Parallel Programming Homework4
All-Pairs Shortest Path in CUDA

CS 101062337 Hung Jin, Lin

Abstraction

This project aims to improve the performance of Floyd-Warshall algorithm while it's an $O(n^3)$ method. However, when it comes to paralleling, this kind of task is very suitable for GPU computing, large amount of well-defined data and simple computation logics.

Design

- What's blocked all-pairs shortest path method?

Given an $N \times N$ matrix $W = [w(i,j)]$ where $w(i,j) \geq 0$ represents the distance (weight of the edge) from a vertex i to a vertex j in a ***simple directed graph*** with N vertices. We define an $N \times N$ matrix $D = [d(i,j)]$ where $d(i,j)$ denotes the shortest-path distance from a vertex i to a vertex j. Let $D^{(k)} = \left[d^{(k)}(i,j)\right]$ be the result which all the intermediate vertices are in the set $\{1,2,\cdots,k\}$.

We define $d^{(k)}(i,j)$ as follows:

$$d^{(k)}(i,j) = \begin{cases} w(i,j) & if\ k = 0; \\ \min\left(d^{(k-1)}(i,j), d^{(k-1)}(i,k) + d^{(k-1)}(k,j)\right) & if\ k \geq 1. \end{cases}$$

The matrix $D^{(N)} = d^{(N)}(i,j)$ gives the answer to the APSP problem.

FIGURE 1. THE MATHEMATICAL REPRESENTATION OF BLOCKED ALL-PAIRS ALGORITHM[1]

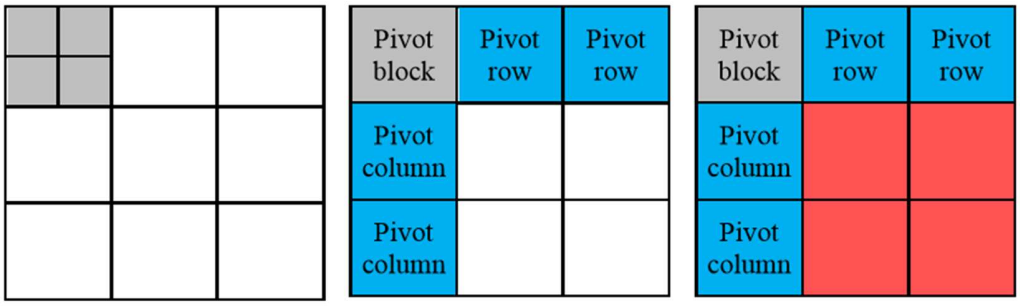


FIGURE 2 THE 3 PHASES OF BLOCKED FW ALGORITHM IN THE 1ST ITERATION[2]

[1] From homework4 spec document

[2] Same as previous

- Steps

  - The data is first transformed into adjacency matrix to represent the length of path between nodes.
  - In the blocked all-pairs shortest path algorithm, there are three computing phases and later one is dependent on previous calculations, so each phase has to execute in order.
  - Divide original raw matrix data into blocks by ***block factor*** in width and height.
  - The program will run ***(N + (block factor - 1) / block factor)*** rounds, and each run contain three phases of computations.
  - Each phase is controlled by one CUDA kernel function, inside kernel function, we can make a bundle of kernel threads do the same computing tasks in parallel.
  - Inside every phase, it may have different size of grid and block number in kernel function due to different size of paralleling data width.

Configuration

- Single GPU

As describing above, the program will run ***(N + (block factor - 1) / block factor)*** rounds, and in each phase has its own configuration on kernel resources usage. In my implementation, in order to make the best usage of GPU device threads, I make the **block size** = $block\ factor \times\ block\ factor$.

- **Phase I**

  Only focus on itself, namely, pivot block, which is a single block of ($block\ factor \times\ block\ factor$) elements inside.

- **Phase II**

  Focus on *pivot-row* and *pivot-column* blocks except pivot bock, it contains two lines of blocks which all locate at the same row or column of pivot block. We know the width and height of blocks is equal to the program needed running times ***round*** and now we assign it to a symbol **r**.

- **Phase III**

  Compute with the remaining part which in fact are $(\mathbf{r}-\mathbf{1}) \times (\mathbf{r}-1)$ blocks.

So, here is how's **the configurations in each phase of CUDA computing**. I use two dimension of grid structure to represent the block distribution of above three situations and mapped device's threads to each element inside block.

	Grid Size	Block Size
Phase I	(1, 1)	block factor × block factor
Phase II	(r, 2)	block factor × block factor
Phase III	(r, r)	block factor × block factor

Multi GPU

While in multi-GPUs environment, the data will be equally split into N parts by rows' ID. In the most general case, it may be complicated to exchange between multi-devices. **However, in our current workspace environment, we can only access two devices at the same time, so I developed a specific method to control two pieces of device.**

In my opinion, the algorithm of blocked all pairs shortest path will take three phases in computation, and the phase 1 and phase 2 effect little to the overall consuming time, so I decide to **focus on optimization on phase three** with two GPUs on single node.

