Parallel Computing - WA3

1st José Carvalho
Departamento de Informática
Universidade do Minho
Braga, Portugal
PG53975@alunos.uminho.pt

2nd Miguel Silva
Departamento de Informática
Universidade do Minho
Braga, Portugal
PG54097@alunos.uminho.pt

***Abstract*—**
This is the final report of the Parallel Computing work assignment. On this document we expose our optimisations upon the code provided by the professors, comparing the multiple versions of the code with each other via test result analysis and providing meaningful conclusions focusing on this phase's CUDA implementation.

***Index Terms*—Parallel Computing, C, Vectorization, SIMD, OpenMP, CUDA, performance, gain**

I. Introduction

With the conclusion of the final stage of the work assignment, we now have three different versions of the code provided by the professors.

The previous two are the ones we wrote about on the previous reports, which means the first one is an optimised sequential version that takes advantage of vectorization and other tweaks to improve execution time, and the second one uses *OpenMP*'s capacities to obtain parallelization via threads and tasks.

As for the third version, which is the one specially made for this phase, we opted to use *CUDA* to take full advantage of *NVIDIA* graphics capacity. We chose this option instead of *MPI*.

MPI is better for distributed memory parallelism, passing messages between the multiple nodes on the cluster. This could have been a very valid approach as well. But we believed that implementing *CUDA* on the code would not be that much more complex and due to the *GPU*'s immense amount of processing units, we could have a better performance depending on our workload distribution.

In this document we will briefly explain some of the most important aspects of both previous code versions and the new *CUDA* version. We will also present multiple tests we created to measure each version's performance, comparing these results and finishing the document with hopefully meaningful conclusions and critiques.

II. Implementations

Before explaining what we did for this phase it is important to remember our work on the previous stages.

A. Hot Spots

As a starting point it is important to list the bottlenecks present in the code. These bottlenecks are responsible for taking up most of the execution time and need to be the focus of our optimisations.

We classified these functions as our hot spots:

- *Potential*
- *ComputeAccelerations*

The complexity of these functions are *O(N^2)* for *Potential* and *O($NLogN$)* for *ComputeAccelerations()*. Their complexity is the reason why they take up most of the execution time specially for high values of *N*, since we are working with values such as 5000 their execution time really ends up consuming almost the entire execution time of the code.

These functions are also evoked multiple times, which only further increases the percentage of time spent executing them.

B. Sequential

Our sequential code is by no means the best, but when we developed it we focused on improving memory hierarchy, loop unrolling, maths simplifications, pipeline and vectorization.

The loop unrolling part is simple and as explained on the first report, we changed the structure of as many loops as we could to implement it.

For memory hierarchy we adapted the array *rij* into 3 different variables on *computeAccelerations*, speeding up the reading of registers. We know that this was not sufficient and our grade reflects that, still, as mentioned on the first report we also tried other approaches but we could not implement them in time and/or they caused our execution time to increase, which was not the objective.

Maths simplifications consist of the majority of our optimisations. We mostly changed complex operations containing multiplications into simpler operations.

We also removed the *pow* function whenever we could as that function ended up taking too much time. This is likely due to the fact that pow implementation comes from an external library and the implementations of the *pow* function itself, as it works with floating point numbers and utilises logarithms. Here is the code the function utilises:

$$pow(x, y) = e^{(ylog(x))}$$

The multiple code simplifications vary, one of them simply removed constants, whilst others were far more complex and require us solving equations, that process is explained step by step on the first report so we will not repeat it here.

The way we handled vectorization was using the *gcc* compiler's capabilities via code legibility and flags to implement it for us. This ended up not being the best approach and in retrospect maybe we could have done manual vectorization. Either way, to use this type of vectorization which uses 32 bytes *SIMD* instructions we had to re-organize our data structures, more specifically r,v a and F matrix.

The matrix had to be changed into arrays with starting in an index multiple of 32, which means that they had to start on these positions: 0,32,64,96,128, (...), until the last element multiple of 32 can fit on memory. So our variables became this:

```
double r[MAXPART*3] __attribute__((aligned
    (32)));
double v[MAXPART*3] __attribute__((aligned
    (32)));
double a[MAXPART*3] __attribute__((aligned
    (32)));
double F[MAXPART*3] __attribute__((aligned
    (32)));
```

Due to this change, we now have to update every loop that worked with this arrays, to iterate over 3 positions each iteration.

