Programming in CUDA for Kepler and Maxwell Architecture

Esteban G. W. Clua [1]
Marcelo Zamith [2]

Data de submissão: 18.08.2015
Data de aceitação: 11.11.2015

Resumo: Desde o lançamento das primeiras versões de CUDA, muitas inovações foram incluídas em GPU Computing. A cada nova versão de CUDA, importantes recursos novos foram sendo acrescentados, tornando esta arquitetura cada vez mais robusta e próxima de um hardware paralelo de uso concencional. Este tutorial apresenta inicialmente a arquitetura de uma GPU e os princípios da linguagem CUDA. Intercalado com esta apresentação, procura-se destacar e contextualizar com os novos paradigmas trazidos pela NVIDIA. Este texto também procura apresentar princípios de otimização para CUDA.

Abstract: Since the first version of CUDA was launch, many improvements were made in GPU computing. Every new CUDA version included important novel features, turning this architecture more and more closely related to a typical parallel High Performance Language. This tutorial will present the GPU architecture and CUDA principles, trying to conceptualize novel features included by NVIDIA, such as dynamics parallelism, unified memory and concurrent kernels. This text also includes some optimization remarks for CUDA programs.

[1] Instituto de Computação, Universidade Federal Fluminense, Niterói - RJ, Brazil
{esteban@ic.uff.br}
[2] Depto. de Ciência da Computação, Universidade Federal Rural do Rio de Janeiro, Nova Iguaçu - RJ, Brazil
{zamith.marcelo@gmail.com}

1 Introduction

While CUDA was released by NVIDIA in June 2007, GPUs were already being used for general computation purposes from many years before, due its high parallel architecture [5]. Due its pipeline based solution, the available technology allowed to include in a single chip dozens of dedicated cores, with a low cost final product. However, its original architecture was completely dedicated to process the typical real time graphics pipeline, requiring that all input and output data should be or become vertices, polygons, pixels, texels, etc.

In the very beginning, graphics on computers were managed by VGA controllers that basically handled the output video memory. In 1997 some manufactures started to include hardware functionalities that were able to implement part of the real time graphics pipeline, such as rasterization, texture mapping and in the most powerful cards even shading calculations. At the end of 2000, this devices were already capable to process most stages of the pipeline, leaving the CPU free of many rendering tasks.

While in that stage GPUs were very "primitive" compared with nowadays architecture, it was on this moment that its DNA was developed. The real time graphic pipeline consists on processing thousands or millions of triangles per frame. However, one triangle processing can be made completely independent of another. This creates an important paradigm of the GPU: its cores do not need to interact among each other for rendering a polygon. Another important characteristic is that sets of polygons share the same process/algorithm for being draw, due its similarities in shading models, textures and materials. This defines a second fundamental paradigm for GPUs: they can run the same algorithm for a large amount of data.

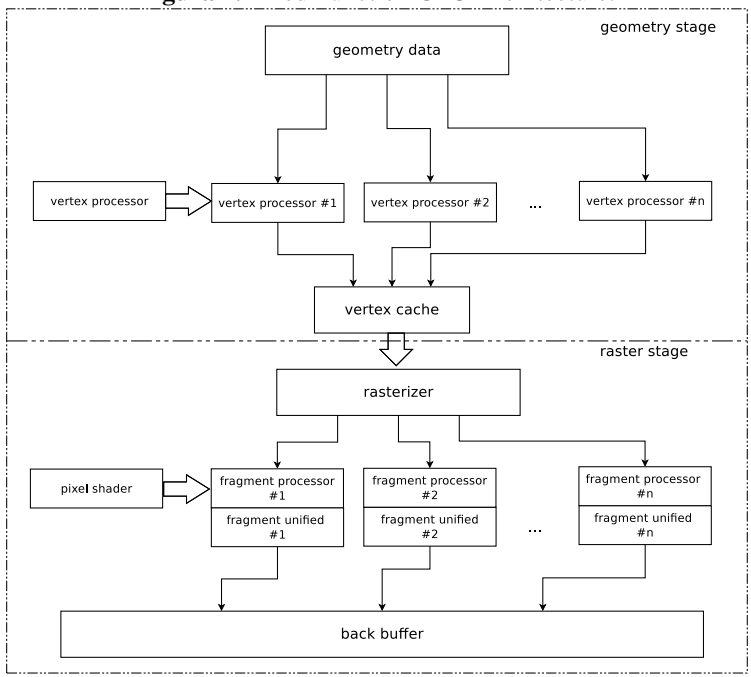
These two paradigms drastically differentiate the GPU with the CPU. CPUs are designed for processing different threads/programs in each processor, requiring a huge amount of transistors for its control flow. Even being able to process many threads at the same time, its thread switch is heavy and lots of cache must be available for a fast context change. In order to be fast, CPUs must deliver data the fastest as possible for its cores. In this sense, they spend a lot of transistors with branch predictors, schedulers and instructions Queues. GPUs will process thousands of threads at the same time, but in a different way, following the described paradigms: all threads correspond to the same function, running on different parts of the data.

Before CUDA, GPUs were designed with two different set of processors: vertex and a pixel processors. The first were dedicated to compute the geometry stage of the real time graphics pipeline, having the vertex as the fundamental data structure. The second were responsible for transforming the geometry data into pixels at the screen, rasterizing the triangles and taking care of possible occlusions among these elements. These GPUs were known as "Fixed function GPUs", since all the functionalities of the graphic pipeline were implemented at the hardware level. At this stage, it was not possible to program GPUs and all

data/communication was made using graphic APIs, such as OpenGL and DirectX.