- First, create two independent host thread or process to handle one GPU device, and then mark one of them as **master** which will control some tasks for communication.
- Then, in each device, allocate an area of memory for full size of adjacency matrix, and the master one will need to provide another full size for swap space for later usage.
- Master will do the phase one and two, at the same time, another process must wait for these two phases due to the dependencies.
- Master at host-side will copy the result from previous procedure to another device and let it get the data for corresponding phase three.

> ***Difference: OpenMP v.s. MPI***
>
> There are some different between OpenMP and MPI version. OpenMP use peer-to-peer communication that means making GPU device may copy its data to another device **through their high-speed DMA transfers between the memories of two GPUs on the same system/PCIe bus**[3].
>
> While with MPI, it also has a well-defined strategy to transfer between devices called *MPI aware* in CUDA, however, in the implementation, I cannot make it run with this feature. So, **I finally use an explicit way of MPI aware**, in which it just hides the memory copy between host and device on one machine, it actually need some code manipulating memory transfer between CPU and GPU.

- In **phase three, both processes or threads launch the GPU kernel and compute on their own part**. And also need to wait and make sure two asynchronous kernel functions are complete and data is synchronous.

[3] Nvidia Developer document about GPU Direct Access (https://developer.nvidia.com/gpudirect)

- **Copy back the result to the master's swap space** that we mentioned before from another device. And then let master process or thread to **merge result to get a complete one**. Then, if procedure not finish yet, go back to step one.

System Spec

There are three machines, gpucluster0, gpucuster1 and gpucluster2. They are connected with **Ethernet**.

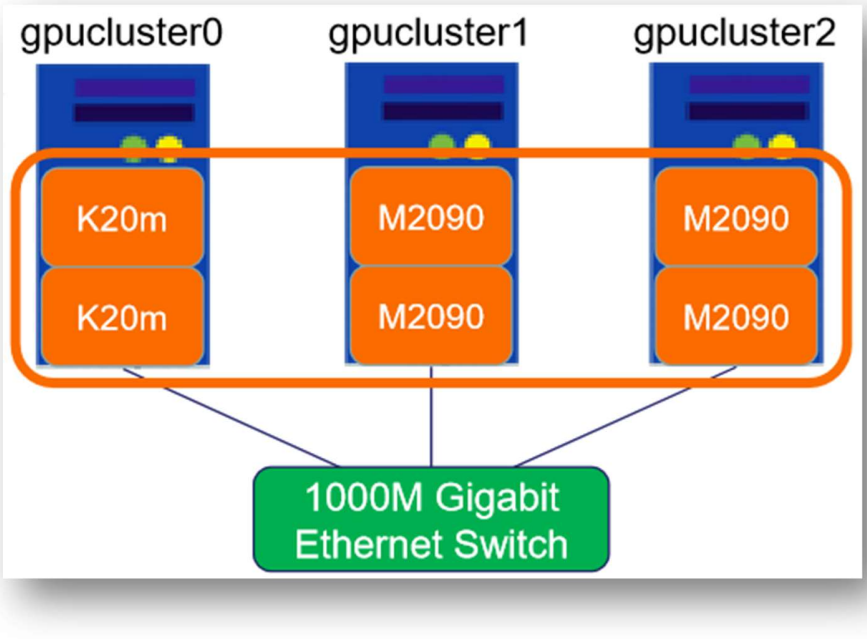
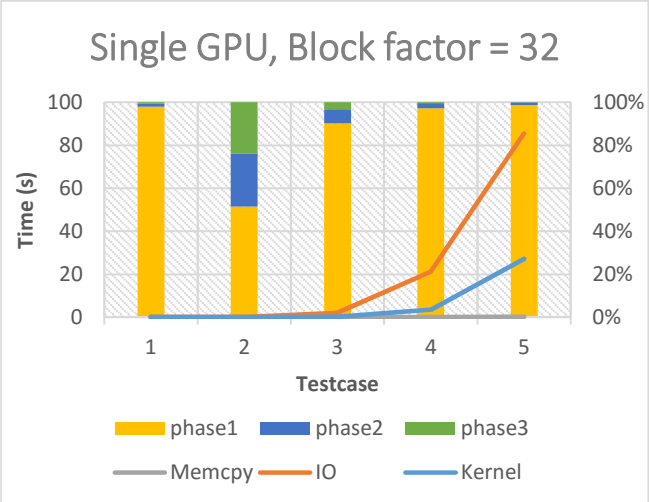
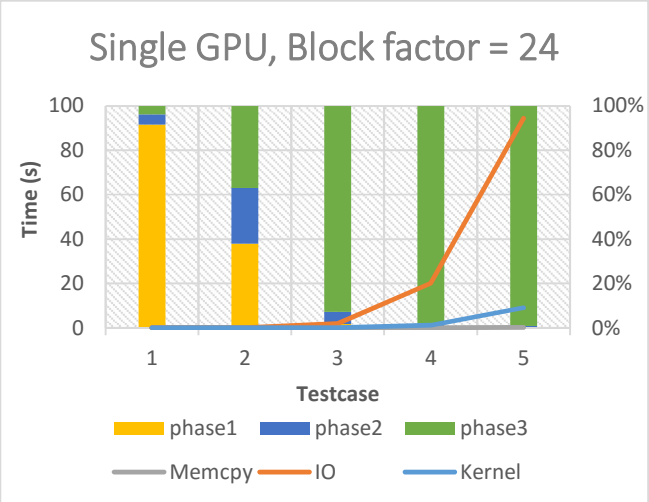
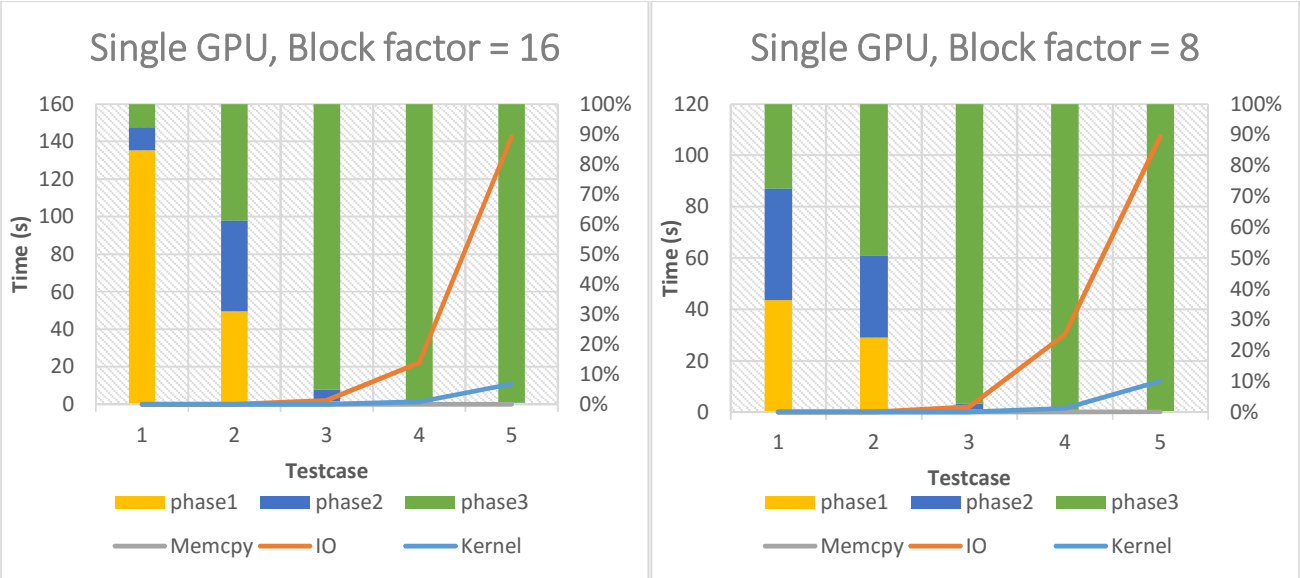