In terms of results our vectorization managed to reduce instructions 1.467 times which is less than the theoretical value of 4 times (doubles occupy 8 bytes and 32 / 8 = 4).

As for flags, we used:

- -O3: enables aggressive compiler optimisations like inlining functions, loop unswitching, and some vectorization
- -Ofast: enables all O3 optimisation and disregards safety standards maximising execution speed. Can cause floating point errors.
- -pg: enables profile information generation
- -funroll-loops: applies loop unrolling
- -mavx: allows usage of Ad- vanced Vector Extensions (AVX) instructions for vectorization.

Throughout the code we focused on reducing the values that compose the equation that calculates execution time: $t_{\text{exec}} = \#I \cdot (\#CPI_{\text{CPU}} + \#CPI_{\text{MEM}}) \cdot T_{\text{cc}}$.

As stated on the first report, this version of the code, has reduced cache misses by 3.5 times, 40 times less instructions executed, 32 times less clock cycles and most impressively 31 times less execution time. The CPI grew by 0.1 due to instructions now being more dependent on each other.

C. Open-MP

For the second stage of the work assignment we implemented parallelization using *Open-MP*. This API provides a plethora of options to implement parallelization in C/C++ or Fortran and we opted to use it's tasks in our definitive version. We also have a version that relies on threads, but it will not be detailed here because it's results were worse.

Before going in detail into our implementation of this version it is important to refer what were the zones in our code where we could apply parallelization. This alongside with every detail mentioned on this subsection is explained in more detail in our second report.

We focused on three parallelizable zones:

- The functions mentioned on II-A due to each of them being mostly composed of a cycle that can be split into parallel workloads via tasks.
- The main function's loop cycle, as it contains parts that are independent from each other and the only part that requires sequential execution is the *VelocityVerlet* function with the rest of the body of the cycle is unnecessary for the subsequent iterations.

Also it is important to note that on this phase we had to be extra careful regarding data races and other issues specific to parallel coding.

The reasons why we used tasks is that firstly the parallel zones are inside functions that will be evoked multiple times, without tasks we would need the create all the threads over again every time which affects performance. And lastly, the main function's loop is very difficult to parallelize without tasks.

Starting with *computeAccelerations*, this function's goal is to update values of the *Acceleration* array. These values are calculated based on the values inside of the *Position* array and the hot spot is located on the second loop inside the function. We can not use a *taskloop* at the outer loop or inner loop, because in both scenarios, we would cause a data race on the Acceleration array, so we had to think of a different solution. We created a matrix with replications of the *Acceleration* array on it's lines, with the number of lines being the number of threads created by Open-MP retrieved by the *omp_get_num_threads* function. Every value of the matrix starts at zero.

So, every iteration of the cycle works with a specific line of the matrix with the replicas. We only have to calculate what is the limits each task handles when dividing the work, the issue is that due to the complexity of the function being $O(n \log n)$ the work is not divided equally with the latter divisions handling less work the first ones. In the end all we need to do is sum the values on the replica matrix, associating them to the original array and free the memory allocated.

On the *Potential* function we only had to solve a data race caused by the *Pot* variable. We split the iteration of the main cycle into 2 different tasks. This division is done by creating 2 different variables and giving each division one of those independent variables. In the end we sum both variables and we get the final value of Pot, emulating a reduce operation. This time since the complexity is $O(N^2)$ the work gets divided equally amongst the tasks. The reason why we did not use a *taskloop* on *Potential* like we did on *computeAccelerations*, is due to the paralization that we have on main function. This paralization ensures the execution of *Potential* and *computeAccelerations* at the same time and

since *computeAccelerations* needs more time, we prioritise the execution of its own tasks.

Lastly, for the main function, we created a task that executes the independent part of loop iterations, the preservation of the values from the Velocity and Positions arrays and the variable Press is ensured by the primitive first private, that creates a copy of these arrays and variable with their most recent values. Here we had to be careful with the values of *Tavg* and *Pavg* as they can easily create a data race, that was solved by an array that contains the values after each iteration of the cycle, and then we sum the values once again emulating a reduce.

One last optimization was obtained via the usage of the constructor *pragma omp simd* to vectorize part of the code of the functions *PotentialDivision*, *PotentialMath* and *computeAccelerationsDivision*.

An important idea to reference for this version of the code is that one of our main goals was minimizing the effects of Amdahl's Law, that states that the speedup of a program is limited by the sequential portion of the code. We implemented various strategies to optimize parallelizable portions of the code, enhance parallel processing capabilities, and ultimately improve overall system performance.

