



*GPU programming and diffusing
parabolic PDEs for financial
applications (Fourth version).*

Lokman A. ABBAS TURKI

January 2016

Contents

1	Introduction	1
1.1	The course context	1
1.2	From parallel to serial then return back to parallel	2
1.3	Parallel simulation on GPUs	5
1.4	Organization of these lecture notes	9
2	First steps in parallel programming with CUDA/C	11
2.1	Starting C and CUDA programming	11
2.1.1	Preprocessor directives	11
2.1.2	Variable types and manipulations	12
2.1.3	Functions: Declaration and use	12
2.1.4	Displaying	13
2.1.5	CUDA new kind of functions and variables	14
2.1.6	Starting pointers and arrays manipulation	15
2.1.7	Loops and if clauses	15
2.2	Brief description of the GPU hardware/software architecture	17
2.2.1	Some hardware aspects	17
2.2.2	Some software aspects	19
2.3	Examples: Device query, adding vectors and memory speed comparison	19
2.3.1	Empty kernel and device query	19
2.3.2	Adding vectors	21
2.3.3	Memory speed comparison: Registers, shared, constant and global	23
2.4	Exercises	26
2.5	Solutions	30

Chapter 1

Introduction

Most important of all was Fibonacci's introduction of Hindu-Arabic numerals. He not only gave Europe the decimal system, which makes all kinds of calculation far easier than with Roman numerals; he also showed how it could be applied to commercial bookkeeping, to currency conversions and, crucially, to the calculation of interest. Significantly, many of the examples in the *Liber Abaci* are made more vivid by being expressed in terms of commodities like hides, peppers, cheese, oil and spices.

Niall Ferguson "The Ascent of Money"

Chapter contents

1.1	The course context	1
1.2	From parallel to serial then return back to parallel	2
1.3	Parallel simulation on GPUs	5
1.4	Organization of these lecture notes	9

1.1 The course context

The quotation above mentions the fundamental relation in western countries between mathematics and finance. Indeed, although theoretical physics is the first field to which we could think when talking about applied mathematics, history teaches us that finance is the first activity to use applied mathematics. This information has to be conditioned by the fact that mathematics in finance appears much more natural even for novices, in contrast with the use of mathematics in physics that is impossible without mastering some fundamental principles and relations. This argument is maybe the best one that justifies the historic fact and the important number of mathematicians that are continuously interested by this application field. Indeed, for a mathematician, it is easier to have a good intuition for a financial problem than to have it for a gravitational or quantum phenomenon.

These lecture notes illustrate what we just said above. Indeed, this course comes from a growing interest in the parallel multi-core and many-core¹ simulation and the considered applications are taken from mathematical finance. In addition to the theory and the experience, the numerical simulation has become a pillar of scientific knowledge. In this document, the numerical simulation takes an important place. The other goal of these notes is a quick and efficient introduction to parallel programming on GPUs using CUDA/C.

We will use this introduction to raise the historical development of the computing architecture since 1945 and to detail the current directions that these developments take. Throughout this brief presentation, we will try also to understand the progressive changes that occurred on the theoretical advances. Afterwards, we analyze in details the latest evolution of parallel computing throughout the utilization of heterogenous architecture and the use of GPUs. We point out that our objective is to provide a general survey on the evolution of computer architecture as well as their use in simulation, thus, technical aspects will be considered only when needed.

1.2 From parallel to serial then return back to parallel

Unlucky is the human being when he seeks to be replaced to reduce his efforts, because this research is already requiring a lot of it. Fortunately, this quest has not been in vain because it allowed at least to consolidate the detailed knowledge of some tasks, since no one can give a machine a job to do without mastering the different simple parts of it.

Some trace back the evolution of computers from the Chinese abacus (7 centuries BC) but, as announced before, we will focus exclusively on postwar electric machines². Thinking of a task that can be split into elementary parts (ex: memory allocation of a variable, initialization of a variable, adding two variables) that enable non-communicative execution (independent within the computing meaning), the definition of both the parallel machine³ and the serial one becomes natural:

- A serial machine is able to execute the different independent elementary parts only in a serial fashion, ie one after the other.
- A parallel machine is able to execute at least two independent elementary parts in a parallel fashion, ie more than two at the same time.

By this definition, it is very difficult to find now a serial machine although this was not the case, during a long period, after the commercialization of the first transistor computer. As astonishing as it could seem, the first electronic general-purpose machine

¹said differently, parallel and massively parallel.