FIGURE 3 WORKSTATION MACHINES

Profiling





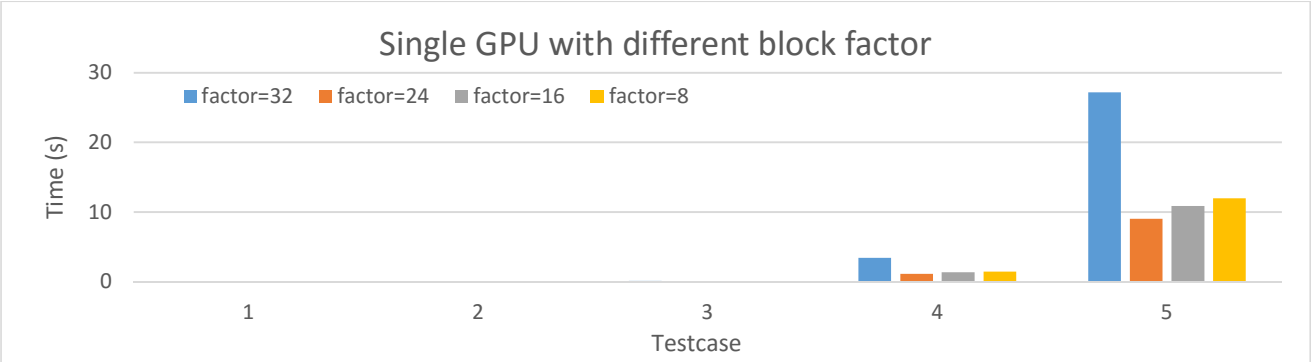


These charts show the performance of program on single GPU. The statistics all come from ***nvprof*** profiling report as below profile shows. In the above graph, bar charts show the percentage of time in three phases; line graphs are time costing on I/O, memory copying, and time on device kernel of computing.