To finish this section we would like refer our speed up. The theoretical optimal speed up was equal to the number of threads until we reached 20 which is the maximum gain. This version managed to attain a gain that is pretty much equal to the maximum value for 20 threads and in fact we even surpassed the theoretical gain for less threads, likely due to the usage of the *simd* constructor. For more than 20 threads we could not maintain the speedup, and lost performance, we believe that was caused by scalability not being perfect, task granularity increasing, and the effect the aforementioned Amdahl's law stating that the maximum improvement in speed will always be determined by the code's most significant bottleneck which always limits the maximum speedup at 20.

D. CUDA

In relation to our *Cuda* version, we tried to simplify as much as possible. We only used *Cuda* on the hot spots, i.e. inside functions *computeAccelerations* and *Potential*.

Before explaining the code itself, it is important to point an issue that we faced because of data-races. The *GPU* from the *SeARCH* cluster is a *Nvidia 7900GT* and this *GPU* is old (2006). This GPU does not support atomic operations with double precision floating points by default. Since our solution needs this kind of atomic operations, more precisely for sums and subtractions, we had to figure out a solution to this problem.

To solve this, we took a look at the documentation on the official *NVIDIA* website (*Link Atomic Operation Documentation*). On this link we found an implementation of the code we needed and could use, we present it on the section VI-A.

One more important note is that we used the default values for the block size and threads per block. Both default values are 256. The idea behind this is to create as many threads as possible in order to better handle the scalability. This lead to us creating 65 536 threads. Even though, when we were testing, we have used other values, in order to compare the performance of our *CUDA* version in relation to the number of created threads.

Starting with *Potential* function. This function initially has a complexity of $O(N^2)$. On our solution, each *thread* from *GPU* will execute one iteration from the outer cycle, so we managed to reduce the complexity from $O(N^2)$ to $O(N)$.

To make this, we have to copy the values from *Position* array and create a place on *GPU* memory where we can store the calculus. This place will be a double and will be initialised with the value 0. *PotentialDivision* is the function that each *thread* from the *GPU* will execute. This function will sum all the values from *PotentialMath* results. Next step is to sum all the values from all threads and to achieve that we used the *ourAtomicAdd* function. We had to use this function in order to write the result on the shared variable between threads without having a data-races. At the end, we copy the result value from *GPU* memory to the *CPU* memory and free the allocated memory on *GPU*.

In relation to the *computeAccelerations* function, we have a similar approach as *Potential* function. This function, originally has a complexity of $O(n \log n)$ and we were able to make that each *thread* from the *GPU* execute a iteration from the outer cycle, resulting in a complexity of $O(\log n)$.

To this approach, we have copied the *Position* array to the *GPU* and create an array that will represent the *Acceleration* array on *GPU*. This last mention array will have all its positions initialised to 0. On *computeAccelerationsDivision*, the calculus are made using private variables and when we want to store the results on the array that is shared, we used *ourAtomicAdd* and *ourAtomicsSub* functions, in order to write the results on the *Acceleration* array on *GPU* memory. To finalise, we copied the *Acceleration* array from *GPU* to *CPU*, free the allocated memory on *GPU* and multiply each element of the *Acceleration* array for 24.

After implementing this version, we noticed that most of the threads that we create, would not make anything. So we tried to implement a new version that uses more threads, but the code's performance got worse. Our idea was to, instead of each thread executing the one iteration of the outer cycle on both function already mentioned, the threads will only make half of the job. So each iteration of the outer cycle will be executed by 2 threads, one doing the first half and the second one completing the rest. The reason why our performance got worse, was because of the atomic operation, since we have more threads trying to write on the same place on memory. On section IV we will present and discuss the results between these two *CUDA* versions.

In relation to the theoretical gain, we could say that our gain could be the number of threads created, but there are two main problems. The first one is that not all the code is going to be executed on the *GPU*, so we would never achieve the speed up proportional to the number of threads. The second one, and maybe the most critic, is related with the memory, since when we transfer content from *host* to *GPU* we have to

use the *PCI*, which nowadays is still a bottleneck.

Now we are going to list some optimisations that could improve our performance.

1) Memory: Memory is the main reason why our *CUDA* performance is worse than *Open-MP*. The results are shown on section IV-B. The reason behind this is because the hot spot functions are called many times, and each time we are copying the information to the *GPU*. With this in mind, one improvement that we could have made to both of our *CUDA* versions, would be to try and reduce those constant copies, since, for example, we are copying the same array twice, more specifically, the *Position* array.

