CS239 Programming Exercise 1

Junelle Rey C. Bacong
Department of Computer Science
University of the Philippines Diliman
bacong.junelle@gmail.com

29 September 2019

1 The Exercise

Consider three square matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C} \in \mathbb{R}^{N \times N}$. We want to analyze the parallel execution of a simple matrix addition in three different ways such that $c_{ij} = a_{ij} + b_{ij}$:

- `kernel_1t1e` where each thread producing one output matrix element
- `kernel_1t1r` where each thread producing one output matrix row
- `kernel_1t1e` where each thread producing one output matrix column

2 Preliminaries

I used my personal computer with `GeForce GTX 1080 Ti` graphics card and 3584 total cores. Its maximum block size (in 2D) is $(1024, 1024)$ while its maximum grid size is $(2.1 \times 10^9, 1024)$. The total global memory of my device is at most 11GB. In my experiments, I will only consider square matrices with dimension $N \leq 10,000$ for convenience, although the square matrices could be bigger than these since the maximum number of threads along the x direction could reach up to 1M. In this exercise, we want to observe the performance of each kernel in executing matrix addition according to their execution time, CGMA and memory bandwidth.

3 Results and Discussion

I checked the execution time of each kernel for varying dimension of $N \times N$ square matrices using an optimal block and grid configuration. In this setup, the `blockDim` was fixed and consisted of 32×32 number of threads – the warp size and the maximum number of threads in a block. I used this to recreate the matrices and to maximize the parallelism among the available threads, especially for `kernel_1t1e`. The `gridDim`, on the other hand, was set to match the dimension of the matrices i.e. `ceil(N/32)` $\times$ `ceil(N/32)`.

Figure 1 shows the result of the experiment. Clearly, there is a big difference between `kernel_1t1e` and the two other kernels, with more than 1000x and 2000x speed up for `kernel_1t1r` and `kernel_1t1c`, respectively. This performance of `kernel_1t1e` is trivial since each thread is working in parallel to compute for the value of each element c_{ij}. What is more interesting is the performance of the kernels `kernel_1t1r` and `kernel_1t1c`.

Between the two other kernels, `kernel_1t1c` had a higher execution time. The difference can be attributed to the way the global memory is accessed by the compiler. For C/C++ compilers in particular, multidimensional arrays are stored in a contiguous physical memory in row-major order. This implies that accessing the neighboring memory addresses is much faster than, say, accessing it randomly. In `kernel_1t1r`, we restricted each thread to compute for the value of c_{ij} in its corresponding row index. This computation, however impractical and inefficient, is still faster than the `kernel_1t1c` because it accesses the global memory array contiguously, unlike the latter which reads and writes every N increments in the memory. We see the effect of this in Figure 2 wherein the difference in the execution time of these two kernels increases almost exponentially with N. For row-major order compilation, larger values of N mean more skips and large increments for the next value of c_{ij}. As such, this way of accessing the memory takes a toll to the overall speed of the kernel.