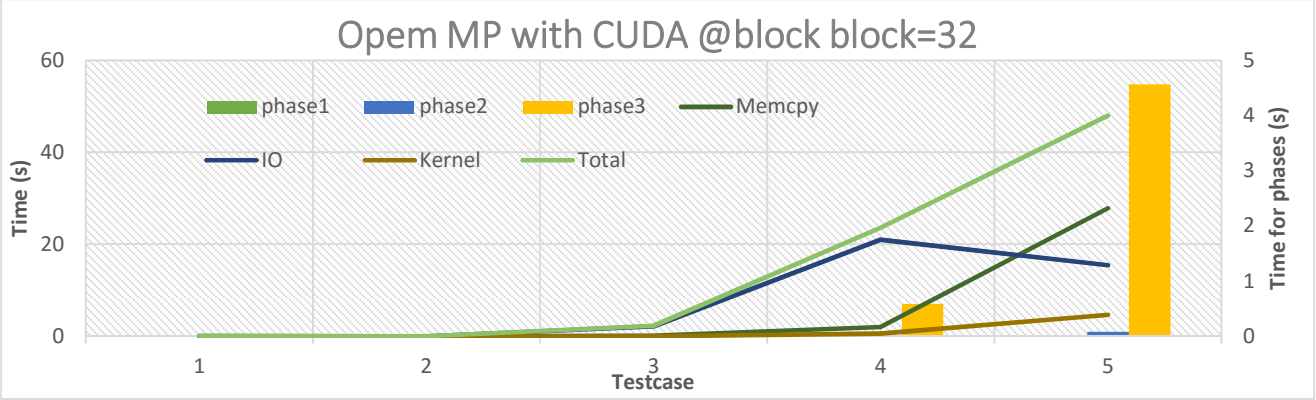
```
gpucluster0:  ~/hw  master {2} U:2 ?:16 ✗
→ nvprof -i prof5_single
======== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.17%  2.54761s       188  13.551ms  13.162ms  13.559ms  kernel_phase3(int, int, int*, int)
  8.68%  250.73ms         1  250.73ms  250.73ms  250.73ms  [CUDA memcpy DtoH]
  1.86%  53.666ms         1  53.666ms  53.666ms  53.666ms  [CUDA memcpy HtoD]
  1.18%  34.206ms       188  181.95us  149.25us  188.07us  kernel_phase2(int, int, int*, int)
  0.11%  3.2005ms       188  17.024us  16.641us  17.728us  kernel_phase1(int, int, int*, int)

======== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 81.28%  2.57783s         1  2.57783s  2.57783s  2.57783s  cudaEventSynchronize
  8.77%  278.24ms         2  139.12ms  2.3280us  278.24ms  cudaEventCreate
  7.95%  252.03ms         1  252.03ms  252.03ms  252.03ms  cudaMemcpy
  1.70%  53.996ms         1  53.996ms  53.996ms  53.996ms  cudaMemcpyAsync
  0.20%  6.1924ms       564  10.979us  9.9140us  58.400us  cudaLaunch
  0.03%  913.28us       166  5.5010us     344ns  205.65us  cuDeviceGetAttribute
  0.03%  885.44us      2256     392ns     326ns  9.4580us  cudaSetupArgument
  0.01%  393.34us         1  393.34us  393.34us  393.34us  cudaFree
  0.01%  312.50us         1  312.50us  312.50us  312.50us  cudaMalloc
  0.01%  301.33us       564     534ns     450ns  8.8040us  cudaConfigureCall
  0.00%  104.18us         2  52.088us  50.882us  53.294us  cuDeviceTotalMem
  0.00%  97.354us         2  48.677us  44.554us  52.800us  cuDeviceGetName
  0.00%  24.436us         2  12.218us  4.6520us  19.784us  cudaEventRecord
  0.00%  23.398us         1  23.398us  23.398us  23.398us  cudaEventElapsedTime
  0.00%  17.634us         1  17.634us  17.634us  17.634us  cudaSetDevice
  0.00%  8.7340us         2  4.3670us  1.7200us  7.0140us  cudaEventDestroy
  0.00%  3.9160us         4     979ns     386ns  2.4540us  cuDeviceGet
  0.00%  2.5480us         2  1.2740us     610ns  1.9380us  cuDeviceGetCount
```

