3rd Project
Advanced Computer Architectures

João Ramiro and Miguel Pinho, Instituto Superior Técnico

Abstract—This project report tries to exploit the Massive Data Level Parallelism of a sample application by using a GPU in order to achieve the maximum speed-up when compared with the CPU only execution. This project will be using NVIDIA's CUDA C to do so.

Index Terms—GPGPU, Massive Data Level Parallelism, GPU Architecture, NVIDIA, CUDA.

1 Introduction

In this project we will analyze Data Level Parallelism (DLP) exploitation for increased performance, in contrast with the previous studied Instruction Level Parallelism (ILP). Whereas in ILP we exploit the fact that multiple instructions can be executed in parallel, when there are no dependencies between them, by mechanisms such as pipelining, reordering and VLIW, in DLP we increase performance by processing different chunks of memory simultaneously. This can be done by applying the same operation over different data or by dividing the data among multiple processing elements.

DLP is particularly useful when there is a high amount of data to be processed and the operations over it are mostly independent between elements, which is called Massive Data Level Parallelism (MDLP). A typical example is graphics, as the generation of each 2D pixel is mostly independent between the others. For this problem specific processing systems were developed and optimized to take benefit of this: Graphics Processing Units (GPUs). However, MDLP is present in a variety of other problems, which may benefit from the usage of GPUs, and so the concept of General-Purpose computing in General Processing Units (GPGPU) emerged.

2 GPU parallelism

Just like described in section 1, the GPU purpose is to exploit MLDP. Therefore, the principal design concern is achieving a very high throughput and the same operation of a lot of data at the same time (a high number of ALUs). As graphics processing has become quite complex all the same, support is also given for executing different operations, different tasks at the same time (task-level parallelism). This is unified by the definition of thread as the base unit processed in a GPU (thread-level parallelism).

In a GPU, we are not so worried with latency, as long has we have more threads to execute concurrently. The previous studied mechanisms, such as out-of-order execution, branch prediction and forwarding mechanisms, are no longer important and we can allow longer pipelines, because we have more threads to execute when hazards cause some to be stalled.

Threads are grouped in a SIMD approach (Same Instruction Multiple Data), where for a group of threads, the same instruction is issued for all, enabling a simpler control and easier IF stage, at the expense of stalling all of theses threads when one has to stall. In modern GPUs this is adapted to a MIMD (Multiple Instruction Multiple Data), where multiple instructions from multiple thread groups are issued at the same time and dynamic scheduling allows different groups to be issued when some are stalled. As well as avoiding stalls, this allows to use the functional units better.

In NVIDIA these static thread groups are called warps and have 32 threads. Their GPUs are constituted by several SMs (streaming multiprocessors) which contain several CUDA cores (ALUs). To assign work to this structure the GPU divides the workload into Blocks, which all do the same operations. This blocks are further divided into warps that operate simultaneously. The GPU used in this lab is a GTX1070 which has a Pascal Architecture. This graphics card has 15 SMs each with 128 CUDA cores. Each block has a maximum number of threads of 1024, and since we will only be assigning one block per SM, each SM will also have a maximum of 1024 threads (although it has actually a maximum of 2048).

2.1 GPGPU

The GPUs are used as accelerators for the CPU, in a slave-master scheme, respectively. The CPU transfer the data and the operations to performed to the GPU and receives its results.

In CUDA programming, the sequence of operations to be applied over a thread are defined in a kernel, with an interface similar to a function. The kernel can be issued by the processor or by the GPU device itself. When launching a kernel, the intra-block dimensions in threads can be defined, in an up to 3D structure, and the same can be done for the grid of blocks to be processed.

There are specific commands for allocating memory in the GPU and for transferring the data.

2.2 Communications

One of the biggest overhead in acceleration with a GPU is that the need of transferring the data to be processed to the

GPU and retrieving the result to the CPU. The CPU also has to communicate with the GPU when issuing commands (kernel calls).

This can be reduced by implementing the algorithm in a way that communications are reduced and data is kept in the GPU between operations. The transfer operations can also be configured as non-blocking and other operations can be performed while the transaction is not completed.

3 CPU Only Implementation

For the purpose of applying the GPGPU concepts in CUDA, a sample program is optimized.