²World War II.

³This definition is really simplified and does not include the existence of several types of parallelism.

(Turing-complete [BL74]), ENIAC, was parallel [Gol72]. Later, a modified version of it became the first Von Neumann machine [GH93], that refers to a machine in which the program is also stored in the memory. In addition to this revolution, Von Neumann contributed to the development of Monte Carlo method. All this makes ENIAC the first supercomputer to execute during one year the tests of the hydrogen bomb.

Then, there is the period that begins with the Intel commercialization of the first microcomputer in 1971. Between the date of the invention of the first transistor machine (TRADIC 1954) and the seventies, we had always had parallel machines because they were intended to be manipulated by specialists and generally for military applications. Democratizing computers for the general public has raised big hardware and software challenges. The hardware ones were resolved essentially thanks to a better conception of memories and to their hierarchization. Once that we provided sufficient⁴ cache memory⁵ and RAM⁶ to the computing unit, we could execute much easier sufficiently complex tasks to interest the amateurs of new technologies.

Like in other areas, the idea of amortizing the production costs, by selling to the large public, has become inescapable in computer science. Moreover, as the human being reasons sequentially, for his first purchase, he is quite satisfied by a machine that has one core and sufficient memory for a text editor, drawing ... or watch a low resolution video. Thus, the question of increasing the number of cores on one chip was not justified because, each 18 months period, both the number of transistors on each chip and the operating frequency were doubled⁷. Subsequently, one core twice faster could be considered virtually as two cores working at the same time.

As far as the parallelization is concerned, between 1980 and 2000, the scientific user was almost constrained to use connected serial machines. This has not been without pain for the scientific community that was very disadvantaged by the multiplicity of platforms, by an inefficient inter-machine communication and insufficient documentation. Faced to these difficulties, applied mathematics has seen its expansion in the world of serial resolution of PDEs⁸, promoting highly serial iterative methods. A striking standard example of this serial orientation of theoretical research is given by matrices multiplication: In a parallel implementation and with small cache memory for each core, it is simple to separate the multiplications done by each core independently from the others so that the execution time is divided by the total number of cores. However, in a serial PDE-like implementation, first we try to manipulate matrices which are as sparse as possible, then implement a multiplication algorithm that intensely uses the big quantity of the cache available when we use only one core [Tew73].

This orientation in computer architecture and theoretical developments lasted twenty years, until the moment that we reached the limit number of transistors that can be in-

⁴“Providing sufficient” includes the quantity of the memory and the access speed to it.

⁵this memory is a part of the CPU.

⁶Random Access Memory: This memory is separated from the CPU.

⁷Moore’s Law.

⁸Partial Differential Equations.