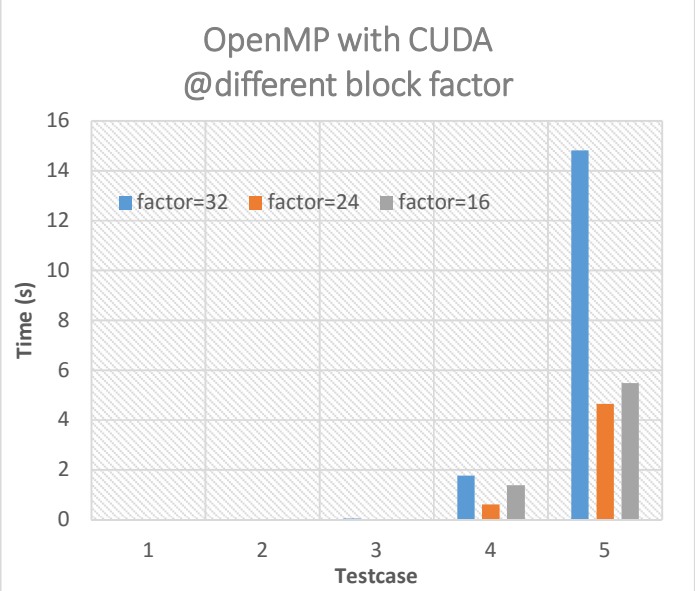
Test case 5, block factor=32 @K20m

FIGURE 4 PROFILE ON SINGLE GPU



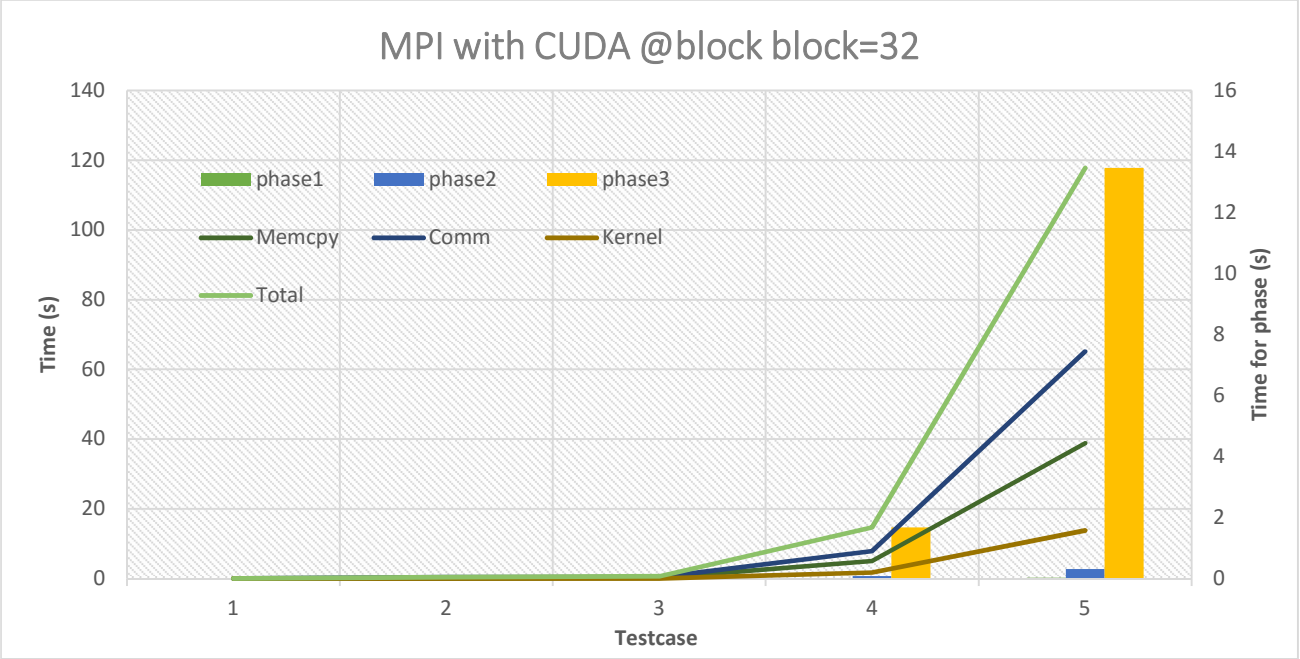
An overall view of four factors effecting the computing performance. We can observe that as block factor is 24, the program gets the better results. Maybe it's due to over usage of threads on single device; the *max threads per block is 1024 on K20m and M2090*, and in my implementation, I'll use at most *(block factor × block factor)* threads simultaneously, as the result, when block size is 32, it will use up to 1024 threads! It's mentioned in the spec that *higher occupancy does not necessarily mean higher performance*. And in this experiment, it may somehow show the similar phenomena.





We can notice that the **OpenMP implementation version runs faster than single GPU version in phase three**, the **phase three is parallelized** and completed by two devices, so the running time is cut half down! But, it cost more time in copying data though have used high speed way of peer to peer transfer. Similar situation appears again, as block size is 24, this program performs better.