2) Atomic: Like already mentioned, we used atomic functions in order to avoid data-races. One improvement could be to stop using atomic functions, and instead of multiple threads writing in the same place, we could change this in so that each thread writes on its own place. But this solution has a problem and that was the reason why we did not implemented it. As we already referred, working with the memory causes a bottleneck, and to make this solution viable, we would need to allocate more space on *GPU* memory to create independence between threads, and at the end, copy all those values in the *GPU*'s memory to *host* memory. This operations are really slow.

3) Shared Memory: One optimisation that we could have implemented was shared memory between the *GPU* threads. This memory is really powerful and would be useful because we are accessing the same memory addresses multiple times, which is the case on *Position* array in both hot spot functions. Unfortunately this shared memory is limited to a maximum size and when the variable N is equal to 5000, it does not have enough space to store all of *Position* array values, so it is not possible to use this shared memory.

III. Tests

When it came to testing the code, we had 4 versions to compare:

1) Sequential.
2) Open-MP.
3) Cuda v1.
4) Cuda v2.

For the tests themselves, we decided to run each version of our program for different values of N. Those values were:

- 2500
- 5000
- 10000 (*)
- 20000 (*)

(*) **Note**: For these tests, we had to change the variable *MAXPART* accordingly.

For each assigned value of N, we ran our multiple version multiple times and got the execution time of each execution. The table I shows how many times we ran our program for each N and for each version. It is important to note that for N equal to 10 000 and 20 000, we opted to reduce the times we ran the versions due to the amount of time that we needed to wait. This is also the reason why we executed the Sequential version less times when compared to the others.

TABLE I
Tests 1

Version \N	2500	5000	10000	20000
Sequential	20	3	3	0 (*)
Open-MP	20	24	7	6
Cuda v1	20	24	7	6
Cuda v2	20	24	7	6

(*) **Note**: Our sequential version for N equal to 20000 reached the 10 minutes limit, so we considered an arbitrary default value of 700 seconds.

After all this, we decided to make one last test dedicated to scalability. We only ran this test for the *Open-MP* and the first *CUDA* version of our code. We only utilise the first version of *CUDA*, since it is the version with better results. We established a fixed value for N, which was 5000, and then we changed the number of threads created on the both versions. The threads for *Open-MP* version are 5,10,20,40 and for the *CUDA* version are shown on the following table:

TABLE II
Tests 2

Test	Number of Blocks	Number of Threads per Block	#Total Threads
#1	256	256	65536
#2	512	512	262144
#3	128	128	16384
#4	80	80	6400

At the end, we collected the execution times and created files that contained the data collected. With those times, we analysed metrics like average, maximum and minimum execution times and speedup so that we could discuss which version performed better.

IV. Results presentations

A. Presentations

In the figures 1 and 2 we have 4 histograms that show us the following metrics:

- Average time.
- Maximum time.
- Minimum time.
- Median time.

X axis values legend:

- Avg - Average
- Max - Maximum
- Min - Minimum
- Med - Median
- S - Sequential
- O - Open-MP
- C1 - CUDA v1
- C2 - CUDA v2

The figure 3 shows the average time and the speed up between all versions. The images 4 and 5, shows the differences of performance when we change the number of threads created. It is important to note that when we when we have 80, 126, 256 and 512 threads per block on the *GPU* graphs, the real amount of threads is equal to the square of those numbers (80^2, 126^2,...).