While this architecture made possible that most of the computer graphic processes of an application were done at the hardware, it straightened the computation possibilities for a limited set of algorithms. Some years later, it was presented the programmable GPUs, which basically allowed to send a program for specifying what should be done at both geometry and raster stage. This solution was still composed of two set of processors: the vertex processor should execute functions known as vertex shaders and the pixel processors (also called framegment processors) should execute the pixel shaders. It is important to stand that the paradigm of the same function being executed for a huge amount of different data was still being respected: the same vertex/pixel shader is applied for a set of polygons/pixels being processed. Whenever a new shader is necessary, a context switch must be executed. An obvious optimization in this paradigm is to join all the geometry that shares the same properties so that less context switch is made. Figure 1 shows this architecture design.

Figura 1. Fixed Function GPU Architecture.



The programmable GPUs at this stage started the concept of GPGPU (General Purpose GPUs), where different algorithms could be executed either by the vertex or the pixel

processor.

In June of 2007 NVIDIA announced a new arrangement of the architecture, where both vertex and pixel processors were joined into a more robust and general processor. This was called Device Unified Architecture, which gives the meaning of CUDA.

CUDA did not change the mentioned paradigms of GPUs: single kernel function being executed in parallel by different processors and small control flow among cores. However, since the time it was launched, many improvements were made and GPU programming capability substantially increased among the years. Today more than one kernel can be executed simultaneously in a single device, GPU and CPU memories can be conceptually treated as a single one, recursions can be called with more freedom and Object Orientation can be used.

The remaining of this tutorial is organized as following: section 2 will present in more details the modern GPUs hardware architectures and its workflow; section 3 will present CUDA in details, showing new features available in Kernel, Maxwell and Paschal. Section 4 will briefly present important optimization concepts, exploring the CUDA profiler features. Finally, section 5 will present the conclusion and trends for the next generation of GPUs.

2 An architecture Overview

Programming in CUDA requires understanding the GPU architecture. The better this knowledge is, the better will be the code and the optimizations. Figure 2 shows the Maxwell architecture.

Basically, the GPU is called a Single Instruction, Multiple Thread (SIMT) architecture. In this model, multiple independent threads execute concurrently using a single instruction. Threads are arranged in groups, called streaming multiprocessors (SM). At each clock, all cores inside the SM execute the same instruction at the same time. The group of threads that are executing the same code at the SM level is called a warp. If more threads than cores are created, more than one warp will be generated and handled by the SM controller, called warp scheduler. Each SM has a dedicated memory, called shared memory. This memory is physically close to the cores and has a very fast access, but is small in size (64KB in Kepler and 128KB in Maxwell). The number of SM can vary according with the GPU generation and model. Many changes in the design of a SM has been made since the first CUDA GPU device, in order to better allocate registers, optimize the warp scheduler and economize energy. Maxwell's SM is called SMM and Kepler's SM is named as SMX.

Threads can be grouped into blocks. Each block is executed at the same SM and for this reason all threads inside it can use the same shared memory. While a block has a limit of number of threads inside it (more recent architecture allow 1024 threads per block), there is almost no limit of number of blocks. The set of blocks created for a CUDA program is called

Figura 2. Maxwell Architecture (figure extracted from [9]).

grid. Note that in optimization processes, it is important to check the occupancy rate, so the distribution of threads per block and blocks per Grid may give different performances.

GPUs are composed by different memories:

- The Global Memory, which is the main memory and can go up to 12Gb in Maxwell architecture. All cores/threads can directly access this memory, but there is a high latency and a low throughput. Data from this memory has the lifetime of the grid.
- The Shared Memory, which is a low latency memory, has a high throughput and is dedicated to each SM. Only threads belonging to the same block can access this memory and the lifetime of it is the same of the block. When a block ends, the shared memory is erased, even for a subsequent block that will use the same SM.
- The Local Memory, so named because its scope is local to the thread, not because of its physical location. Local memory is off-chip making access to it as expensive as access to global memory. This memory is accessible only for a specific thread and its data persists only during the thread execution.
- The Texture Memory, that corresponds to a read-only memory and cached resulting

on one read from the cache. It is optimized for 2D spatial locality, enabling threads that are close to use the same read operation for corresponding data, in case there is coalescence. This memory is also capable of data interpolation, typical operation when solving texture anti-aliasing.

- The Constant Memory, which is also cached and cost one read operation from cache in case it is avoided a cache miss. It is a small memory, with 64KB.

CPU can only read/write data from the global memory and all communication of the host with the device is made through this way.

Threads on CUDA are organized in Warps. Independent of the architecture and the number of cores per SM, each warp is composed by 32 threads that are running together. In this sense, it is recommended, for obtaining maximum performance and obtain the maximum rate of occupancy, to create number of threads per block multiple of 32. Aligning data used by threads of the same warp in a coalescent way is one of the best optimizations possible in a code. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The partition of the block into warps is made by creating and indexing consecutive thread indexes. Since a warp executes common instructions for all its threads, an important optimization is also achieved when all 32 threads have the same path. In this sense, if branches are present and there is a divergence among threads of the same warp, some threads may generate idle time for the corresponding core.

In oldest versions of CUDA, it was only possible to create grids with the same kernel, i.e., it was not possible to execute concurrent kernels. Since Fermi, it is possible to have more than one kernel executed at the same time. However, only Kepler implemented a real concurrent architecture, since in Fermi the Work Distributor joined the code into a single pipeline. Concurrent kernels are scheduled internally by the GPU, but a single SM can execute only the same kernel. In practice, this means that each block can have threads instantiated by the same kernel.

