Graphics Processing Units - Fall 2018
Programming Assignment 1

Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

1. **Experiment 1**
   Table 1 and Figure 1 show a comparison of the execution time CPU vs. GPU for different sizes of the matrix M with a fixed number of iterations $i = 1\,000$. In figure 1, the primary vertical axis (time) has a logarithmic scale, whereas the secondary vertical axis (speedup of the GPU implementation with respect to the CPU) has a linear scale.

Table 1

Matrix size n	Execution time $t(s)$ GPU	CPU	Speedup S
1 000	0.29	10.42	35.93
2 000	0.67	40.35	60.22
4 000	2.11	150.42	71.29
8 000	7.71	580.67	75.31
16 000	30.30	2,276.14	75.12

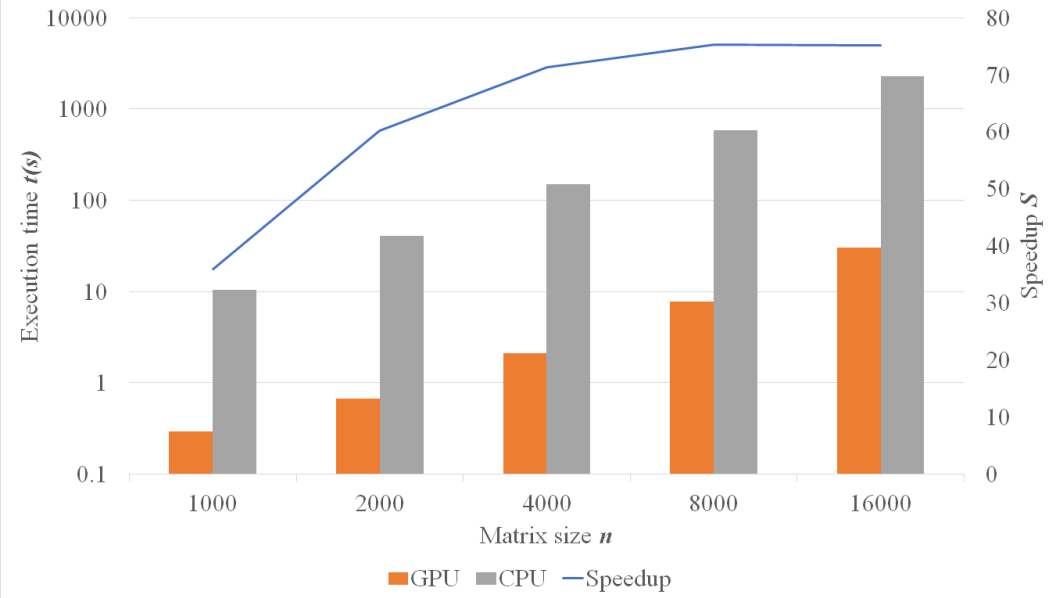

Figure 1

2. **Experiment 2**
Table 2 and Figure 2 show a comparison of the execution time CPU vs. GPU for different number of iterations i with a fixed-size matrix $M \in \mathbb{R}^{1\,000 \times 1\,000}$. In figure 2, the primary vertical axis (time) has a logarithmic scale, whereas the secondary vertical axis (speedup of the GPU implementation with respect to the CPU) has a linear scale.

Table 2

No. of iterations i	Execution time $t(s)$ GPU	CPU	Speedup S
1 000	0.29	10.42	35.93
2 000	0.42	20.56	48.95
4 000	0.69	37.80	54.78
8 000	1.22	72.21	59.19
16 000	2.23	141.34	63.38
32 000	4.25	277.85	65.38

