AGH University of Science and Technology

Laboratory report 1

Authors: Pedro Alexandre Simões dos Reis,
Iñaki Urrutia Sánchez
Subject: Introduction to CUDA and OpenCL
Year: 2019/2020

Contents

1 Task 1

1.1 VectorAdd behaviour with different size of data

As it can be easialy observed in the graphic, when the Number of Elements in the vectors that are being added is increased, the execution time of the function vectorAdd is increasing exponentially.

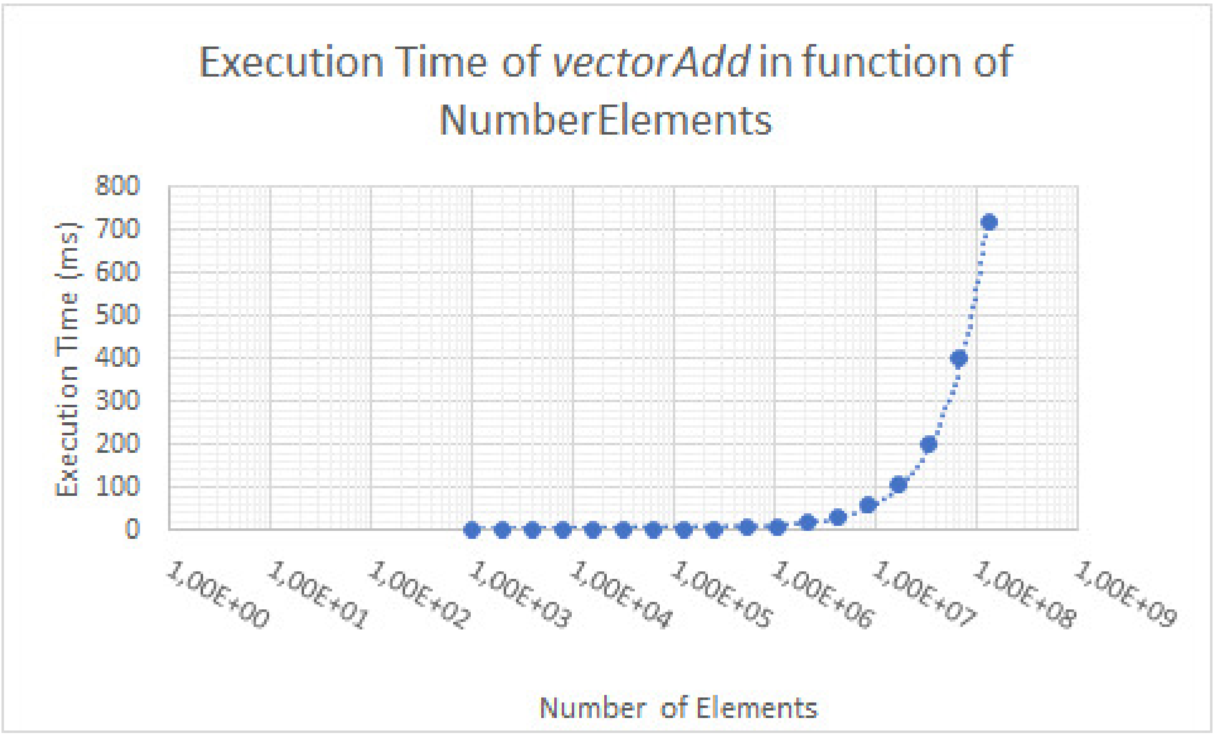

Figure 1: GPU execution time vs number of elements

The function cudaMallocManaged allows us to allocate memory in the Unified Memory space, which is a single memory address space that is accessible from both the CPUs and the GPUs in ths system. This allows data to be used by both the CPUs and the GPUs.

In the graphic we can note that the execution time of cudaMallocManaged is almost constant (it only fluctuates in a small range of ms).