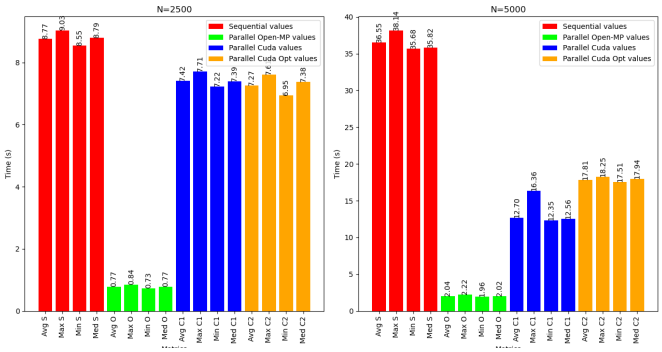

Fig. 1. N=2500 and N=5000 results

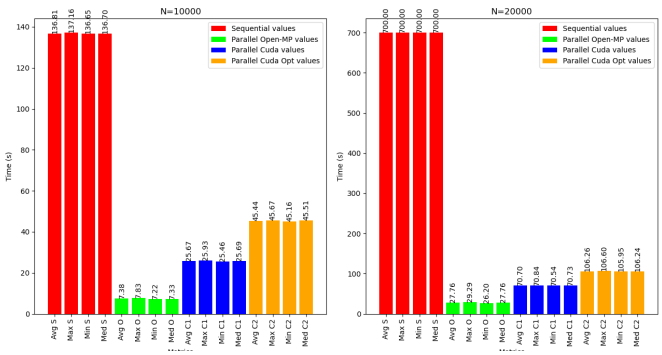

Fig. 2. N=10000 and N=20000 results

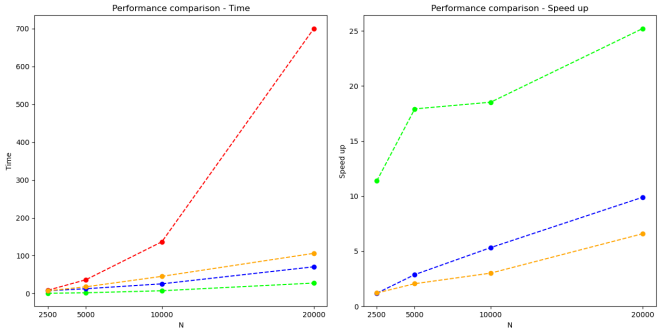

Fig. 3. Time and Speed up comparison

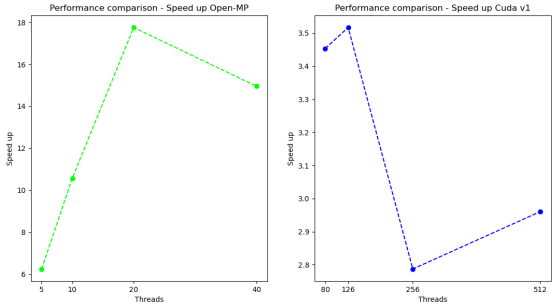

Fig. 4. Scalability comparison 1

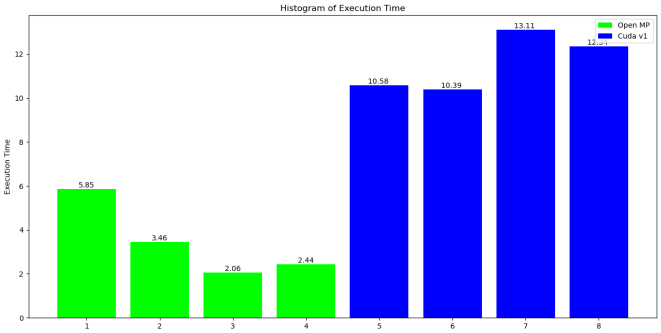

Fig. 5. Scalability comparison 2

The theoretical gain when N=5000, for each situation would be:

TABLE III
COMPARASION GAIN THEORETICAL X REAL - OPEN MP

#Threads	Theoretical Gain	Actual Gain
5	5	6.25
10	10	10.58
20	20	17.75
40	40	14.96

TABLE IV
COMPARASION GAIN THEORETICAL X REAL - CUDA V1

#Threads	Theoretical Gain	Actual Gain
$80 \cdot 80$	6400	3.45
$126 \cdot 126$	15876	3.52
$256 \cdot 256$	65536	2.79
$512 \cdot 512$	262144	2.96

B. Analysis

After analysing the graphs shown on section IV-A, it is clear that our best version is the one that relies on Open-MP (second stage of the assignment).

It is also obvious why the worst version is the sequential, since it is the only version that does not have parallelism implemented.

We already mentioned this on section II-D, but our second version of *CUDA* is slower than the first one, because we have more threads trying to write on the same place in memory.

The reason why our *CUDA* versions are slower than the *Open-MP* version is due to the *PCI*, that is really slow. *PCI* is the connection between the *host* and the *GPU*, where the data is transferred between these two components. This proves what *nvprof* showed, because 80% of the time is spent on copying the values from *host* memory to *GPU* memory. The results from *nvprof* are on section VI-B.

Another observation we can point out, is that as we increment in the value of N, the speed up values obtained for the *Open-MP* version remain far better than the *CUDA* version. This is due to the fact that the value of N is proportional to the memory copied from *host* to *GPU*, which is why the *Open-MP* version can scale as much as *CUDA* versions.