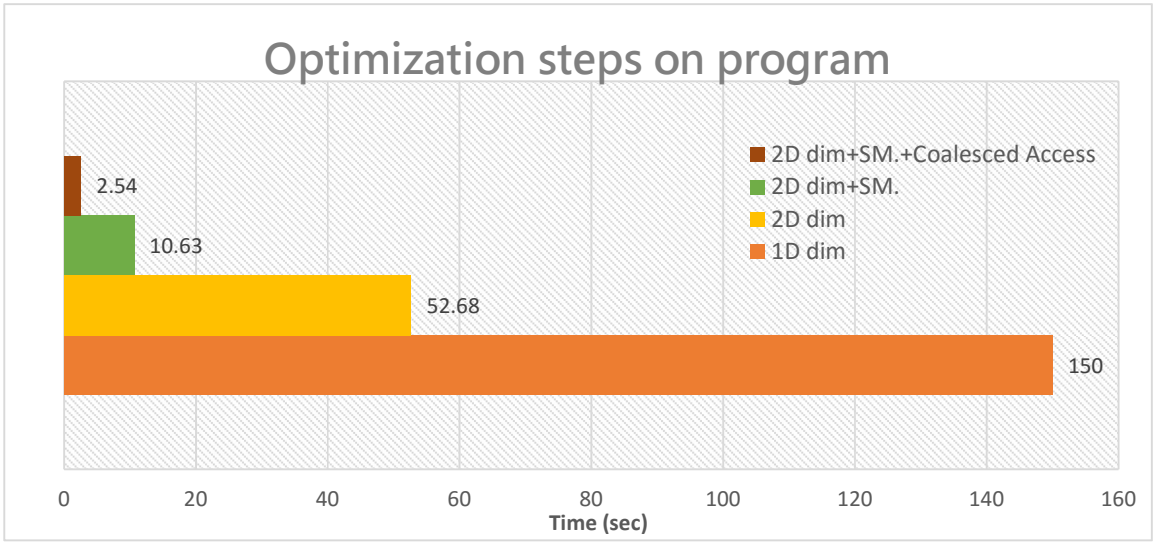
And in **MPI implementation version**, phase three is speedup, too. However, at the meanwhile, communications cost a lot and slow down the total performance when the **kernel functions still perform better than single**!



Optimizations

Use test case 5 in spec as testing sample. Run with block factor=32 on *GPU cluster#0 on Tesla K20m*.

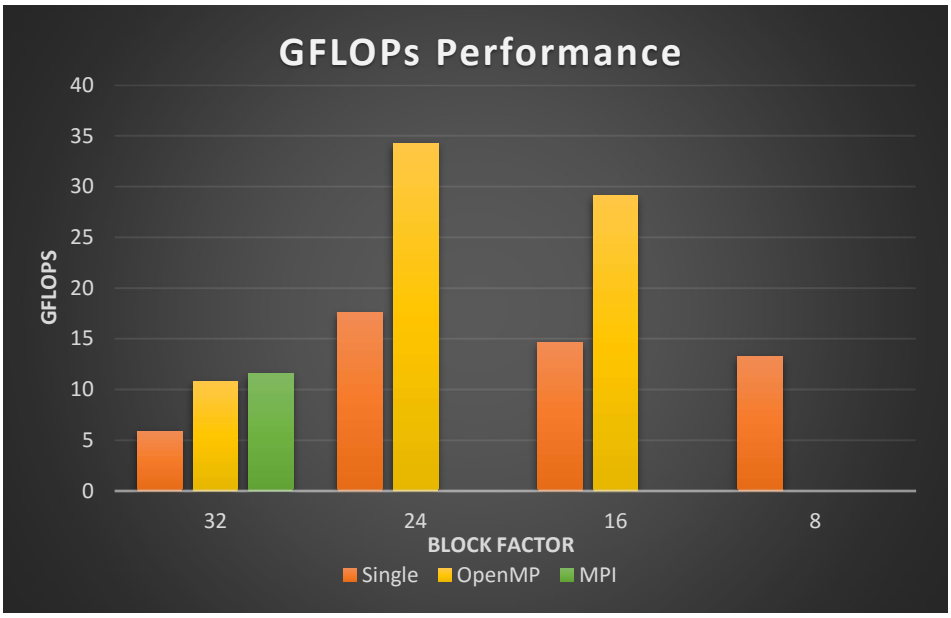
- Use 2 dimension of blocks (square of block factor) which can make use of max threads per block.
- **Shared memory**, it can reduce cost time on testcase5 from 50 sec. to 10 seconds with *K20m*.
- Consider in **memory alignment and coalesced access**, tuning the way of indexing global data array in kernel functions.