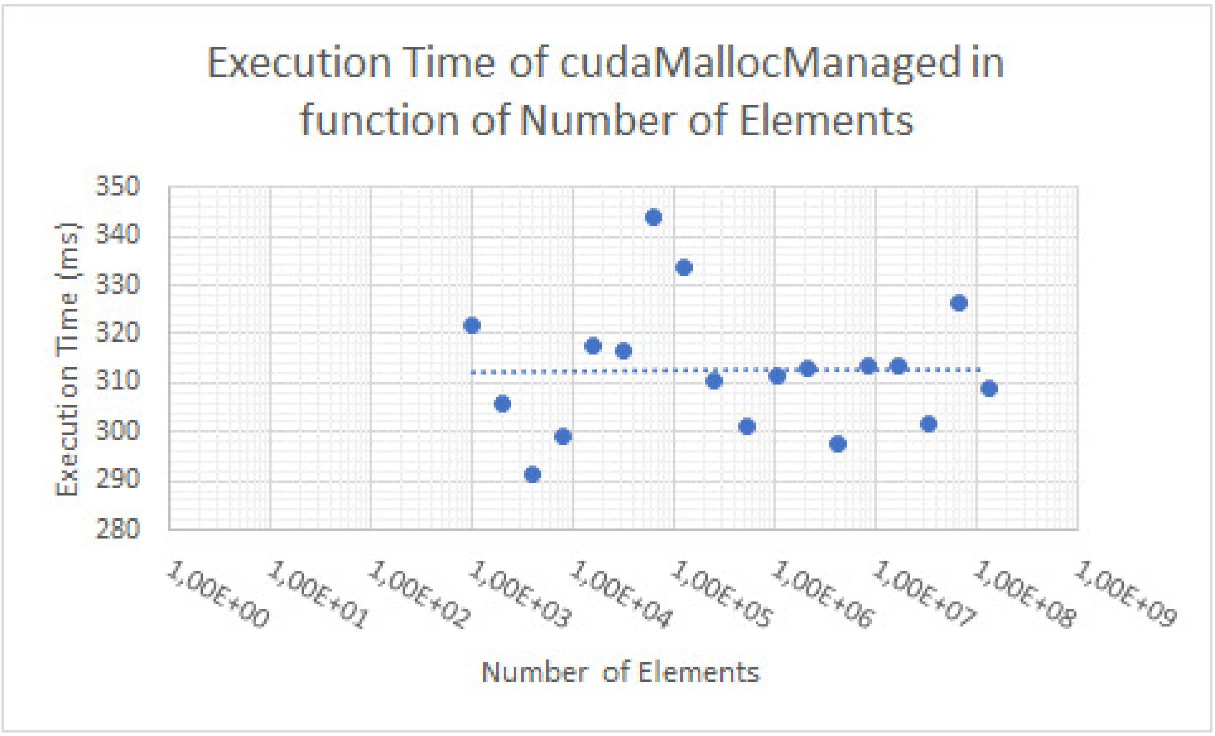

Figure 2: cudaMallocManaged execution time vs number of elements

As we can see, both host to device and device to host times increase exponentially as the number of elements growths, as in the case of the execution time.