2.1 Kepler Architecture [9]

Kepler introduced a new technology, called Hyper-Q, which enables different CPU programs or threads to launch specific and different kernels at the same device. In practice, this means that different programs or systems can concurrently use the same GPU. Before this, whenever a kernel was launched, the GPU could not receive jobs from other CPU threads or programs, being necessary an explicit synchronization barrier for finishing one grid and starting another.

Another important new feature included by Kepler generation is the possibility of

launching kernels inside other kernels. In past architectures, only the CPU was able to launch a kernel, but kepler SM improvements made possible to call a kernel directly from the device. This feature is called dynamic parallelism and allows more complex kernel management and even recursive kernel calls. For enabling this feature, Kepler introduced a new Grid management unit (GMU), which is able to pause the dispatch of new grids and queue pending and suspending grids until they are ready to execute.

Typically, each new GPU generation has new SM models. Kepler SM is called SMX (Figure 3a), which is composed by 192 cores and 32 SPU (Special Function Unit), responsible for fast approximate transcendental operations as in previous-generation GPUs. Each SMX has 4 warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Due the instruction dispatch units improvements, it is possible to attend warps with 2 independent instructions, reducing some of the possible disparity issues. Kepler's SM improvements also delivers a more efficient energy management, allowing almost 3 times more performance per watt. The SMX architecture included a feature called Shuffle instruction, which allows that threads of the same warp to interchange registers values among them. A full version of Kepler is composed by 15 SMX units, with six 64-bit memory controllers.

2.2 Maxwell Architecture

Maxwell increased still more the performance per watt, improving its rate almost 2x when compared with Kepler.

Maxwell grouped 4 SM at Graphic Processors Clusters (GPC). A full version of Maxwell is composed of 4 GPC, 16 SM and 4 memory controllers (figure 3b). The SM are called SMM, which have 128 CUDA cores each. Each SMM has 4 warp scheduler that are also capable of dispatching two instructions per warp every clock. Each SM has its own dedicated resources for scheduling and instruction buffering, with better improvements in comparison to Kepler, due a number of cores multiple of 32 available, giving better warp alignment by the scheduler. Maxwell has changed the paradigm of dividing shared memory and L1 cache at the same space, dedicating 96KB exclusive space for shared memory (L1 cache was moved to the texture caching area).

Figura 3. Kepler SMX Architecture (a) (figure extracted from [9]) and Maxwell SMM Architecture (b) (figure extracted from [1]).

3 CUDA Programming

GPU Computing is related to heterogeneous computing, since it is always necessary to program the applications with CPU and GPU code. The workflow basically consists on preparing the data at the CPU, sending data from CPU to GPU, call the kernel (GPU function) from the CPU, rearrange the data inside the GPU, work on the data and finally send back to the CPU. The communication of the CPU with GPU must be done through the main memory (CPU) to the global memory (GPU).

While a Maxwell GPU is able to reach around 7TFlops of calculation power, the PCI Express bandwidth is able to deliver up to 224Gb data. This means that it is possible to send from CPU to GPU around 56G floating points per second, which is an amount of 125 lower than the capacity of calculation of floating points. For this reason, trying to optimize and minimize the data traffic from and to the GPU is a must when trying to reach the maximum performance.

The GPU program basically consists on a special function, called Kernel. Code 1 shows a Hello World sample.

Code 1. Hello World program

```
1.
2. //GPU code
3. __global__  void the_kernel(...){
4. }
5.
6. //CPU code
7. int main (void){
8.       the_kernel<<<1, 1>>> ();
9.       printf("Hello␣world!");
10. }
```

The $\_\_global\_\_$ statement indicates that the_kernel function is a kernel and its code will run at the GPU. In this case, nothing is being done and of course there is no sense to print in a parallel way lot's of "Hello Word" Messages. For this reason, the CPU is responsible for printing the message. This code also shows that the first kernel must be launched by the CPU, nevertheless other kernels can also be called inside existing kernels. When calling a kernel, it is necessary to indicate how many threads will be instantiated from it. As presented before:

$$totalThreads = numberOfBlocks * BlockDimension$$

Where $BlockDimension$ corresponds to the number of threads per block. In code

1, the threads are created by line 8. In the syntax $the\_kernel <<< 1,1 >>>$ the first parameter is the number of blocks being created for the Grid generated by the kernel and the second corresponds to the $BlockDimension$, i.e., the number of threads per block. There is a limit for the number of threads per block, which is 1024. Although there is also a limit of number of blocks to be launched, this number is large (64k $\times$ 64k $\times$ 64k), which in theory can be seen as infinite. Nevertheless, creating huge number of blocks maybe not a good granularity strategy.

In CUDA 5 and beyond is possible to launch concurrent kernels, each one with different distribution of number of blocks and threads per block. However, each block of the same kernel will have the same number of threads.

Code 1 is useless, so let's see a more convenient application for the GPU. Code 2 is a simple sum function, also processed at the GPU. Although it still has no parallelism and in practice we will never do something in such a way, this example presents an important stage of GPU programs, which is the data communication of CPU and GPU.