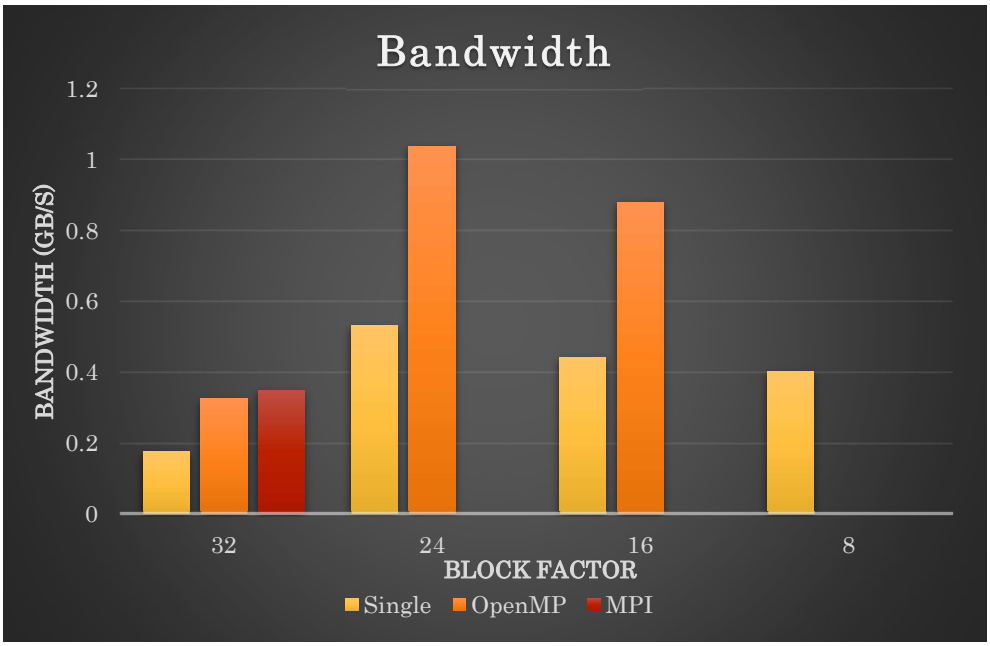


Blocking Factor

- Below are benchmarks of GFLOPs and bandwidth, they're testing on ***M2090***.

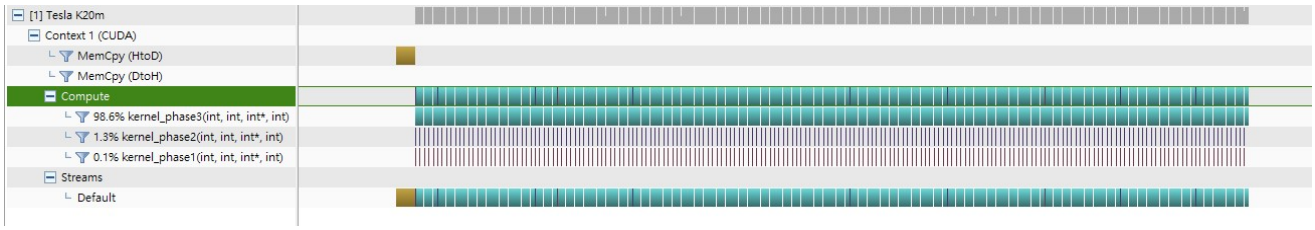
GFLOPs means how many add and multiplication operations is done during execution. It's $O(n^3 \times \text{instr})$.





Nvvp Visual Profiling

Here are some visual profiling reports about program performance on **single GPU program**. It's clearly show that **phase3 take most heavy tasks**!