3.1 Parallelism

The initial code provided, uses the Finite-Difference Time-Domain method to get the electric and magnetic fields. The main part of the code consists in pseudo code 1.

```
for n from 0 to N //time iterations

  //ez calculations
  for  j from 0 to JE  // x dimension
    for i from 1 to IE-1   // y dimension
              ez_calculations(j,i)
    end
  end

  //ez field generator
  for  j from 0 to JE  // x dimension
    ez_fied_generator(j)
  end

  //h calculations
  for  j from 0 to JE  // x dimension
    for i from 0 to IE  // y dimension
      hx_calculations(j,i)
      hy_calculations(j,i)
    end
  end
end
```

Listing 1: Pseudo Code of CPU only code

The iterations of the outer loop are not parallelizable since this loop iterates through time and an iteration always depends on the previous one. In the first pair of nested loops where the `ez` is computed, there is no dependency between iterations so every iteration is parallelizable. This is because every 2D point only needs its own value on the previous time iteration and the values of `hx` and `hz` fields. The `ez` field generator loop, can also be parallelizable since it only uses constants and the current time iteration number to compute the `ez` field in a 1D line. Finally the last nested loop calculates `hx` and `hz` using their values on the previous time iterations and the `ez` field previously calculated in this iteration, so this loops have dependencies to the `ez` calculations loops. In conclusion three parallelizable groups were identified and will be later be converted into kernels an run on the GPU.

3.2 Timing Analysis

By analysing the code it can also be understood that the code sections that take longer correspond to the nested loops where the `ez`, `hx` and `hy` are computed which have a $\mathcal{O}(n^2)$ complexity. Even though the code was compiled using the -O3 flags, which allows for loop unrolling and instruction reordering, such optimizations, for somewhat big matrices the CPU takes a long time to compute the matrices. Because of this, the GPU will be used to compute this matrices. The loop where the `ez_field_generator` is computed, will also be calculated on the GPU as avoid unnecessary data transfers between the GPU and CPU.

4 Kernels

As stated on section 3.1 there are massively levels of data parallelism in 3 groups, the `ez calculations` the `ez field generator` and the `h calculations`. The Kernels were created in an iterative process where in the end 3 different working .cu files were created. The optimizations made will now be explained and later their performance will be compared.

4.1 Program A

In this first approach the 3 kernels for each parallellizable area perform different operations about a very similar structure. In each kernel there are as many blocks in the grid X dimension as defined by the JE variable. The pseudo code for the `ez calculations` and `h calculations` kernels are displaying on listing 2

```
j = blockIdx.x;

 //e calculations
 for i
 from threadIdx.x to IE step ThreadsPerBlock
           calculations(i,j)
 end
```

Listing 2: Pseudo Code of Kernel for Program A

For this kernel the blocks process elements along the IE dimension, with one block for each of the JE lines, and the threads in a warp acess coalescent memory.

For the `ez field generator` the pseudo code 3 was implement in all 3 programs. It is important to note that this kernel could have been merged with the first kernel with an extra condition, reducing communication with the CPU, but we were not convinced that this would be advantageous, because it would also increase the complexity of the kernel.

```
for  j
from threadIdx.x to JE step ThreadsPerBlock
      ez_field_generator(j)
end
```

Listing 3: Pseudo Code of Ez ovewrite loop

Another important characteristic is the threads in side each block are grouped and all process elements in the same line. This assignment is important because the vector is stored in memory in the same way (See figure 1), so

by arranging threads this way the data accesses will be coalescent, which means that each block will access memory regions that are close to each other thus improving execution time.

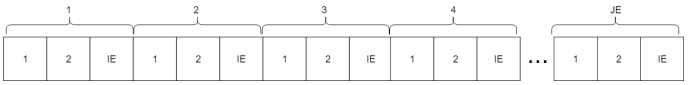

Fig. 1: Mapping from threads to memory accesses

In figure 2 this thread organization is shown, and how the threads iterate through the workload.

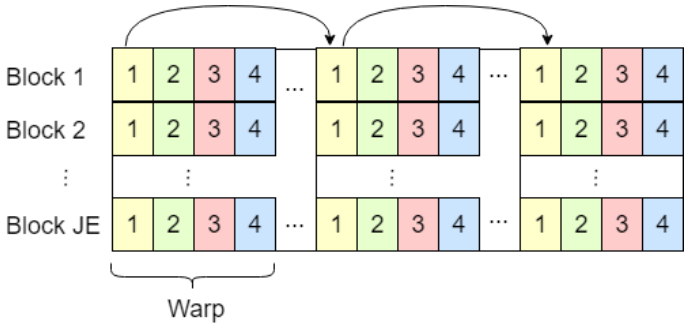

Fig. 2: Program A Thread Iterations

4.2 Program B

To verify that coalescence is in fact an important factor another program was developed, where threads in the same warp are not consecutive.

