A Study in Cuda Usage

Pedro Carrega[1] and Vasco Ferreira[1]

Departamento de Informtica da Faculdade de Cincias da Universidade de Lisboa
{fc49080,fc49070}@alunos.fc.ul.pt

Abstract. This paper was developed with the purpose to explain GPU programing and the improvements that can be made. To demonstrate this we used the Floyd-Warshall algorithm since it is very demanding for the CPU due to the high number of computations required. During our research we discovered that the available materials for learning GPGPU are quite difficult to understand for those who are looking to learn the basics of GPU programing. For that we developed this paper explaining from scratch GPGPU and some of the most significant improvements that can be made. We will start with a simple sequential solution of the algorithm in CUDA and from there go step by step till we use most of the tools and processing power that a GPU has to offer. This paper is helpful for someone who intends to learn the basics of GPGPU or could be used as a teaching guide in an IT course to introduce GPGPU to students.

Keywords: GPGPU · CUDA · Floyd-Warshall.

1 Introduction

The purpose of this paper is to teach GPU programing and the different improvements that can be made, showing the different improvements in each version so that the solution can perform even better taking advantage of the GPU optimizations. To demonstrate this, we used a well known problem that is the Floyd-Warshall algorithm. The algorithm calculates all the possible paths between two points and saves the shortest path. It was chosen because it requires a enormous amount of computation to get the final result since it is has $O(n^3)$ complexity. With a high amount of entries, that is, a big matrix and so it can be very demanding for the CPU. This is where the GPU benefits due to its high number of threads.
For this paper all the examples will be in CUDA.

To make a method to be computed in the GPU you need to declare it with the "__global__" ... so that it will run on the GPU when called in the CPU. Each kernel runs a grid which is the group of blocks that are launched by the kernel, where each block is a set of threads. The grid can be represented as a one dimensional array, a two dimensional array or a three dimensional array. This can be used to improve the mapping of the work that each thread will do.

When launching a kernel it has to be declared the number of threads per block and the number of blocks that are going to be used. It should be taken

into consideration the fact that blocks are composed of warps which are a set of 32 threads, so when declaring the number of blocks being used, it should be a multiple of 32 since all threads in a warp use the same program counter. Therefor if the number of threads in the blocks is not multiple of 32 there will be threads that are busy, waiting for the other threads in their warp that are actually doing some work.

The composition of the kernel grid has a huge impact on performance, with that in mind a programmer of GPGPU should aim to use the largest amount of threads per block possible.

It also needs to be considered that the GPU does not have access to the memory used by the CPU, so all pointers used as input for the GPU kernel will have to be allocated and copied to the GPU global memory before being able to access it. The pointers of the memory that were allocated need to be passed through the parameters of the kernel call so that they can be used by the GPU. In case of a primitive type, it can be passed only by the parameters of the kernel (i.e. int) and does not need to allocate memory on the GPU memory manually(???????). After all the computations are finished, the result should be accessible by the CPU so that it can copy back the result to the CPU and do what it needs with it. To do that it needs to copy the result similarly to the opposite operation but this time from the device to the host. Then the memory that was allocated in the GPU memory should be freed so there are no memory leaks just like in the CPU. In the next figure you can see an example of a input being copied to the GPU, the call of the kernel, the corresponding composition of the grid which will be represented in the (NUMERO DA FIGURA COM A DISPOSICAO) and then the result being copied back to the CPU memory:

Listing 1.1. CUDA Example

```
//Consider the graph pointer a matrix of int with each entry
with its corresponding value filled earlier and the same length on each axis
//graph_size being the size of each axis of the matrix
int* matrix_pointer_device;

cudaMalloc(&matrix_pointer_device, sizeof(int) * graph_size * graph_size);

//This will copy from the graph pointer to the matrix_pointer_device, that is co
cudaMemCpy(matrix_pointer_device, graph, sizeof(int) * graph_size * graph_size,

//Definition of the parameters that describe the kernel grid

//Number of threads per blocks should be multiple of 32
dim3 threads(16,32);
//Number of blocks in the grid
dim3 blocks(4,2);
```

```
//The number of total threads used by the kernel is equal to the total number of
//Launch of the kernel
computes<<<blocks, threads>>>(graph_size, matrix_pointer_device);

//Result being copied back to CPU memory
cudaMemCpy(graph, matrix_pointer_device, sizeof(int) * graph_size * graph_size,

//Freeing the GPU memory
cudaFree(cudaMemcpyDeviceToHost);
```

2 Background

This is an optional section, as it depends on your project. In projects where a given specific knowledge is required to understand the article, you should give a brief introduction of the required concepts. In the case of genetic algorithms, you should present the basic algorithm in this section.

3 Approach

In this section, you should present your approach. Notice that an approach may be different than an implementation. An approach should be generic and ideally applied for different machines, virtual machines or languages. You should present algorithms or generic diagrams explaining the approach.

4 Implementation Details

In this section you can talk about details of how you implemented your approach. Depending on the detail of your approach, this might be an optional section.