Code 2. Simple GPU program, with most necessary steps for a complete CUDA program

```
1. // GPU code
2. __global__ void sumKernel (int *a, int *b, int *c) {
3.                 *c = *a + *b;
4. }
5.
6. // CPU code
7. int main(void) {
8.       int a, b, c;
9.       int *d_a, *d_b, *d_c;
10.      int size = sizeof(int);
11.
12.      // Alocacao de memoria no device
13.      cudaMalloc((void **)&d_a, size);
14.      cudaMalloc((void **)&d_b, size);
15.      cudaMalloc((void **)&d_c, size);
16.
17.      // Valores iniciais
18.      a = 10; b = 20;
19.
20.      // CPU -> GPU - copia
21.      cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
22.      cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
23.
```

```
24.     // invocacao do kernel com apenas 1 thread
25.     sumKernel<<<1,1>>>(d_a, d_b, d_c);
26.
27.     // GPU -> CPU - copia
28.     cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
29.
30.     // Liberando a memoria alocada
31.     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
32.}
```

In fact, this code shows more or less all the necessary stages for running a solution at the GPU:

a kernel definition (lines 2 to 4);

b GPU variables definition (line 9): Since GPU has a separate space of memory (global memory), it is necessary to create and allocate variables and memory spaces separately from the host;

c GPU memory allocation (line 13 to 15): In the same way that it is required to allocate memory space at CPUs, it is also necessary to allocate the space for the GPU. This is made by using the $cudaMalloc$ function with the required space, in Bytes. The $cudaMalloc$ will care about the referencing the declared variable at stage (b) for the GPU;

d send data from CPU to GPU (lines 21 and 22): The function $cudaMemcpy$ will transfer data from the CPU memory for the device global memory. Notice that the last argument is used for showing the direction od the data transfer ($DeviceToHost$ or $HostToDevice$);

e execution of the kernel (line 25): The kernel is executed creating a total number of threads equivalent to the number of blocks x number of threads per block;

f send data back from GPU to CPU (line 28): Using the same $cudaMemcpy$ function, the data is copied from the device to the host;

g free GPU memory (line 31): The function $cudaFree$ frees the memory space indicated as parameter. In most CPU program languages there is garbage collection, i.e., when the program ends, all the memory is cleaned by the operation system. This do not happen in CUDA, since there is no operational system running inside the device. In this sense, if the developer does not explicitly frees the memory, the allocated region will remain blocked, even when the program is finished and another one starts.

As previously mentioned, GPU computing is considered as hybrid computing and must deal with at least two hardware architectures. In this sense, each hardware have its own memory and whenever it is required to deliver a job from one to another, it is necessary to transfer the data to the memory. Transferring data from CPU to GPU is one possible bottleneck for programs and must be minimized. In the near future, it is intended that the system will have only one single memory, not being necessary this transfer process. While this is not available yet, last versions of CUDA allow to virtually manage the memory as if it were unified. This means that it is no necessary to make explicit the data transfer using the $cudaMemcpy$ function. Code 3 shows an alternative implementation of code 2. Since the kernel is the same, we omit this part of code.

Code 3. Same as code 2, but using CUDA unified memory.

```
1. // CPU code
2. int main(void) {
3.       int a, b, c;
4.       int size = sizeof(int);
5.
6.       // Allocate space for device
7.       cudaMalloc((void **)&a, size);
8.       cudaMalloc((void **)&b, size);
9.       cudaMalloc((void **)&c, size);
10.
11.      sumKernel<<<1,1>>>(d_a, d_b, d_c);
12. // c can be used now by the CPU
13.
14.      //Clean memory
15.      cudaFree(d_a);cudaFree(d_b); cudaFree(d_c);
16.}
```

Notice that in code 3 there are no more duplicated variables for the device. However, it is still necessary to make an explicit $cudaMalloc$, since we must point that a, b and c will be also allocated and instantiated at the GPU. When doing so, CUDA will automatically take care about the copy. While this makes the code much more simple and similar to a regular C code, it is important to notice that there will be a small loss of efficiency.

CUDA kernels are launched in asynchronous way in relation to the host. This means that after launching the kernel, the CPU is free to go with anything that comes in sequence. However, if just after launching the kernel the CPU requires a result intended to be produced by the GPU execution, it is almost sure that it won't be available yet. For this reason, it is necessary to include an explicit synchronization point, which means that the CPU will wait until the GPU finish its job. This is made using the $cudaDeviceSynchronize$ function.

Notice that code 2 does not call this synchronization. This is because $cudaMemcpy$ already makes an internal synchronization. Since in code 3 we are not using it any more, we need to make it manually.

Since Kepler there is also an important improvement at global memory manipulation, which is the inclusion of GPU Direct. These feature allows that global memories of multiple GPUs may be treated as an unified memory. See [6] for more details on its usage.

Finally, let's include some parallelism in our example. Of course the algorithm for adding two regular numbers can't be massively parallelized, so let's now sum 2 vectors of size N. Code 4 show the kernel for this operation.

Code 4. Kernel for adding two vectors .

```
1.__global__ void Vecadd(int *d_a, int *d_b, int *d_c){
2.    int i = threadIdx.x;
3.    d_c[i] = d_a[i] + d_b[i];
4.}
```

This kernel shows an important CUDA reserved word, called $threadIdx$. Each thread of a block will receive an individual index. The index attribution will be made by the warp scheduler and independent of the number of created threads, this is made in constant time. In this sense, let's say that we create N threads. In a parallel way, we will have the following execution:

$$Thread0 : d\_c[0] = d\_a[0] + d\_b[0]$$
$$Thread1 : d\_c[1] = d\_a[1] + d\_b[1]$$
$$\ldots$$
$$Threadn : d\_c[n-1] = d\_a[n-1] + d\_b[n-1]$$

If the number of threads is less than the number of available cores in one block execution, we can say that the sum will be made in one GPU cycle. However, if N is larger, then a pool of threads should be created and there will be a row of thread groups being executed. Each wave of threads executed in parallel is called as warp.