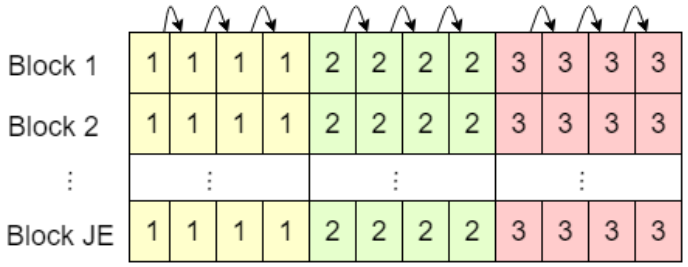

Fig. 3: Program B Thread Iterations

For this program the pseudo code for the `ez calculations` and `h calculations` kernels group threads in a different way (see listing 5).

```
group = ceil(IE/THREADS_PER_BLOCK);
i = threadIdx.x * group
j = blockIdx.x, it;

  for it from 0 to group and i < JE
        i++
          calculations(i,j)
  end
```

Listing 4: Pseudo Code for Kernels of Program B

The blocks continue to be aligned in the same way as program A, but know the threads in each warp are not consecutive, and thus memory accesses are not coalescent.

4.3 Program C

In this program a slight different approach is taken, where each line is divided into several blocks and each thread calculates only one matrix element. The blocks are thus arranged in a 2D grid: one of the dimensions will have the same number of blocks JE, and in the other dimension the blocks will be divided in a way that every block takes care of `#THREADS_PER_BLOCK` as long as there are still elements in that dimension to be computed. So the number of blocks in this dimension is equal to `ceil(IE/#THREADS_PER_BLOCK)`.

This approach makes it so there will be no loops on the kernels and their sequence is simpler.

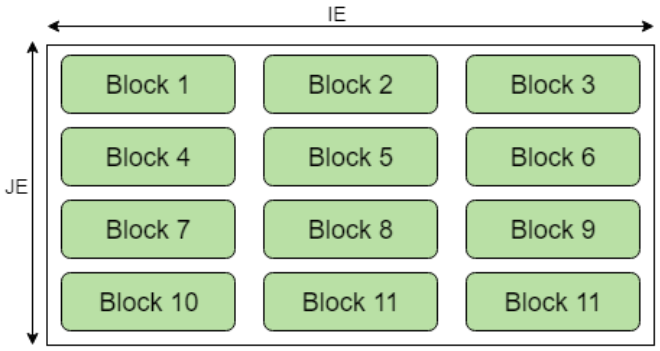

Fig. 4: Block Iterations

For this program the pseudo code for the `ez calculations` and `h calculations` kernels group threads in a different way (see listing 5).

```
i = blockIdx.y * blockDim.x + threadIdx.x;
j = blockIdx.x;

calculations(i,j)
```

Listing 5: Pseudo Code for Kernels of Program C

Each kernel only needs to compute its matrix element based on the grid and block position.

5 Performance

To evaluate performance, the 3 programs were tested and timed. The variables that were tweaked correspond to the matrices size and the number of threads per block (512 or 1024). This numbers were chosen because GPUs perform better when the number of threads per block is a multiple of 32, which is the fixed warp size. These numbers are references that usually perform well. The number of iterations was kept at 500 so that the total execution time is big enough for a good comparison. However this favors the GPU implementation since it hides the initial and final communication overheads. In general high matrix dimensions are used to explore the massive data level parallelism of the GPU.The execution times output of this project are presented in tables 1 and 2

As it can be seen the time for the GPU executions increase as the matrix dimensions grow, in general. There are exceptions, such as when the size increases from 4095

TABLE 1: Execution Times for 1024 Threads per block

Matrix size	CPU Only [s]	ProgA [s]	ProgB [s]	ProgC [s]
1023	2.73	0.21	0.24	0.22
1024	1.39	0.21	0.23	0.21
2048	5.45	0.60	0.66	0.66
4095	23.38	2.06	2.41	2.09
4096	22.37	1.91	2.17	2.10
8192	90.77	6.50	16.80	6.56
16384	349.69	24.89	107.93	25.11
25000	-	66.35	-	67.22