As an example, I would detail how I implemented the phaser in the genetic algorithm, or how I implemented parallel mergesort on ForkJoin. Another aspect could be how to organize your arrays to minimize overhead in recursive calls.

5 Versions

5.1 Sequential CPU

In order to demonstrate the effectiveness of the usage of GPGPUs in the processing of some problems we started by implementing our problem in the CPU so we could use the results as a reference point to our others solutions.

This solution will only use one thread of the CPU as the purpose of this paper is to demonstrate the impact of the improvements that can be made on the GPGPU. The graph we will be using for the Floyd-Warshall problem will be a two dimensional graph, in another words, a 2D grid. With that in mind,

we by creating 3 *for* cycles each will start at 0 and will iterate a number of times corresponding to the size of the graph. The second and third cycle will correspond to the X and Y coordinates, respectively, while the first cycle will match an intermediate point for which the shortest path will be computed. Inside the last cycle it will be compared the direct distance of X to Y with the addition of the distances of X to K and K to Y, in case the latter is smaller the distance of X to Y is updated with the value of the addition.

5.2 Sequential GPU

To start our learning of how to program in the GPU we will start by doing the most basic implementation possible, one thread computing the entirety of the Floyd-Warshall problem. GPU functions work like any other CPU functions barring two simple differences.

First, a GPU function has either the *__global__* or the *__device__* ... where *__global__* is a function that will be called by the CPU but run on the GPU and *__device__* is a function that is called by the GPU to be run by the GPU. This ... can also be used to initialize a global variable on the GPU.

Second, the syntax of the call of a GPU function also differs. Kernels work on a grid basis, and such, you need to indicate the grid your program will be working on. That indication is made by placing $<<< X, Y >>>$ after the name of the function, where X represents the number of blocks to be used and Y the number of threads to be used per block, respectively.

Having copied our graph to the GPU memory, we call our kernel with one block and one thread per block, Inside our GPU function we apply the same programing logic that was applied for the sequential CPU version of the Floyd-Warshall problem. We create 3 *for* cycles to iterate through our graph and compute our problem.

5.3 Parallel GPU

Using a single thread to compute a problem on the GPU goes against it's strengths. Due to the low core clock speed, the strength of the GPU comes from it's high thread count being able to compute multiple values at the same time.

When running a function parallel on the GPU each thread will run it's own instance of that same function, in other words, the implementation of the function now needs to take into consideration how we partition the work load for each thread. Having decided the number of threads per blocks we want to use, the most common way to split the work load is to use that number to divide the size of our problem, this way obtaining the number of blocks we are going to use to compute our problem.

Obviously we are going to follow that same logic, but seeing that our problem uses a 2D grid, applying a 2D grid to our blocks and threads. Such can be made using *dim3()* ..., and so our kernel call will look like:
parallelGPU <<< dim3(GRAPH_SIZE/NThreads,GRAPH_SIZE/NThreads),

dim3(NThreads,NThreads) >>>.
Where GRAPH_Size is the size of our graph and NThreads the number of threads we decided to use.

Regarding the implementation of our function, we have to change it completely. The kernel is now called K times by the CPU, where K goes from 0 to GRAPH_SIZE, each thread will now be attributed two coordinates determined by their coordinates in the 2D grid. If the coordinates are withing the scope of the problem they will try to compute the shortest path between X to Y, using the value of K provided by the CPU when the kernel is called, updating the value if needed.

5.4 Work Load

How we partition the work load for each thread may have a big impact on the performance of our solution. Like mentioned before, the major strength of the GPU is it's high thread count so that it can compute multiple values at the same time. During our research into the topic we came up with the following guidelines in order to optimize work load:

1. Every thread works the same amount
2. Use the most amount of threads possible
3. Avoid using the scheduler

1. Considering every thread follows the program counter of the warp, meaning the thread is only allowed to compute a certain line when the warp allows it. Threads with less work load might find themselves in a situation where they must sit idle waiting for others threads to catch up. This scenario is avoided by a correct split of the work load, making every thread in the same warp be synchronized on the computation it needs to execute.

2. Simply put, the more threads you have available to compute your problem the more you can divide your problem. A major cavite however is to not use more threads than those that are available on the GPU. When the number of threads used is bigger than those available the GPU scheduler begins to be involved a major overhead is added to the computation cutting a lot of the performance offered by the GPU.

3. Like mentioned before, using the GPU scheduler adds a major overhead to the computation, so it makes sense that we try to avoid using it. One possible workaround, on a kernel call that uses loops and the problem is bigger than the number of threads offered by the GPU, is to increment the value of the coordinates variable by the product of the dimension of the block by the dimension of the grid. Doing this will allows us to reuse certain threads without having to rely on the GPU scheduler.

To show the importance of using a correct work load we will implement another solution to the Floyd-Warshall problem. First we need to decide how many blocks and threads we will be using, being both in a 2D grid. For the threads you should use the number that makes most sense considering your system, in our case that number is 8. Regarding the number of blocks we decided to use the squared root of the number of blocks allowed per multiprocessor times the number of multiprocessors present on our GPU. We then determined how much positions each thread would need to compute and send that value with our kernel call.