To execute this kernel, one possible way to call from the main program could be as such:

```
VecAdd <<<1,N>>>(d_a, d_b, d_c);
```

As mentioned, this means that we are creating one block, with N threads inside it. While here we already have a parallelism happening, we are still using a very limited amount

of resource, since there is only one block, which means that there is only one SM being used. There is also an important limitation, which is the maximum number of threads per block, which is 1024. That means that if the vector has more than this amount of elements, this strategy is not going to work. One possible, but not ideal, solution for supporting larger numbers for N could be as shown in code 5, where in each thread more than one element of the vector will be calculated:

Code 5. Increasing the granularity of the kernel.

```
1.__global__ void Vecadd(int *d_a, int *d_b, int *d_c){
2.    int i = threadIdx.x;
3.    while( i < N){
4.       d_c[i] = d_a[i] + d_b[i];
5.       i+= blockDim.x;
6.    }
7.}
```

However, this strategy is still not efficient, since all SM except one will be idle. For a complete usage of the GPU it is necessary to create also many blocks, so the following kernel call could be launched:

```
VecAdd <<<K,L>>>(d_a, d_b, d_c);
```

In this case we are creating K blocks, each one with L threads. In this case it is possible to say that the Grid size is $N = K.L$, which is the number of elements of the vector. For doing so, it is necessary to adjust the kernel, so the blocks are correctly used to map the vector elements. Code 6 shows the new kernel.

Code 6. Increasing the granularity of the kernel.

```
1.__global__ void VecAdd(int *d_a, int *d_b, int *d_c) {
2.      int i= threadIdx.x + blockIdx.x * blockDim.x;
3.      d_c[i] = d_a[i] + d_b[i];
4.}
```

The warp scheduler keeps creating L threads per block, so each block will have the same set of $threadIdx$ for its threads. However, the scheduler will attribute different block indices for each block. These indices can be composed with the thread indices for individual vector indices, as shown in code 6.

While this kernel is much more efficient than code 4, there will be a problem in case N is not multiple of any K,L tuple. Suppose that $N = 101$. In this case, one could choose among many possible configurations $K = 5$, $L = 21$. In this case, 105 threads will be created

and for $blockIdx.x = 4$ and $threadIdx.x = 20$ we would have the index $i = 104$, which does not exist in the vectors. In this case, there would be a memory access error, accessing inexistent memory areas. For avoiding this problem, it is very common to select the thread as can be seen in code 7.

Code 7. Avoiding inexistent memory access.

```
1.__global__ void Vecadd(int *d_a, int *d_b, int *d_c) {
2.      int i= threadIdx.x + blockIdx.x * blockDim.x;
3.        if (i < N)
4.            d_c[i] = d_a[i] + d_b[i];
5.}
```

In the listed examples, both thread and block where created in a one-dimensional array. CUDA allows that the arrangement of threads and blocks can be made also in 2 or 3 dimensions. In the previous codes, we used only $threadIdx.x$ and $blockIdx.x$. Although this will not change the performance, some problems can be easily modeled with more dimensions for Grids and Blocks, such as in matrices operations. This can be made using ($threadIdx.x$, $threadIdx.y$, $threadIdx.z$) and ($blockIdx.x$, $blockIdx.y$, $blockIdx.z$) coordinates. Code 8 shows an example of matrix multiplication using two dimensions for both block and grid indexing, one for referring the lines and another for referring the columns.

Code 8. Using more than one dimension for Block and Grid reference. This corresponds to a naïve implementation of matrix multiplication.

```
1.__global__ void add(int *d_a, int *d_b, int *d_c, int K) {
2.      int i = threadIdx.x + blockIdx.x * blockDim.x;
3.      int j = threadIdx.y + blockIdx.y * blockDim.y;
4.      int cValue = 0;
5.      for (int k = 0; k < K; k++)
6.       cValue += d_a[i][k] * d_b[k][j];
7.      d_c[i][j] = cValue;
8.}
```

It is important to mention that all the memory that has been used in all the past examples belong to the global memory. While this is a large space, that can be accessed from all threads and blocks, it is has a big latency and may take a lot of time for accessing.

Shared memories are an important alternative for efficient memory access. While a data in a global memory space can take up to 400 cycles to be accessed, the same data can take 4 cycles if it is located at the shared memory. The main constraint of the shared memory is that it is much more limited in space (Kepler has 64Kb for each block shared memory). There is also an important limitation, which is the fact that the shared memory is only visible

for threads that are in the same block. When a block is finished, all the data of its shared memory is discarded.

Before Kepler architecture, it was only possible to call kernels from the host. The GPU was not able to create threads, calling kernels. CUDA 5 included the feature called dynamic parallelism, which allows the creation of kernels from other kernels.

Another important feature available from Fermi and beyond is the concurrency of kernels. Originally, the concept of a GPU makes that it is only possible to have one same kernel code being instantiated many times and running in different cores, with no thread swap overhead. Newer GPUs architectures enhanced the SM architecture and made them powerful enough for managing individual kernels. This means that different kernels can be called simultaneously. Each kernel will be allocated to a set of SM, which will create blocks of different threads. Code 9 shows a sequence of host code calling two different kernels.

Code 9. Generic example of a concurrent kernel calls.

```
cudaMalloc( &dA, sizeA );
cudaMalloc( &dB, sizeB );
...
cudaMemcpy( dA, A, sizeA, H2D );
cudaMemcpy( dB, B, sizeB, H2D );
...
kernelA <<< gridA, blockA >>> ( ..., dA, ...);
kernelB <<< gridB, blockB >>> ( ..., dB, ...);
...
```

4 CUDA Optimization

Optimization is a set of methods applied to the code, improving its quality and efficiency. In quality is related to code size, requiring lesser memory or in speed which makes the code execute faster.