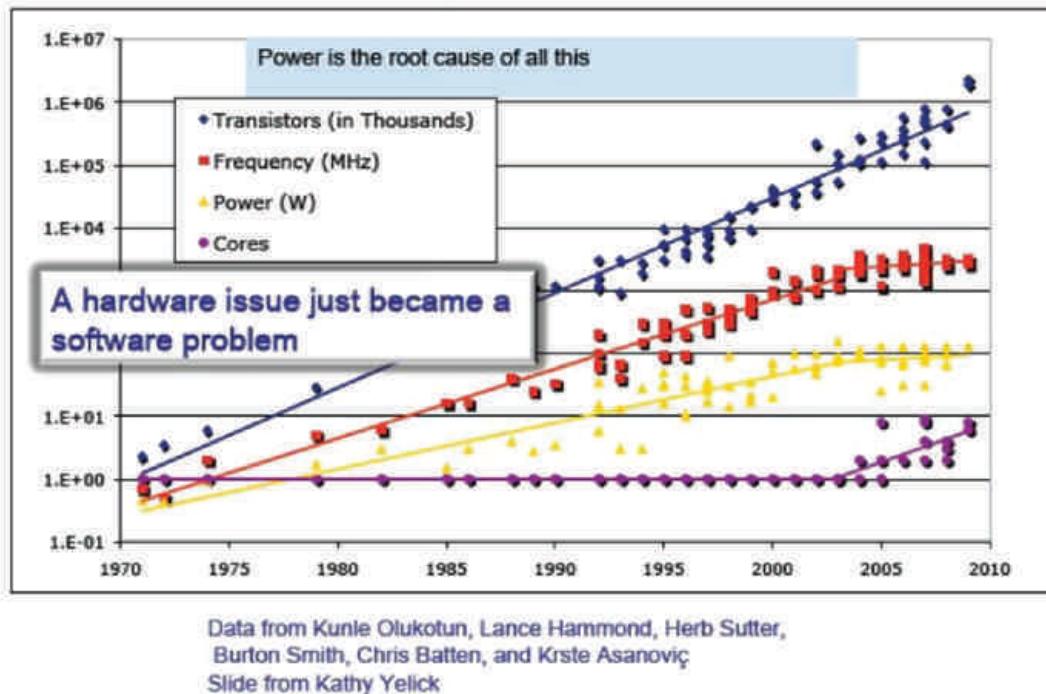


Figure 1.1: Historic evolution on one CPU of: cores number, transistors number, operating frequency and operating power.

cluded in one core. According to Figure 1.1, one can see that from some frequency, the electric operating power has become unsustainable to justify a monocore conception of CPUs⁹. The limit of this architecture is due not only to the fact that the CPU operating power is approximately the cube of its frequency¹⁰ (see [DAS12] for more details), but also to the faced problems to realize thinner printed circuits. To ensure that the Moore's Law remains fulfilled, the direct solution was to increase the number of cores on a single chip. This worked well for two and four cores and has led to the ascent of the number of cores in supercomputers (see Figure 1.2).

Then, the same memory access problem has become, one more time, an important hardware issue. In fact, the cache memory represents about 80% of the CPU and increasing the number of cores reduces considerably the amount of cache available for each core. This is also true for the RAM that is used by multiple cores. Consequently, we should also increase the number of buses¹¹ to access this memory¹². The solution to these problems is based on more localized memory and uses both the code parallelization on different cores and proposing architectures that contain much more than two or four cores. Once more,

⁹Central Processing Unit: The principal computer processor that includes some computing units and cache memory.

¹⁰When the voltage is fixed.

¹¹It is a system (essentially a printed circuit + communication protocol) to transfer data.

¹²In order to keep a good bandwidth for each core.

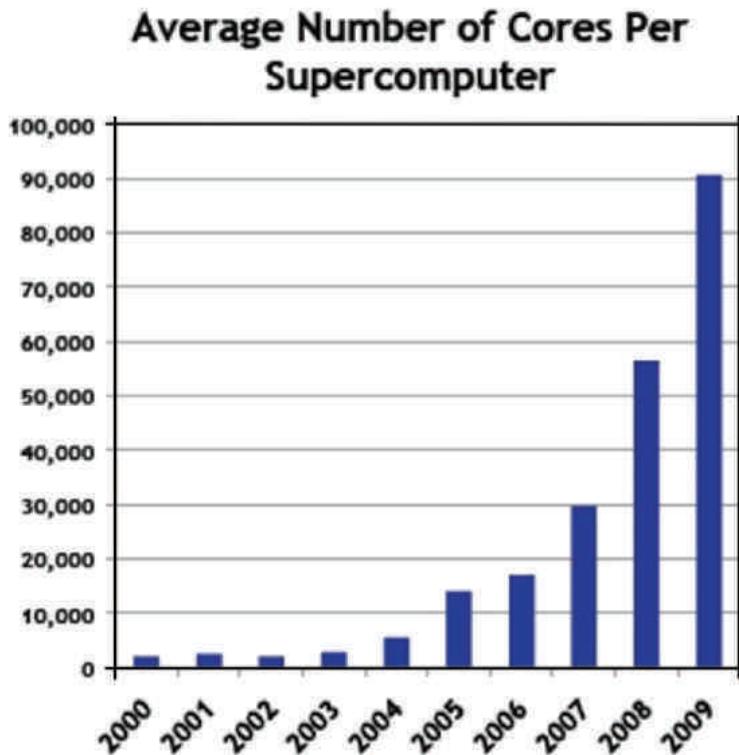


Figure 1.2: Exponential growth of cores number due to a parallelization need.

the commercialization to the large public dictates the rhythm, because it is the GPUs¹³, video game processors, that constitute a serious solution to massively parallel applications. So much true that the introduction of GPUs in supercomputers has completely changed the classification of the most powerful machines in the world (see TOP500 on <http://www.top500.org/>).