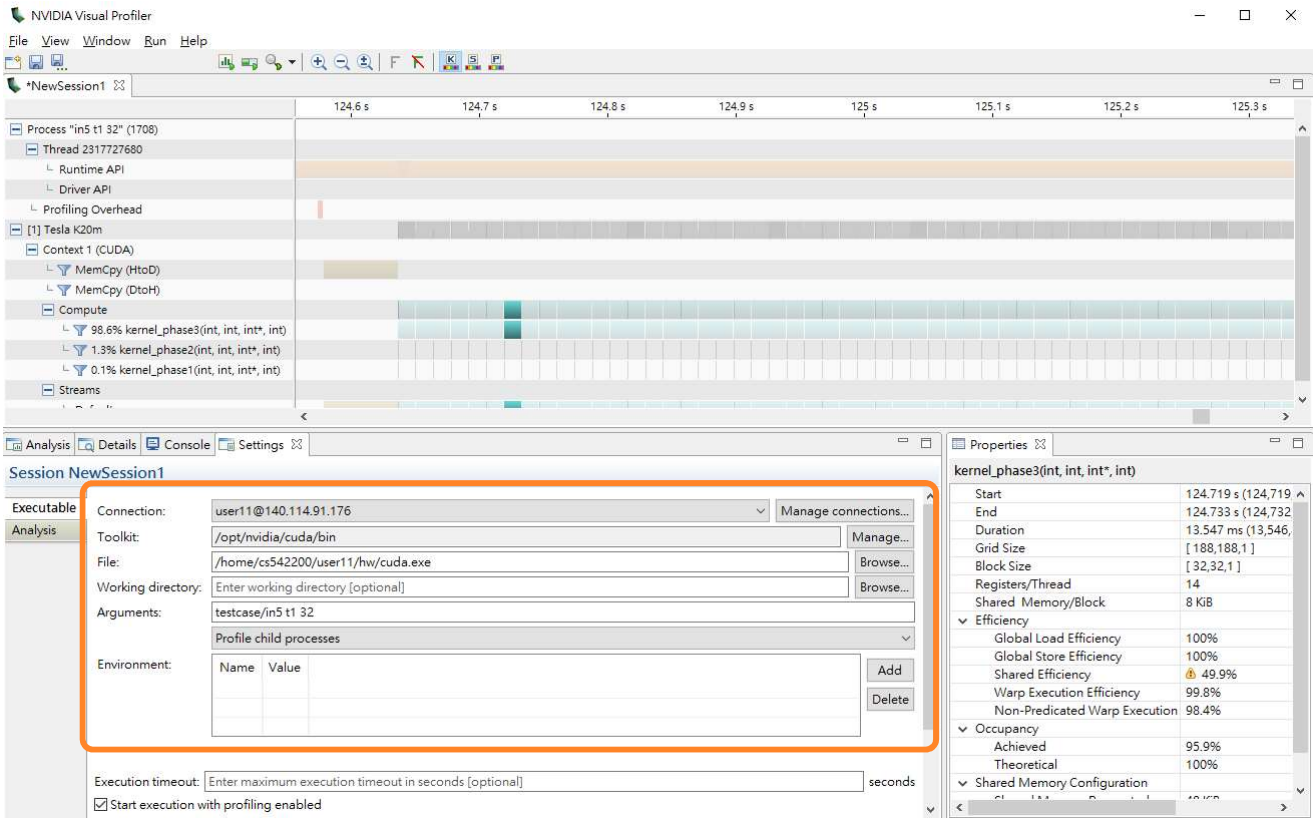
And also from analysis of performance checking, *nvprof's gpu-trace* report that **CUDA memcpy throughputs can reach speed 3.1443GB/s**, and also calculate the shared memory usage: in phase 1, it takes 4.096 KB while in phase 2 and 3 use 8192 KB.

```
gpucluster0:  ~/hw  !master {2} U:2 ?:15 X!
→ time nvprof --print-gpu-trace ./cuda.exe testcase/in3 t1 32
Start computing
==27353== NVPROF is profiling process 27353, command: ./cuda.exe testcase/in3 t1 32
k_time: 13.881440 ms
Compute finished.
io_time: 0.165626 sec
==27353== Profiling application: ./cuda.exe testcase/in3 t1 32
==27353== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
1.35102s  1.2721ms                    -               -         -         -         -  4.0000MB  3.1443GB/s    Tesla K20m (1)         1         7  [CUDA memcpy HtoD]
1.35230s  17.696us              (1 1 1)       (32 32 1)        10        0B  4.0960KB         -           -    Tesla K20m (1)         1         7  kernel_phase1(int, int, int*, int) [188]
1.35232s  34.242us              (32 2 1)       (32 32 1)        13        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase2(int, int, int*, int) [194]
1.35236s  374.09us            (32 32 1)       (32 32 1)        14        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase3(int, int, int*, int) [200]
1.35273s  17.056us              (1 1 1)       (32 32 1)        10        0B  4.0960KB         -           -    Tesla K20m (1)         1         7  kernel_phase1(int, int, int*, int) [206]
1.35275s  33.249us              (32 2 1)       (32 32 1)        13        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase2(int, int, int*, int) [212]
1.35279s  374.28us            (32 32 1)       (32 32 1)        14        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase3(int, int, int*, int) [218]
1.35317s  17.217us              (1 1 1)       (32 32 1)        10        0B  4.0960KB         -           -    Tesla K20m (1)         1         7  kernel_phase1(int, int, int*, int) [224]
1.35319s  34.049us              (32 2 1)       (32 32 1)        13        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase2(int, int, int*, int) [230]
1.35322s  374.09us            (32 32 1)       (32 32 1)        14        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase3(int, int, int*, int) [236]
1.35360s  16.897us              (1 1 1)       (32 32 1)        10        0B  4.0960KB         -           -    Tesla K20m (1)         1         7  kernel_phase1(int, int, int*, int) [242]
1.35362s  33.633us              (32 2 1)       (32 32 1)        13        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase2(int, int, int*, int) [248]
1.35366s  374.16us            (32 32 1)       (32 32 1)        14        0B  8.1920KB         -           -    Tesla K20m (1)         1         7  kernel_phase3(int, int, int*, int) [254]
```

Experience & Conclusion

It's my first time to program in CUDA that I'm long for exploring how to accelerate original task with GPU! It's amazing to use GPU to parallelize original for-loop basic logical problem, and it has a lot of difference comparing to OpneMP's multithreading concept, a brand new thought on threads and block of threads.

And I learn another lesson is that had better do the work as early as possible. During this time of experiment, I had to **compete for GPU resources**, then I can test and debug. It's a terrible time, and the **responses from GPU clusters are not stable**, they have large errors and difference between several same input arguments. Though, anyway, CUDA is one I will continue to try and go on studying, there are still too much skills and optimizations need to be done in my program. Even more, I may use GPU computing to resolve other problems in my professional knowledge domain. Programming in CUDA is really interesting, there is just one NVidia GPU in my laptop which I will first try on!



I also learned how to **remote profiling** my program on GPU cluster with my own compuer as a client. It's more clear know what to do next omparing to text mode of nvprof, GUI is more firendly to a novice. In addition, there are more **functional enhancement hints and warnings from its ananysis**.