The aim of CUDA code optimization is to turn the code the most efficiently possible, independently of its size. We may apply optimization inside the code or through compiler parameter. Inside the code means to improve the memory access pattern, reduce or cut out code branches or even write parallel algorithms with least instructions. In case of compiler parameters, we can define sets of instructions of a GPU hardware architecture or even sacrifice the floating point precision in mathematical functions.

4.1 The memory access pattern

The memory access pattern is the way that hardware accesses data on memory for loading or storing data. In case of GPU, the goal of memory access pattern is to maximize the access of data by consecutive threads in a warp. Considering that each SMP fetches 128 bytes for each memory access, a GPU code can maximize the memory access when each thread of one warp requests a 32 bits word. For greater words such as 64 bits, each SMP access twice the memory. In this case, the threads of one warp are waiting for all words of 64 bits requested by threads of warp.

The problem with memory access pattern rises when the kernel works with structures or data used by consecutive threads that are far from one another. In the first case, the memory is fragmented while in the second case the data are stored far from one to another and the spatial locality is not considerate. In both cases, the threads of warp are waiting until all data requested are delivered.

To exemplify the memory fragmentation, let's consider a kernel that applies a simple physics rules to make particles move through the space. Thus, the kernel calculates the new position of each particle based on equation $p_i^{t+1} = p_i^t + a \times v_i$, where p, v and a are vectors representing the space position, velocity and acceleration of a particle. In this case, we don't take into account the interaction among particle for simplicity. Lets consider three approaches to write the kernel.

Approach 1: Let's work with a structure of the structure, named $st\_particle$ (Code 10). Each field of this structure is another structure so-called $float3$. Float3 structure is composed by three floats x, y and z and occupies 12 bytes. $st_particle$ occupies 48 bytes per element in a vector. We consider that one thread computes one particle and a full warp, which are 32 threads at work. Since each particle requires 48 bytes, we need 1,536 bytes to address 32 particles (Figure 4 illustrates 32 particle array in memory). illustrates 32 particle array in memory).

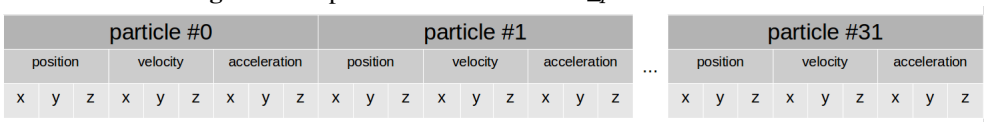
Code 10. Kernel code example for particle movements.

```
1.struct st_particles{
2.    float3 p;
3.    float3 v;
4.    float3 a;
5.};
6.
7.__global__ void K_Particle_01(st_particles *vet){
8.       int i = blockIdx.x * blockDim.x + threadIdx.x;
9.       vet[i].p.x = vet[i].p.x + vet[i].v.x * vet[i].a.x;
10.      vet[i].p.y = vet[i].p.y + vet[i].v.y * vet[i].a.y;
```

```
11.        vet[i].p.z = vet[i].p.z + vet[i].v.z * vet[i].a.z;
12.}
```

The kernel updates the new position particle, computing one component per line as illustrated by code 10, lines 9, 10 and 11.

Figura 4. 32 particles based on the $st\_particle$ structure.



Data of particle #0 begins in position 0 of the memory, the attributes of particle #2 starts in position 96 bytes of memory and so on. To compute the particle #2, a new memory access must be done. In order to finish the execution of line 9, 12 memory access are executed. In the next instruction (line 10), to compute the y component of position, another 12 memory access are executed. This memory access pattern is weak, allowing several threads of warp to be stopped. Despite each memory access fetches 128 bytes, only a small part of data is used by instruction.

Approach 2: To maximize the memory access, we can change the structure to 9 independent arrays, considering three components (x, y and z) for position, velocity and acceleration. This new approach reduces the fetch per warps. Due to all position x attributes of 32 particles are loaded in one access.

Three fetch data are enough to compute x attribute for 32 particles in 32 threads. In this case, the x attribute of the first particle (#0) is in position 0 of memory. The same attribute of particle #1 is in position 4 of memory. Consequently in just one memory access, all threads of a warp is attended in relation to the load of attribute x.

To re-write the code illustrated by code 10 to the code presented in code 11, we included independent attributes arrays.

Code 11. Particle system based on arrays.

```
1.__global__ void K_Particles_02(float *vet_px,float *vet_py,
2.                               float *vet_pz
3.                               float *vet_vx,float *vet_vy,
4.                               float *vet_vz,
5.                               float *vet_ax,float *vet_ay,
6.                               float *vet_az){
7.
```

```
8.      int i = blockIdx.x * blockDim.x + threadIdx.x;
9.      vet_px[i] = vet_px[i] + vet_vx[i] * vet_ax[i];
10.     vet_py[i] = vet_py[i] + vet_vy[i] * vet_ay[i];
11.     vet_pz[i] = vet_pz[i] + vet_vz[i] * vet_az[i];
12}
```

The code $K\_Particle\_02$ (Code 11) presents an optimization memory pattern access due to conclude a warp computation in 9 access while the $K\_Particle\_01$ (Figure 2) requires 4 times more memory access than $K\_Particle\_02$ kernel approach per warp.

As GPUs are SIMT machine architecture, the spatial locality takes into consideration neighbor threads in one warp to deliver the maximum data possible. In doing so, the kernel code must consider this fact to take advantage memory pattern access.

This memory pattern access is the same for all memories in GPUs. From global to cache L2 are transferred 128 bytes by access and the same is applied to cache L2 to L1 [7].