We determine the coordinates like previous implementations with the difference that to that number we multiply the number of positions each thread will need to compute. Then we create two *for* cycles that go from those coordinates to that number plus the number of positions the thread will need to compute. If the coordinates are within the scope of the problem the thread will then compute.

5.5 Synchronization

CUDA offers it's users some forms of synchronization, allowing it's users to have a barrier on theirs implementations. The most common forms of synchronization is using __syncthreads(), which blocks the thread till all threads within the same block call it, and using Atomic variables. Atomic variables have the benefit of offering GPU wide synchronization but usage of such variables is very slow, making them only useful on certain scenarios.

Considering the Floyd-Warshall only has one barrier point, we can use synchronization to reduce the number of kernel calls to just one. We will be using the same implementation used on the previous iteration of our solution, but slightly modifying it so it uses synchronization. We'll start by defining a global variable called *barrier* on the GPU so that we can have GPU wide synchronization, this variable will start at 0. Being a variable whose propose is synchronization, all operations on it will be atomic. But like mentioned before atomic operations carry an heavy performance cost, so to reduce the usage only one thread per block will be computing atomic operations on the variable. Since now we will only be calling the kernel once every thread must now store it's own variable K, which will be iterated by 1 every computing cycle. At the end of the computing cycle one thread per block will increment the value of barrier by one and then be stuck in a empty loop while it waits for every block to increment by 1 the value of barrier. We will be using __syncthreads() so that the other threads await for the synchronization between blocks.

5.6 Shared Memory

Shared memory is a type of memory located on-chip, meaning it is much faster then global memory. In fact shared memory latency is roughly 100x lower than uncached global memory latency (provided that there are no bank conflicts between the threads). It is allocated per thread block, so all threads in the block have access to the same shared memory. The size of memory that can be used is limited, it depends on the GPU used, so when using it be aware of the GPU limitations. Each thread has a local storage which is on-chip as the shared memory, but it is even faster due to being implemented as registers with spill over into L1 cache. This type of memory should be used every time it is possible, as the shared memory it is limited but even more.

Considering that, it should always be taken into advantage the usage of both registry and shared memory. The registry variables are created when defining new variables inside of the kernel implementation, the shared memory are the variables that are declared also inside of the kernel but with __shared__.

The following implementation follows the logic explained in a previous subsection, but now using shared memory to save entries that will be used multiple times, this way we avoid multiple reads from the global memory saving some time in each thread execution resulting in a better performance of the GPU kernel.

MOSTRAR IMPLEMENTACAO DO KERNEL QUE FIZEMOS

The logic in this implementation is that when iterating though a small part the matrix, some positions are being read multiple times, in this specific example lets consider workPerThread with the value 10, in this case each iteration of the outside loop will iterate over the Y axis the 10 times, when doing so the D(x,k) will not change, so by writing this value in the corresponding entry of the matrix in the shared memory we save some time. The logic in writing the D(k,y) is almost the same, when doing the first iteration of the outside loop, we will be reading from global memory all the possible values of D(k,y) for that thread, by doing so it has an unnecessary overhead. To prevent that, we will have a shared three dimensional array, where the first two dimensions represent the thread position in the block, were each thread of the block will have each position and the third dimension will be the different D(k,y) entries for that thread. Note that it should only be done in the first iteration of the outside loop as explained before. In this case it needs to be three dimensional array since it is the inside loop, were in the D(x,k) improvement, we only need one position of the matrix for each thread because that value will be updated at the beginning of each iteration of the outside loop.

6 Evaluation

6.1 Experimental Setup

6.2 Results

To show the results and effectiveness of our solutions we ran two sets of tests:

- We ran both sequential versions with the graph size at 2000, in order to show the weakness of the GPU in single threaded tasks.
- We ran all our implementations, minus the sequential GPU, in order to show the impact of our improvements made on the computation of our problem.

6.3 Discussion

7 Related Work

This section can be either the second one, or the second-to-last. In the case where knowledge of other competing works is required, it could come before. But if you are confident on what you did, it should appear at the end, where you can compare existing works against yours. An example is below:

Chu and Beasley proposed a Genetic Algorithm for the Multidimensional Knapsack Problem [1]. This work introduces a heuristic as a repair operator. Our work makes no optimization with regards to the problem domain, making it more generic and supporting other problems.

When using BibTeX references, I suggest using DBLP[1], leaving Google Scholar as a backup, since DBLP has more correct and detailed information about research papers.

[1] https://dblp.org

8 Conclusions

Here you should resume the major conclusions taken from discussion. Ideally, these should align with the objectives introduced in the introduction.

You should also list the future work, i. e., tasks and challenges that were outside your scope, but are relevant.

Acknowledgements

First Author wrote the part of the program implemented the phasers. Second Author implemented the MergeSort in parallel.

Both authors wrote this paper, with First Author focusing on the introduction, related work and conclusions while the Second Author focused on approach and evaluation.

Each author spent around 30 hours on this project.

References

1. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. J. Heuristics **4**(1), 63–86 (1998). https://doi.org/10.1023/A:1009642405419, https://doi.org/10.1023/A:1009642405419