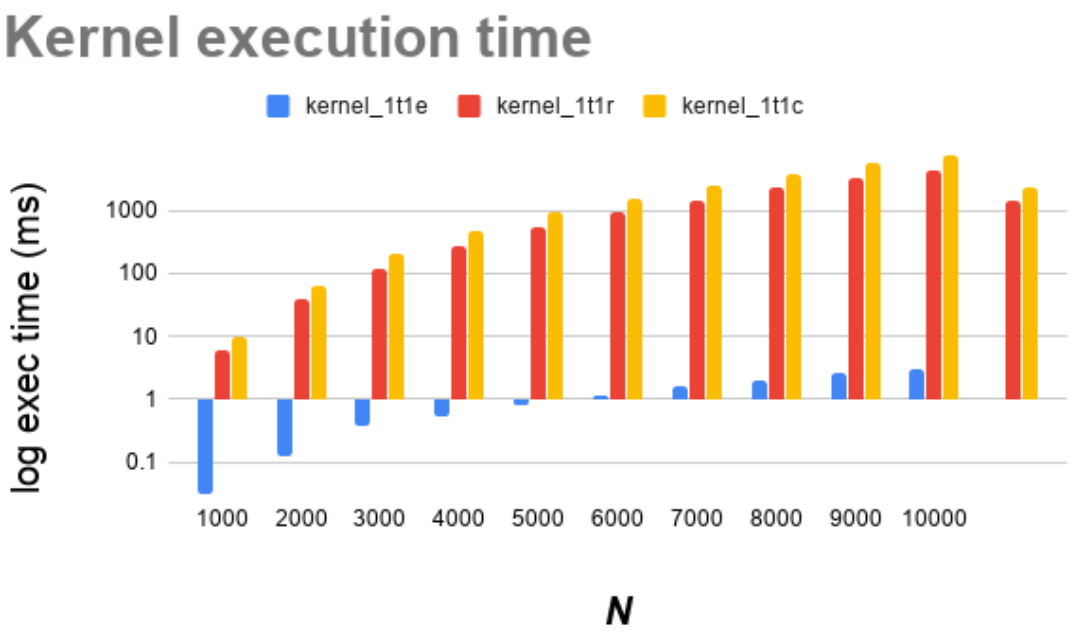

Figure 1: Log execution time of different kernels.

We also analyse the computation at the hardware level of each kernel. The following code snippets from each kernel show how each thread computes for c_{ij}:

```
__global__ void kernel_1t1e(float *A, float *B, float *C, int WIDTH){
    int rowId = threadIdx.y + blockDim.y * blockIdx.y;
    int colId = threadIdx.x + blockDim.x * blockIdx.x;
```

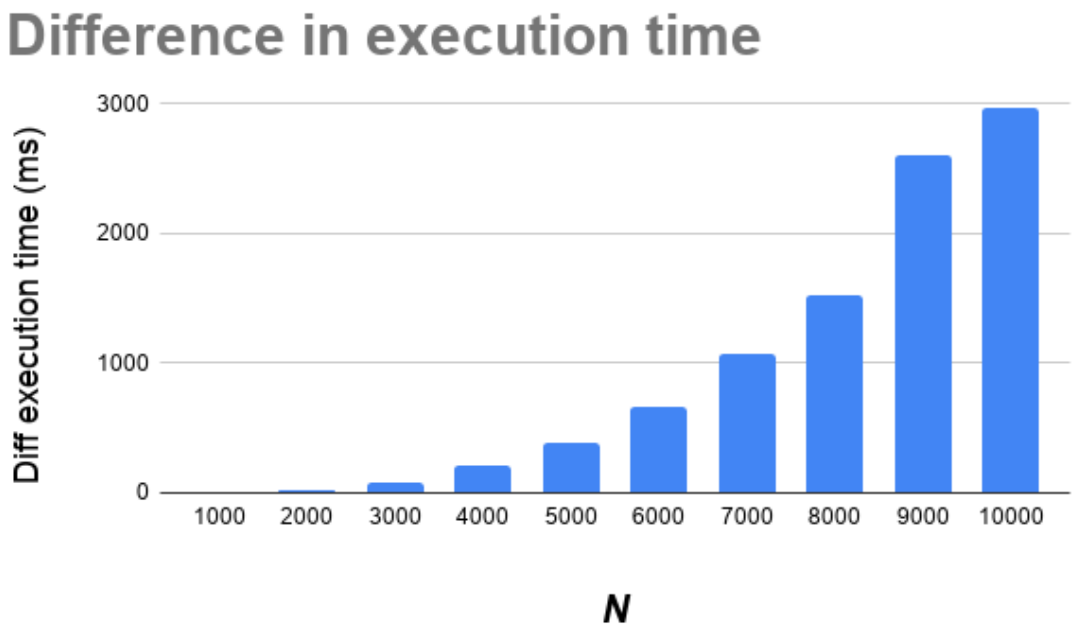

Figure 2: Difference in execution time between `kernel_1t1r` and `kernel_1t1c`.

```
    int elemId = colId + rowId*WIDTH;
    if (rowId < WIDTH) C[elemId] = A[elemId] + B[elemId];
}

__global__ void kernel_1t1r(float *A, float *B, float *C, int WIDTH){
    int rowId = threadIdx.y + blockDim.y * blockIdx.y;
    int elemID;
    if(rowID < WIDTH) {
        for(int i = 0; i<WIDTH; i++){
            elemID = i + rowID*WIDTH;
            C[elemID] = A[elemID] + B[elemID]; ...}

__global__ void kernel_1t1c(float *A, float *B, float *C, int WIDTH){
    int colId = threadIdx.x + blockDim.x * blockIdx.x;
    int elemID;
    if (colId < WIDTH) {
        for(int i = 0; i<WIDTH; i++){
            elemID = i + colId*WIDTH;
            C[elemId] = A[elemId] + B[elemId]; ...}
```

We see from the snippets above that each kernel has three global memory accesses (reading twice from **B** and **A** and then writing once to **C**). Moreover, kernel `kernel_1t1c` has more floating point operations compared to the other two, with eight operations including

the `if` statement while only six for `kernel_1t1c` and `kernel_1t1r`. This makes the CGMA of the former kernel equal to 2.33 while the other is only 2. We recall that higher CGMA implies a more efficient parallelism in terms of thread computation and global memory access.