4.2 Shared Memory

As mentioned in section 3, the shared memory has much more latency that the global memory and is an efficient way for threads of same block communicate one to another. While older GPUs models have 16Kb, newer GPUs have 64Kb. Shared memory is volatile with block scope, which means that only threads of same warp can access a data in shared memory. When the block leaves the SMP, all information in shared memory is lost.

The circuit of shared memory is in each SMP and is composed by 32 independent banks for all GPU models. Each bank provides an access of 32 bits word. In other words, neighbor threads in a warp can access independently the shared memory to read and write words. The bottleneck rises when neighbor threads in a warp write words in the same bank. In this case, the access is serialized. For case several, when loading words in the same bank no bottleneck is created.

4.3 Code Optimization through Compilation Directive

The compiler provided by NVIDIA is so-called *nvcc*. The compiler offers some different code optimization directives. We can optimize the code for a specific GPU or even reduce mathematical function precision allowing the code to execute faster. It is also possible to combine all optimization directive in order to create a fastest kernel.

The compiler allows us to work with virtual and real GPU. This approach makes possible to compile a kernel code for all NVIDIA GPU models, based on CUDA capability.

In face of this, let's review the significance of CUDA capability, before explaining how to optimize the code based on virtual and real architecture. CUDA capability gives us two information: the GPU generation and its version. For instance, the CUDA capability 3.5 means GPU generation is 3 (GPU Kepler) and its version is 5, which supports dynamic parallelism. Table 1 shows the relation of CUDA capability, GPUs architecture and their features available.

Tabela 1. CUDA capability and the GPUs architecture/features

CUDA capability	**Features**
2	Fermi architecture
3	Kepler architecture
3.2	Unified memory programming
3.5	Dynamic parallelism
5.0, 5.2 and 5.3	Maxwell

Additionally, The compilation works in two steps: i) nvcc generates an assembler code defined as PTX; ii) CUDA Driver generates the just in time (JIT) binary code, based on PTX code. Indeed, virtual architecture provides a large generic set of instruction represented by the basic architecture, and binary instructions can be encoded based on PTX code, since PTX is always represented in text format.

The best strategy of combination of virtual and real architecture is choosing the virtual one as the lowest architecture as possible and the real as the architectures that the code will be executed. Virtual architecture is given by parameter $-arch$ and its value is $compute\_xy$ where x is the generation and y the version. In the same way, code is stated as $-code = sm\_xy$ where x and y have the same meaning of $-arch$ parameter. An example of compilation syntax is:

```
nvcc -arch=compute_20 -code=sm_20,sm_32,sm_35,sm_50,
sm_52,sm_53 foo.cu -o foo
```

In this sample, we are compiling $foo.cu$ code file where the CUDA kernels are compiled to Fermi as virtual architecture ($-arch = compute\_20$) and the code is loaded JIT to Fermi ($-code = sm\_20$) or Kepler ($-code = sm\_32, sm_35$) or even Maxwell GPU architecture ($-code = sm\_50, sm\_52, sm\_53$). CUDA Driver compiles (JIT) PTX code to binary one, in according to the GPU used. Virtual architecture must be defined lower or equal to the real one. Moreover, we can reduce the scope of virtual and real architecture. For instance, if we know that the only GPUs used are K40s, we could compile the code to the Kepler architecture 3.5 for both architecture: virtual and real. In this case, the compiler parameters are:

```
nvcc -arch=compute_35 -code=sm_35 foo.cu -o foo
```

It is important to remember that the code will fail if we try to execute it in a different architecture than the one used for the compilation (virtual and real). Lastly, the optimized code is given by real architecture defined in the compilation step.

The compiler allows to reduce the number of instruction in several mathematical function, which makes the kernel run faster. This increase in processing time comes from the numerical precision of mathematical functions used in the kernel code. In practice, this means that several mathematical functions, such as sin, cos, pow, log, among others, can execute faster, providing more imprecise results in precision, relying on parameters given to the compiler. Following is an example of compilation:

```
nvcc -use_fast_math  foo.cu -o foo
```

The parameter $-use\_fast\_math$ reduces the numerical precision of mathematical function executed by GPU, but does not affect the CPU mathematical libraries.

4.4 CUDA Profiler

CUDA Profiler is a NVIDIA tools that provides information about the parallel kernel execution and its efficiency, helping to better understand possible bottlenecks. It comes with the CUDA SDK and is installed automatically with $nvcc$ compiler.

It is possible to work with CUDA Profiler in both console or GUI version. Both versions give a view of kernels, memory copy bandwidth from CPU to GPU and vice versa, the kernel instruction, power consumption, GPU thermal behavior, memories bandwidth, functions unit utilization in relation to the functions executed, float point (single and double precision) instructions, among other features [8].

The console version does not present any analysis and just saves kernels performance in a log file to be opened and analyzed by GUI CUDA Profiler. CUDA Profiler GUI version also provides a remote profiling execution based on SSH connection.

CUDA profiler yields two analysis approach: guided and non-guided. Both of them provide the same analysis. However, the guided gives us tips about the possible bottleneck in our application, considering profiled items, such memories bandwidth or executed floating-point instructions. If we are expert in CUDA Profiler, we can execute non-guided approach. Otherwise, we may run guided and read the tips provided.

CUDA Profiler does not require any application changes. However, we can outline a part of our application in order to simplify some additions or modifications in the code. In this case, we can define the activity region.

