 29 |

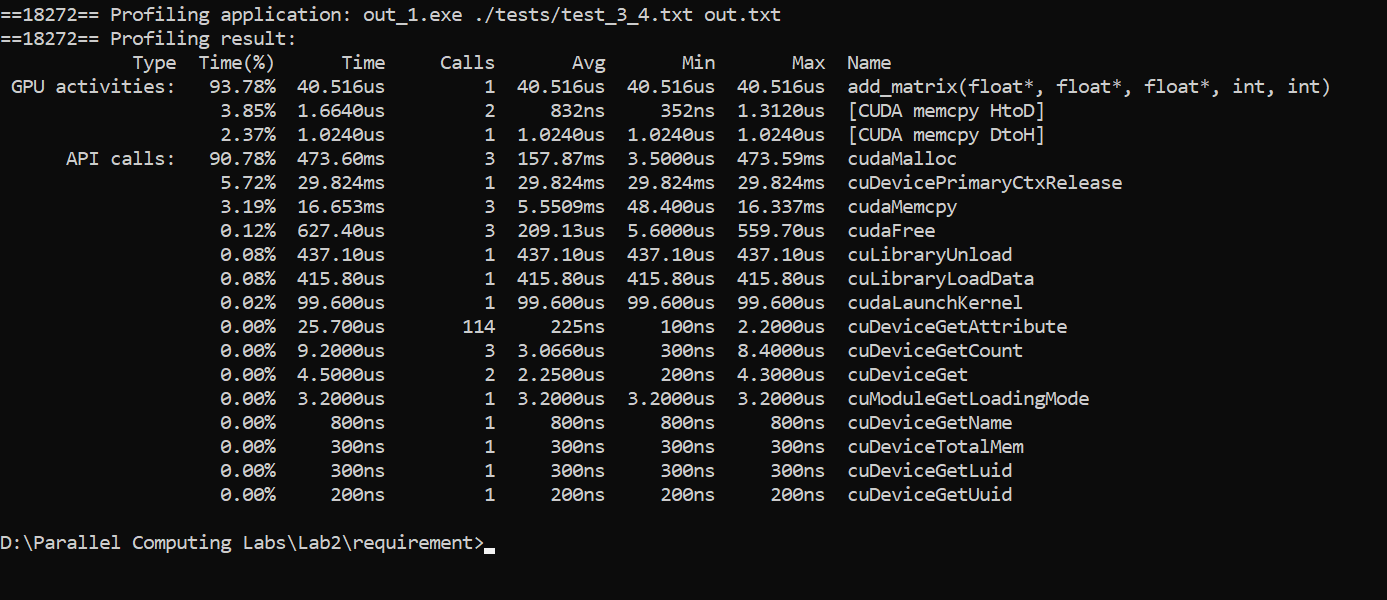
## Requirement(1) [Matrix Addition]:

### Kernel (1) Each Thread Produces one output matrix element:

#### Case(1) 3\*4 Matrix

With configuration that kernel size 16\*16 and one thread per element so:

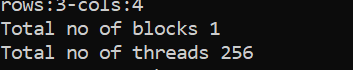


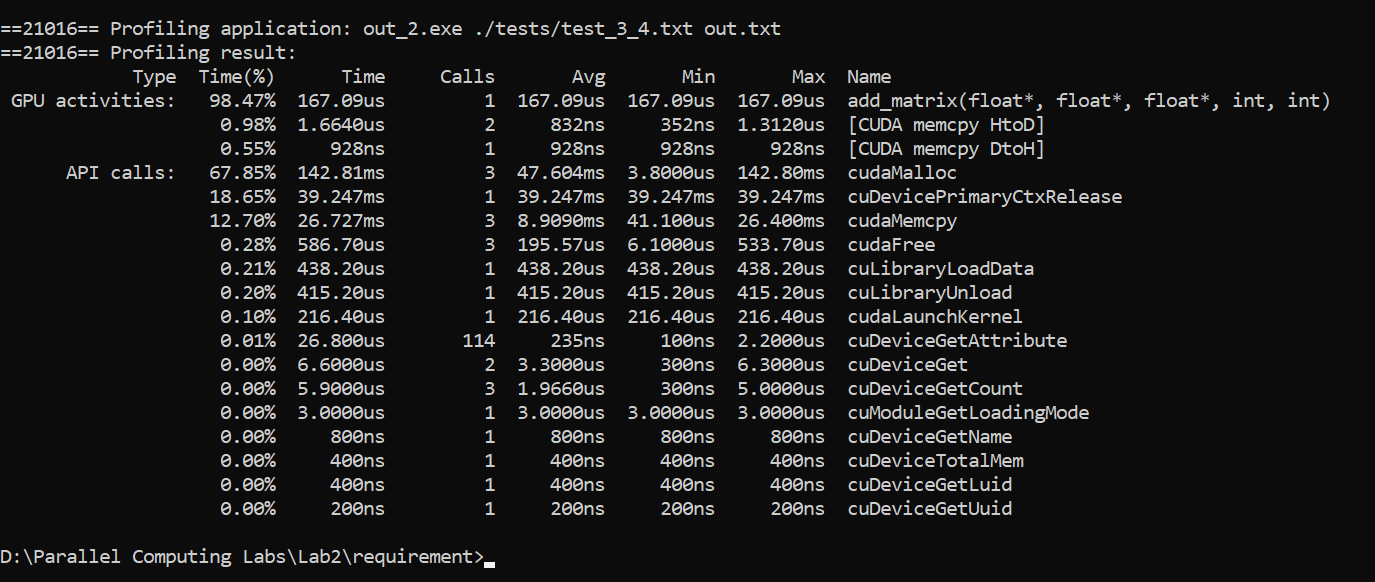


### Kernel(2) Each Thread Produces one output matrix row:

#### Case(1) 3\*4 Matrix

With configuration that kernel size 256 and one thread per rows so:



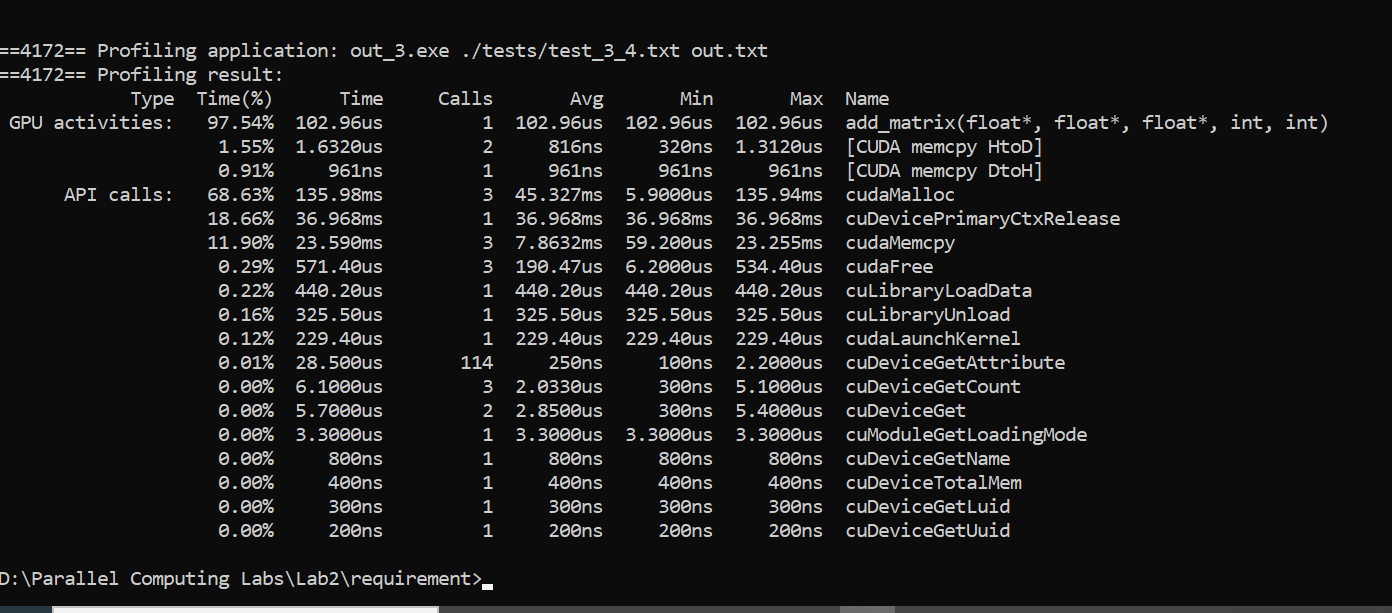


### Kernel(3) Each Thread Produces one output matrix column:

#### Case(1) 3\*4 Matrix

With configuration that kernel size 256 and one thread per rows so:





Comments:

* Case 3\*4 it is clear that the kernel 1 is the fastest regarding computing the addition function and nearly the 3 kernels have near copying time form host to device and vice versa 😊
* We think whatever the matrix size, kernel (1) will be the fastest Are we right ?! 🤔 Let’s see <3

## Bench Marking:

Notice that these results are based on running matrix produced as random numbers generated by a script :D. [Runed on Colab]

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix Shape | Kernel(1) [element] | Kernel(2) [row] | Kernel(3) [col] |
| 3x4 | 80.672us | 279.52us | 213.41us |
| 2x2 | 80.159us | 143.94us | 144.10us |
| 2x4 | 80.480us | 277.12us | 145.79us |
| Total of 10,000 elements | | | |
| 100x100 | 33.184ms | 14.142ms | 13.420ms |
| 50x200 | 31.617ms | 17.931ms | 10.002ms |
| 200x50 | 28.655ms | 10.427ms | 17.857ms |
| 10000x1 | 32.122ms | 30.463ms | 448.05ms |
| 1x10000 | 24.259ms | 286.08ms | 33.404ms |
| Total of 50,000 elements | | | |
| 50000x1 | 238.71ms | 1.16151s | 1.56368s |
| 1x50000 | 318.95ms | 1.43044s | 1.03092s |
| 1000x50 | 1.54740s | 44.100ms | 89.304ms |
| 50x1000 | 1.10863s | 87.770ms | 43.907ms |
| Total of 1,000,000 elements | | | |
| 1000x1000 | 133.538s | 882.08ms | 876.51ms |
| 1000000x1 | 4.85198s | 114.753s | 28.5801s |
| 1x1000000 | 6.47006s | 28.4147s | 141.907s |

**Notes:**

For the First Three example since matrix dim is very small we don’t sense a lot of improvement between the 3 kernels so we will apply like stress test 🎏🎏

Total of 10,000-element Matrix

For 100x100:

* The three kernels are nearly the same except that kernel(1) is bit higher due to more threads are required 100x100 while in both kernel(2)&(3) only 100 thread is required.

For 50x200:

* Kernel(3) is the best bec only 200 thread is required each thread make only 50 operation :D

For 200x50:

* Kernel(2) is the best bec only 200 thread is required each thread make only 50 operation :D

For 10000x1:

* Worst Is Kernel(3) 448.05 ms because simply this is computed by single thread (No parallelization) :D

For 1x10000:

* Worst Is Kernel(2) 286.08 ms because simply this is computed by single thread (No parallelization) :D

Total of 1,000,000-element Matrix

For 1000x1000:

* It is clear that the worst Time is for Kernel(1) where each thread is computing 1 element but since here the large no of elements we need 1000x1000 thread in total which may not be available to be used together so some is computed with the available threads and finishes then others (Scarcity of Thread compared to the huge no of threads required)
* While we get better time for kernel(2) & kernel(3) due to less no of threads required in both cases [1000 only :D] (This proves that our claim above is Wrong: kernel(1) is always the best ❌❌)

For 1x1000000 & 1000000x1:

* The Best will be the one will the orientation corresponding to the no of threads and the other will be the worst because on thread is responsible for the computation (No parallelism)

**Conclusion:**

We have corrected our faulty claim that kernel(1) will be always the same :D. It depends according to the total no of elements.

But it is clear that of course if the no of rows are more then kernel(2) is the best and kernel(3) will be the worst. And the same for the cols case .

**That’s All 😁**