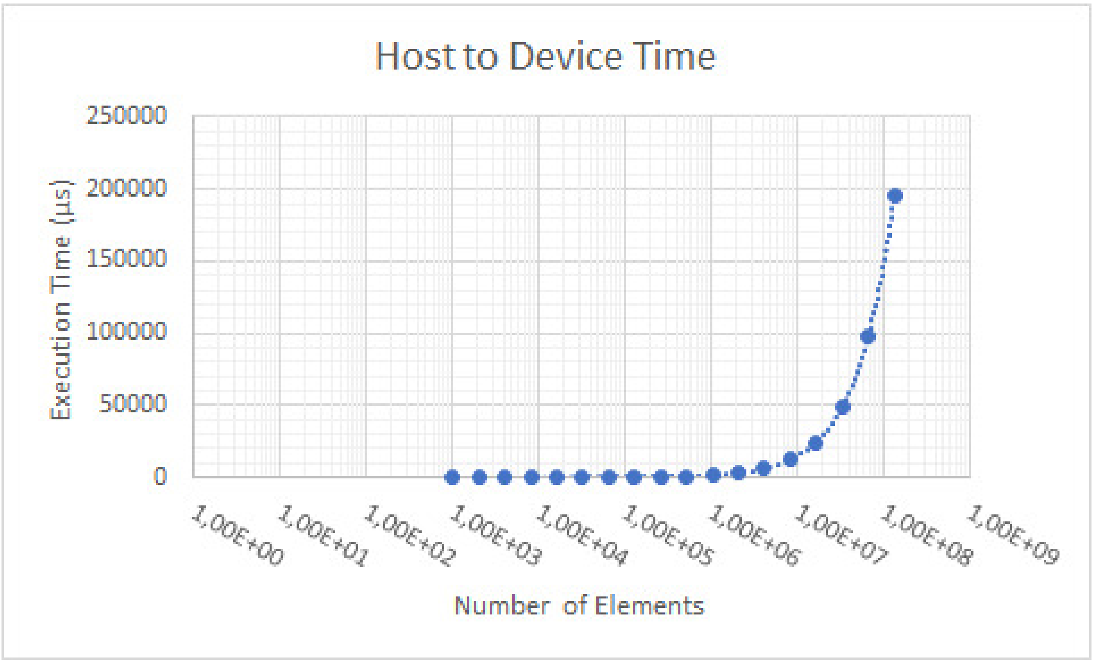

Figure 3: Host to device time

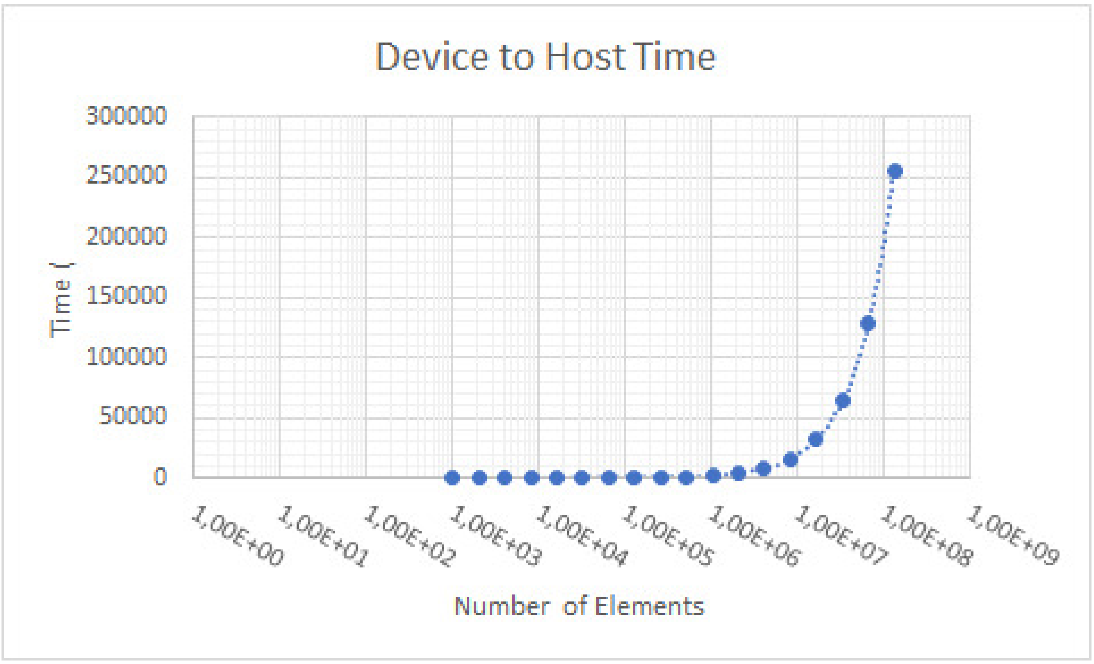

Figure 4: Device to host time

1.2 VectorAdd behaviour with different numbers of threads per block

This graphic shows that the number of threads per block does not affect too much in this concrete problem (with the exception of very small number of thread, this is from one to five). In fact, we can see that the execution time is almost constant.