In case of defining the activity region, the first step is to add the APIs that define the activity region. In this case, we have two APIs in runtime library to outline the application code, which are:

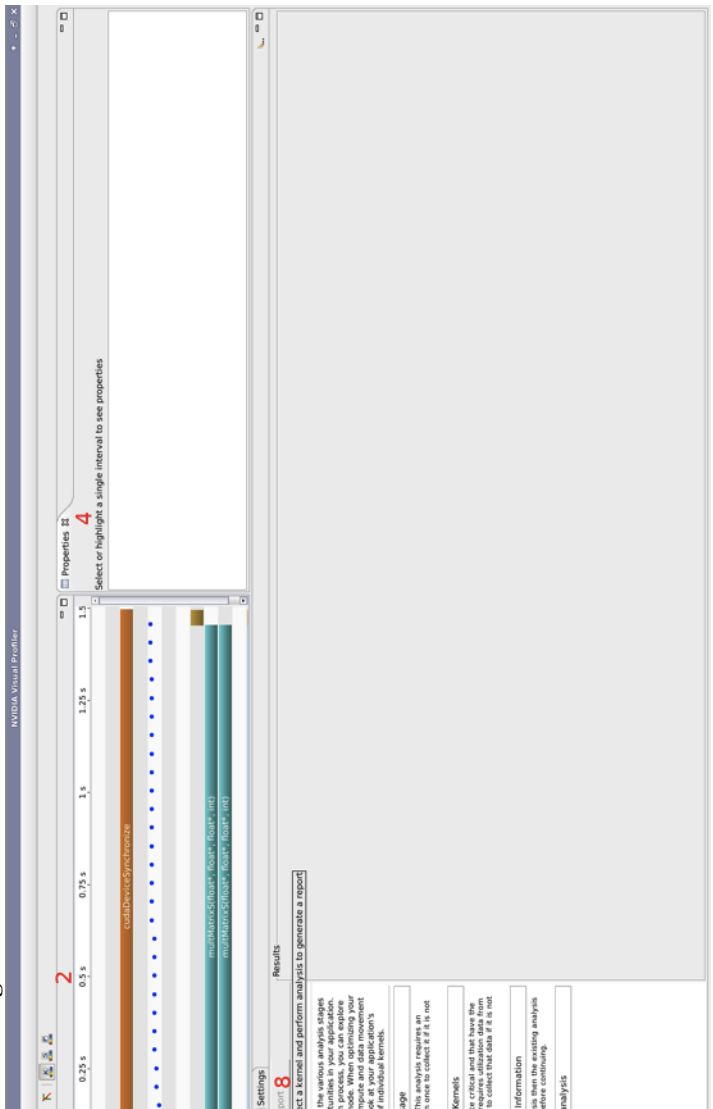
- **cudaProfilerStart()**: outlines the initial part of profile .
- **cudaProfilerStop()**: outlines the final part of profile.

We should include $cuda\_profiler\_api.h$ in our code, otherwise both APIs are not defined. In order that CUDA Profiler can collect information about our application, we must use $-lineinfo$ directive in the compilation step.

CUDA Profiler GUI presents a simple interface. Figure 4 shows the inference, which is basically composed by the following elements:

1. Menu elements.
2. Buttons elements that dispatch the same events of menu elements.
3. Section: it is a profile section and its time line.
4. Properties view: The properties of elements shown in the time line
5. Analysis view: Area that allows to choice the guided or non-guided analysis.
6. Details view: presents information about kernels executions, such as grid and block dimensions registers allocated, shared memory used, achieved occupancy, load a store efficiency, and so on.
7. Console view: shows the output of the application.
8. Setting view: allows to set up some parameters that we want to track.

Figura 5. Interface of CUDA Profiler.



Creating a new section is the first required step after starting running Visual Profiler. For that, it is necessary to press File → New Section (CTRL + N). A new section contains settings, data, and results of the application.

For starting the process, it is necessary to select "Run Guided Analysis" when created a new section. This option makes CUDA Profiler run the application and collect information to profile. If we do not mark it, nothing is executed and we have to execute an analysis by ourselves (guided or non-guided).

CUDA Profiler generates a timeline besides analysis results. We can see in this timeline both CPU and GPU activities, i.e., the CUDA runtime APIs, CUDA driver APIs and kernels. Looking at the timeline view and exploring the profiling information in Properties view is one of the main activities in this tool, making possible to understand how the application is behaving along the time. In the timeline view, we can apply zoom and scroll to focus on specific areas of our application.

In the details view, we can observe specifics metric and events values collected from analysis (guided or non-guided). Metrics and events reveal how the kernels (if we profile more than one) are running in the application.

CUDA Profiler allows to create, save and work with different section simultaneously. This feature is useful since we can re-run an application with different parameters and compare them.

Finally, CUDA Profiler export a report, in PDF file format, with all the analysis.

5 Bibliography

[1] NVIDIA Maxwell specifications. (available at `www.maxwell.nvidia.com`)

[2] Cook, Shane. CUDA Programming. Morgan Kaufmann. 2012.

[3] Kirk, David and Hwu W. Wen-mei, Programming Massively Parallel Processors. Morgan Kaufmann. 2012.

[4] Sanders, Jason, CUDA by Example: An Introduction to General Purpose GPU Programming. Addison-Wesley Professional. In: Springer Science & Business Media, 2010

[5] gpgpu.org (available at `http://gpgpu.org/`)

[6] NVIDIA GPUDIRECT Users Guide (available at `https://developer.nvidia.com/gpudirect`)

[7] NVIDIA Cuda C Programming guide (available at `http://docs.nvidia.com/cuda/cuda-c-programming-guide`)

[8] NVIDIA Profiler Users Guide (available at http://docs.nvidia.com/cuda/profiler-u

[9] NVIDIA's Next Generation CUDATM Compute Architecture: KeplerTM GK110 Whitepaper (available at http://www.kepler.nvidia.com)