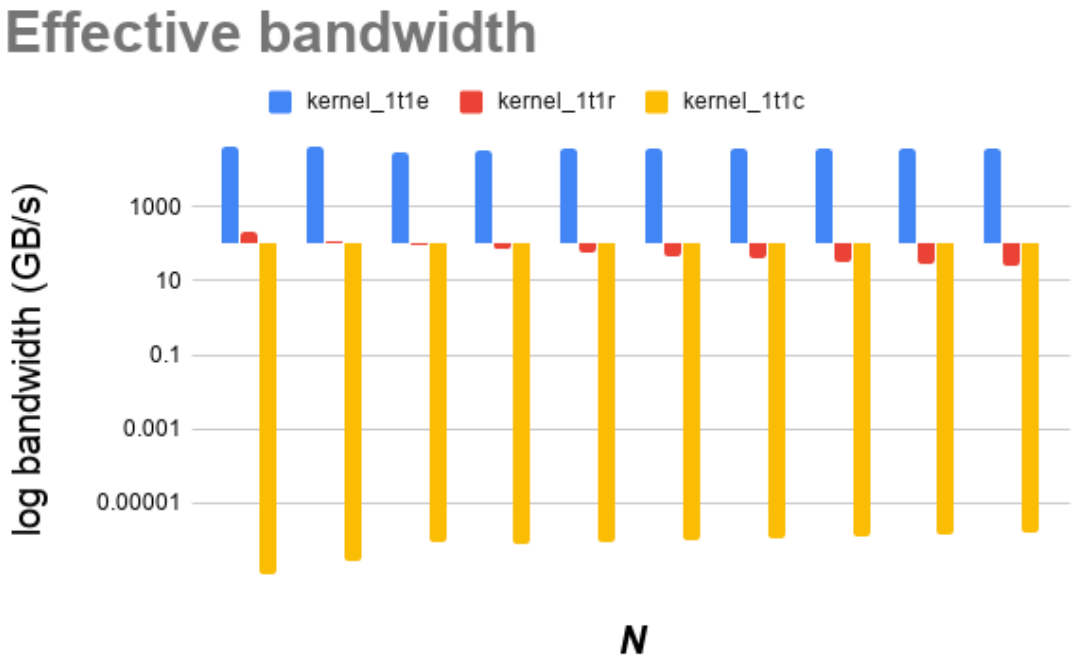

Figure 3: Effective bandwidth of different kernels.

Another metric that we can analyze is its memory bandwidth [Harris, 2012]. The theoretical bandwidth of my device is around 294GB/s [HWCompare.com, nd]. Figure 3 shows the experimental memory bandwidth of the kernel programs given the three constant global memory accesses. Clearly, we expect `kernel_1t1e` to have the highest bandwidth since there are N^2 memory access at a given time. On the other hand, `kernel_1t1c` has the lowest bandwidth despite having the same number of global memory access as in `kernel_1t1r`. This is because `kernel_1t1c` takes too much time to read and write to the global memory.

4 Conclusion

We showed how matrix addition can be parallelized in three different ways using GPU computing. We can delegate task such as element-wise addition of matrices to each thread inside the GPU hardware as in `kernel_1t1e`. The obvious `for loop` for CPU implementation can be avoided by leveraging the huge number amount of parallel threads inside a GPU hardware. The element-wise implementation of matrix addition in `kernel_1t1e` is much faster than that of `kernel_1t1r` and `kernel_1t1c` where each thread is asked to compute for the sum of each element in a row or column, respectively. The difference in the performance can be explained from how the global memory of the GPU is accessed and arranged inside the hardware. However, `kernel_1t1e` has been observed to have the highest throughput among the three kernels.

References

[Harris, 2012] Harris, M. (2012). How to implement performance metrics in cuda c/c++.

[HWCompare.com, nd] HWCompare.com (n.d.). Geforce gtx 1080 vs geforce gtx 1660 ti.