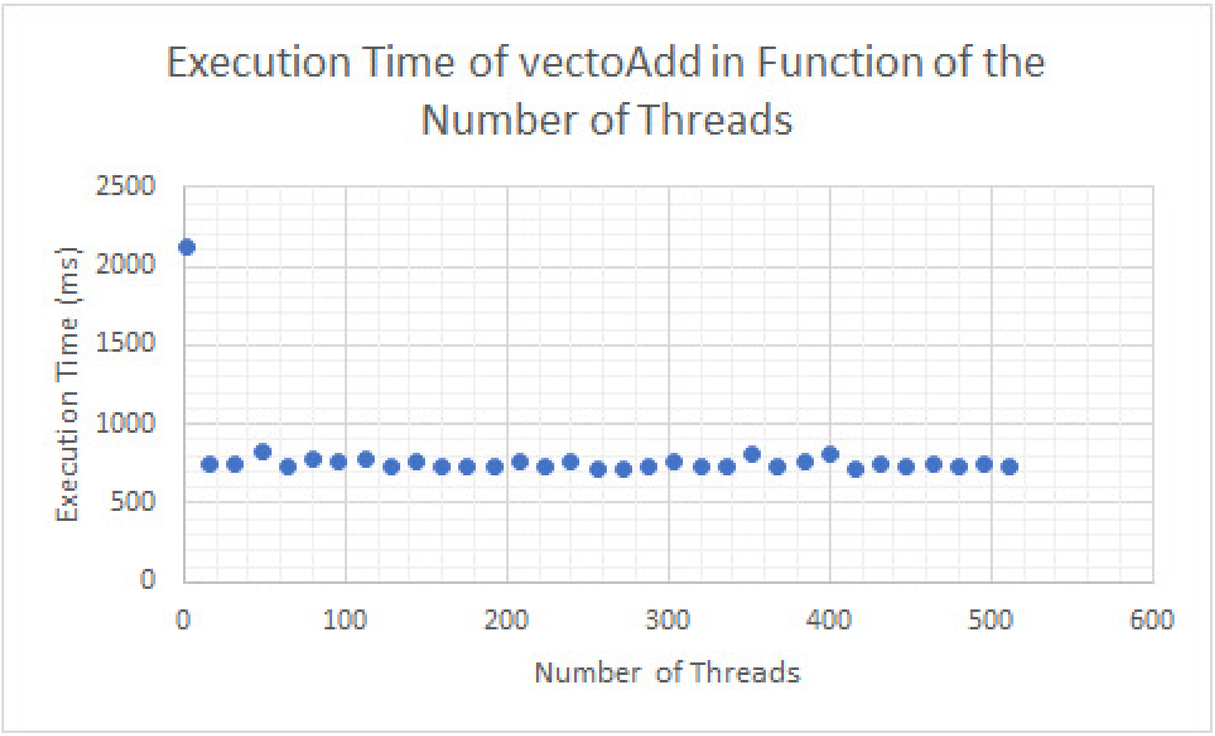

Figure 5: Total execution time vs number of threads per block

2 Task 2

2.1 Use of Memory Managed utility

One important task to solve for every CUDA application we build is the memory allocation. We can choose between doing it manually or we can let the Unified Memory system to manage the allocation, by using the function *cudaMallocManaged (void** devPtr, size_ t size, unsigned int flags = cudaMemAttachGlobal)* . The election of either method can be decided by the following aspects:

- If we are interested in quickly getting a prototype of our application and we are not so concerned with an optimal performance, maybe letting the Unified Memory system to automatically manage the allocation should be our option.
- Using *cudaMallocManaged()* is easier than doing the allocation manually, especially if we need to work with complex structures. In these cases, using the Memory Managed utility is encouraged.
- If we are going to use more than one GPU and we want to use all of them as a single one (sharing resources), using the Unified Memory System is easy and useful.
- On the other hand, in case that we are using all the structures of our program optimally and we really care about performance, manual allocation should be our option. This is it in optimal conditions, manual allocation may performance slightly better.

2.1.1 Use of cudaMallocManaged

Syntax:

*cudaError_ t cudaMallocManaged (void** devPtr, size_ t size, unsigned int flags = cudaMemAttachGlobal)*

1. Parameters
   (a) DevPtr - Pointer to allocated device memory
   (b) Size - Requested allocation size in bytes
   (c) Flags - Must be either cudaMemAttachGlobal or cudaMemAttachHost (defaults to cudaMemAttachGlobal)
2. Returns
   (a) cudaSuccess
   (b) cudaErrorMemoryAllocation
   (c) cudaErrorNotSupported
   (d) cudaErrorInvalidValue

2.2 How to protect against too large data structure to be copied to the GPU

We can use the following piece of code to avoid trying to copy too large data structures to the GPU.

Basically, this code querys the GPU to get the number of devices we have available. If the query is successful, then we calculate the total amount of available memory (sumatory of each GPU memory) in the line *totalMem += prop.totalGlobalMem;*.

Finally we compare the total available memory with the size of the data that we want to allocate (size of data is defined in the first line, by multiplying the number of elements *numElements* times the type of the data). If we have space enough to do the allocation, the program continue. Otherwise, the program will notify that the available memory is not enough and the execution will be stopped.

```
size_t size = numElements * sizeof(float);

cudaDeviceProp prop;
int numDevices = 0;

err = cudaGetDeviceCount(&numDevices);

if (err != cudaSuccess) {
    fprintf(stderr, "Failed␣to␣query␣the␣number␣of␣devices!\n");
    exit(EXIT_FAILURE);
}

int totalMem = 0;

for (int i = 0; i < numDevices; i++) {
    err = cudaGetDeviceProperties(&prop, i);

    if (err != cudaSuccess) {
        fprintf(stderr, "Failed␣to␣query␣the␣device␣properties!\n");
        exit(EXIT_FAILURE);
    }
        totalMem += prop.totalGlobalMem;
}
    if (size > totalMem){
        printf("Memory␣exceeded!\n");
        exit(EXIT_FAILURE);
}
```

2.2.1 Handling large data structures

In order to handle large data structures, we should study the type of problem that we have, because handling these structures is strongly related to the typology of our problem. For example, in the case of the vector add, we can split the main vector into smaller vectors so the allocation of each one of them is possible with our available memory.

However, there are problems with interdependence between data (for example, the case of a matrix multiplication problem), so splitting the main structures into smaller ones could be harder (or even impossible).

Basically, choosing the correct structures and trying to split data (taking into account interdependences between data) are the best techniques to handle large data structures.

3 References

- NVIDIA CUDA Runtime API
- NVIDIA Developer Blog - Mark Harris