TABLE 2: Execution Times for 512 Threads per block

Matrix size	CPU Only [s]	ProgA [s]	ProgB [s]	ProgC [s]
1023	2.73	0.24	0.25	0.25
1024	1.39	0.22	0.23	0.22
2048	5.45	0.61	0.82	0.63
4095	23.38	2.14	4.48	1.94
4096	22.37	1.91	3.52	1.91
8192	90.77	7.05	26.24	6.35
16384	349.69	25.78	125.06	25.02
25000	-	67.69	-	66.09

to 4096, where the time decreases. This is because the line size 4096 is a multiple of the number of threads per block, so all threads have exactly the same execution (no non-working threads). It can also be understood that while Program B is generally slower, because of the non-coalescent memory accesses, programs A and C have roughly the same execution times, even though they have a different implementation. The similarities in execution times between A and C come from the fact that the accesses to the memory within a warp are still coalescent. Version C requires more scheduling (more blocks), but because the work division is finer-grain, it is easier to keep more SMs working in the final moments of the calculation. Where in program A the whole line of elements would be calculated by a single block and thus by a single SM, program C divides it into blocks that will all compute these lines' elements. Whether this compensates the extra block scheduling overhead seems highly dependent on the problem size.

By comparing execution times between blocks with 512 vs 1024 threads, we conclude that there is no advantage of using one over the other since the results fluctuate.

In the figures 5 and 6 it is shown the speed Ups relative to do CPU only implementation for the 512 and 1024 threads per block respectively.

It is important to note that the previous conclusions are harder to draw in terms of speed-up for the lower sizes, because the CPU version is also highly dependent on the problem size (perhaps due to O3 optimizations and/or memory alignment). For example, the decrease in CPU time from size 1023 to 1024 is so significant it negates the decrease in GPU time.

The speedup in general increases with matrix size for both Program A and C, converging to around 14. This is because with more data the GPU hides better overheads from kernel calls. Program B as expected, performed poorly when the sizes started to increase (from 2048, specifically). This is mainly due to the increase of line size, when each thread is responsible for generating more elements, and only then does the impact of non-coalescent memory becomes visible.

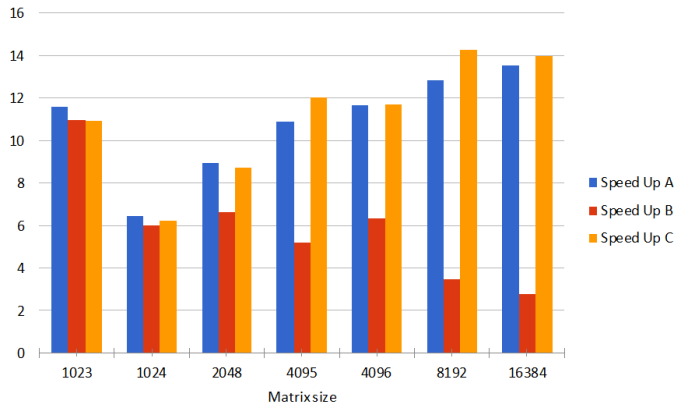

Fig. 5: Speed Ups for 512 Threads per block

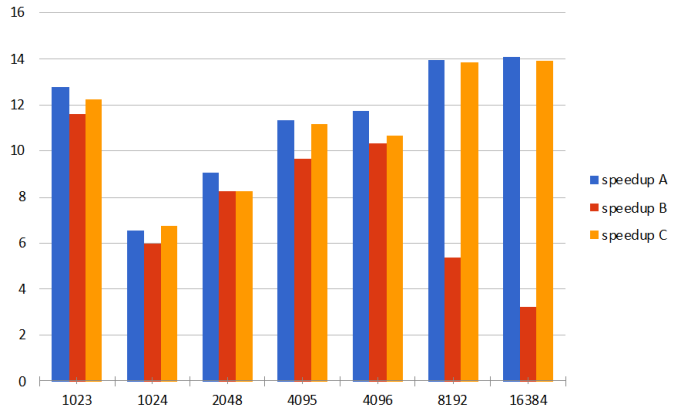

Fig. 6: Speed Ups 1024 Threads per block

6 Conclusion

In this project we learned the basic concepts of GPGPU, and the advantages it has in problems with massive data level parallelism. Whereas there are various tools and functionalities already available to help with this task, it is still important to understand the concepts behind GPU architecture to use them properly.

However, we keep in mind the application optimized is a simpler example, without difficult dependencies, and was not problematic to speed-up in a GPU. There were not any inter-thread dependencies and all the synchronization required was done via the CPU kernel calls. CUDA offers other mechanisms, such as inter-thread synchronization and shared memory for handling more difficult example.

In a real application there would also be more venues for exploiting performance, such as masking use of the time the CPU is idle waiting for the GPU for handling part of the work (using non-blocking communication).