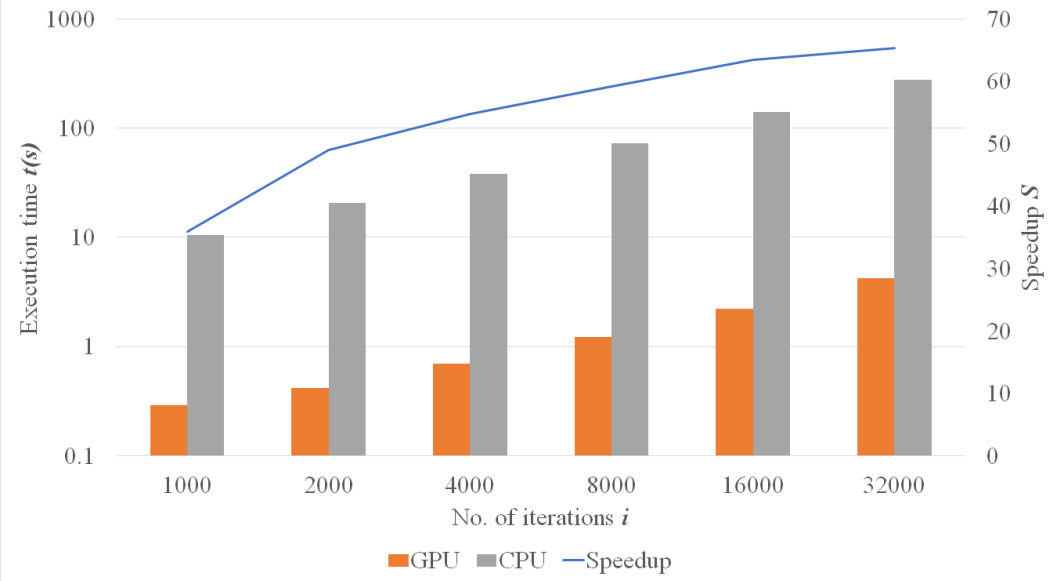

Figure 2

3. **Conclusions**

(a) When is GPU usage more beneficial (at which n)? And why?
Looking at the results from *Experiment 1*, we can see that using the GPU is more beneficial for values of $n \geq 8\,000$. For such values the speedup of the application is more or less stable at a value of 75x compared to the CPU version.
Also, if we look at the profiling results (available in *nvprof.txt*) we notice that the allocation of memory in the *device* using `cudaMalloc()` takes a constant amount of time (approximately $200ms$) regardless of the size of the problem. This means that for smaller problems where launching the actual kernels does not take a lot of time (e.g. with $n = 1\,000$ launching the kernels only takes $21ms$) the cost of memory allocation cannot be overcome as easily.
To further improve the speedup achieved with the GPU (for any problem size in general), it would be necessary to perform more computationally intensive operations. If we take for example the experiments with $n \geq 4\,000$, we notice that the time it takes to launch the kernels (`cudaLaunch` in *nvprof.txt*) is approximately half of the time required to perform memory transfer operations (`cudaMemcpy` in *nvprof.txt*). This means that under these conditions our application is *memory bound*, i.e. limited by the recurrent accesses to memory and the data transfers between *host* and *device*, and not by the computations performed on the data.

(b) When is the speedup (i.e. time of CPU version / time of CUDA version) at its lowest? And why?
The speedup takes its lowest value at $i = 1\,000$ and $n = 1\,000$. As it was briefly described in (a), the overhead associated with transferring the application to run in the GPU accounts for a considerable portion of the overall running time when the problem is too small. Table 3 shows a summary of the profiling results (available in *nvprof.txt*) for this experiment. As we can see, memory allocation accounts for almost 60% of the time required in the *host* to make API calls to the *device*. In second place we get data transfer (memory copy) with a little over 30%, and in third place the actual kernel execution with only 6%.

Table 3: API Calls ($i = 1\,000, n = 1\,000$)

Time(%)	Time	Calls	Name
59.97%	$212.44ms$	3	`cudaMalloc`
31.56%	$111.81ms$	$1\,004$	`cudaMemcpy`
6.08%	$21.555ms$	$1\,000$	`cudaLaunch`

(c) When is the speedup at its highest? And why?
The speedup takes its highest value at $i = 1\,000$ and $n \in \{8\,000, 16\,000\}$. As it was briefly described in (a), these two experiments have reached the maximum speedup possible under the current implementation due to the *memory bound* nature of the application. Table 4 shows a summary of the profiling results (available in *nvprof.txt*) when $n = 8\,000$. As we can see, data transfer (memory copy) is now in first place accounting for almost 65% of the time required in the *host* to make API calls to the *device*. With 33% the kernel execution comes in second place, leaving memory allocation as a far third with only 2.5%.

Table 4: API Calls ($i = 1\,000, n = 8\,000$)

Time(%)	Time	Calls	Name
64.27%	$4.94s$	$1\,004$	`cudaMalloc`
33.00%	$2.54s$	$1\,000$	`cudaMemcpy`
2.53%	$194.59ms$	3	`cudaLaunch`

(d) Which has more effect: number of iterations or the problem size? And why?
The problem size has more effect in the performance of the application and the speedup that can be achieved using the GPU. This affirmation can be easily confirmed by looking at the code of the *host*. Inside the `for` loop that controls the number of iterations (lines 201-204 in *heatdist.cu*), we have both a kernel launch and a memory copy command. If it were the case that *only* the kernel launch were in the loop, then all the calls would be asynchronous and several kernels could (potentially) be executed in parallel in the *device*. However, the call to `cudaMemcpy` enforces synchronization after each kernel launch to ensure data integrity before the transfer can begin. Therefore, the execution of all the kernels is implicitly serialized by `cudaMemcpy`. In terms of the application this behavior definitely makes sense, since we want one kernel to finish performing all the manipulations on the data before it can be copied and used by the next kernel.
In contrast, the size of the problem determines the size of the grid that will be passed to the kernel and is independent of the number of iterations. Therefore, a bigger problem will translate into more occupancy in the GPU which in turn means (potentially) more parallelized execution.
Also, the results reported in Tables 1 and 2 support these conclusions. If we take a look at *Experiment 2* with $i = 32\,000$ and $n = 1\,000$, where the number of iterations is considerably bigger than in *Experiment 1*, the speedup achieved is only 65%, which is about 10% less than the best results for *Experiment 1*.