An important final note is that the last two figures (4, 5), shows that when we reduce the number of threads on *GPU* we get a better performance, even though, the *GPU* results remain worse than the *Open-MP* results.

V. Conclusion

In conclusion, we believe that we managed to accomplish our goals, we managed to implement all the versions and were specially successful in the second stage of the work assignment.

We also managed to create graphs that showcase the capabilities of each version of the code and managed to compare them in a competent manner.

Still we have to criticise ourselves on some aspects, starting with our performance in the first stage because our code was not the best.

We would have loved to have a better performance in our *CUDA* code as well, but sadly we did not manage to improve the memory copies from *host* to *GPU* and the *OpenMP* version ended up performing the best.

As a whole we believe that this was an interesting project, mostly because we learned how to apply what we do with academic examples in a practical manner on real code. Still, we believe the criteria behind the evaluation sometimes was not the best and that the professors had a tendency to not understand our approach, which is normal. There are multiple other groups doing this project which means many implementations of the same exact project and that makes it difficult to memorise and understanding each implementation to the fullest.

We also believe that using the cluster was sometimes painful due to it getting easily compromised by colleagues running code and taking up 100% of the resources. For example, there were times where we were not able to compile our code and there was a day where we could not use the *SeARCH GPU*. Also due to the time limit imposed on process we could not run our sequential code for N = 20000 which would have been helpful for the measures we analysed on this report.

VI. Attachments

A. Atomic functions

```
__device__ double ourAtomicAdd(double*
    address, double val)
{
   unsigned long long int* address_as_ull =
                         (unsigned long long
                            int*)address;
   unsigned long long int old =
       *address_as_ull, assumed;

   do {
      assumed = old;
      old = atomicCAS(address_as_ull, assumed,
                  __double_as_longlong(val +
                       __longlong_as_double(assumed)));

   // Note: uses integer comparison to avoid
       hang in case of NaN (since NaN != NaN)
   } while (assumed != old);

   return __longlong_as_double(old);
}

__device__ double ourAtomicSub(double*
    address, double val)
{
   unsigned long long int* address_as_ull =
                         (unsigned long long
                            int*)address;
   unsigned long long int old =
       *address_as_ull, assumed;

   do {
      assumed = old;
      old = atomicCAS(address_as_ull, assumed,
                  __double_as_longlong(__longlong_as_dou
                      - val));

   // Note: uses integer comparison to avoid
       hang in case of NaN (since NaN != NaN)
   } while (assumed != old);

   return __longlong_as_double(old);
}
```

B. Nvprof results

```
==28017== Profiling application: ./bin/MDcuda.exe
==28017== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   73.50%  6.15087s       202  30.458ms  30.407ms  30.480ms  computeAccelerationsDivision(double*, double*, int)
                   26.33%  2.20358s       201  10.963ms  10.375ms  11.888ms  PotentialDivision(double*, double*, int, double)
                    0.11%  8.8098ms       403  21.860us  21.408us  30.913us  [CUDA memcpy HtoD]
                    0.05%  4.0717ms       403  10.103us  1.5350us  28.096us  [CUDA memcpy DtoH]
                    0.02%  1.3957ms       403  3.4630us  1.7920us  5.4400us  [CUDA memset]
      API calls:   96.41%  8.39618s       806  10.417ms  44.817us  30.600ms  cudaMemcpy
                    3.00%  261.66ms       806  324.64us  3.9390us  213.50ms  cudaMalloc
                    0.48%  41.651ms       806  51.675us  5.7950us  154.96us  cudaFree
                    0.05%  4.6980ms       403  11.657us  8.5410us  40.440us  cudaLaunchKernel
                    0.05%  3.9335ms       403  9.7650us  4.7480us  58.169us  cudaMemset
                    0.01%  553.26us         1  553.26us  553.26us  553.26us  cuDeviceTotalMem
                    0.00%  341.41us       101  3.3800us     370ns  131.08us  cuDeviceGetAttribute
                    0.00%  220.46us         1  220.46us  220.46us  220.46us  cuDeviceGetName
                    0.00%  15.672us         1  15.672us  15.672us  15.672us  cuDeviceGetPCIBusId
                    0.00%  4.2210us         3  1.4070us     508ns  2.8610us  cuDeviceGetCount
                    0.00%  2.3270us         2  1.1630us     492ns  1.8350us  cuDeviceGet
                    0.00%     997ns         1     997ns     997ns     997ns  cuDeviceGetUuid
```

Fig. 6. Nvprof results