1.3 Parallel simulation on GPUs

In Figure 1.3, we illustrate the multi-core architecture of the actual CPU and how this solution faced a memory bandwidth problem. Indeed, according to Sandia National Laboratories, due to a memory access problem with the same bus, increasing the number of cores to more than 12 will produce the same speedup as 2 cores.

As a solution to the memory access bandwidth problem, one can increase the number of buses for a group of cores on the CPU. This solution was, for example, used on the Intel CPUs starting from Nehalem architecture. However, this produced another problem related to the fragmentation of the CPU memory which decreases the memory space for each core. The GPU is the other processor concerned by an increase in the number of

¹³Graphic processing unit: It is a vectorial coprocessor including much more computing units than the CPU, but whose memory organization does not allow to implement efficiently as complex operations as the one that can be performed by the CPU.

cores and the number of buses to access to the memory. According to Figure 1.4, we can see several cores grouped together that access to the memory with the same bus.

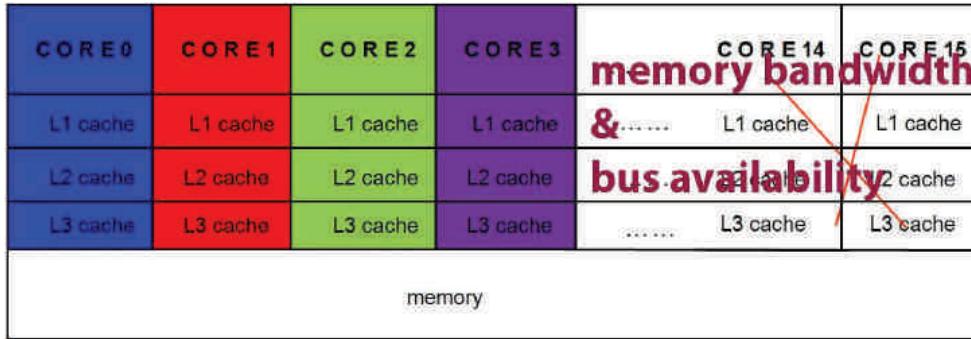


Figure 1.3: Increasing the number of cores in CPU architecture and bandwidth limit.

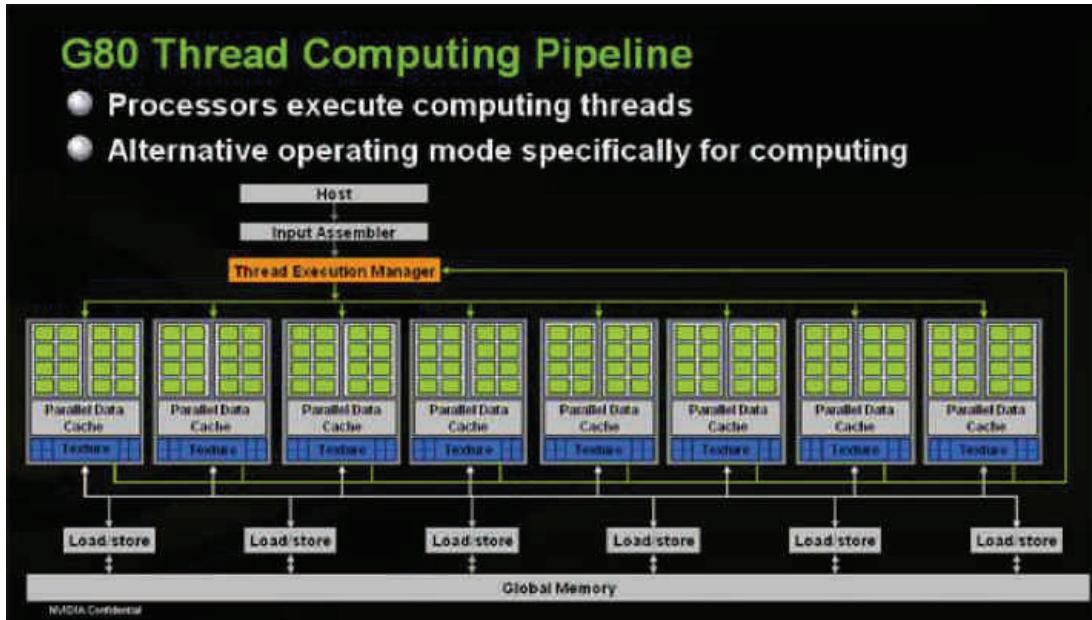


Figure 1.4: GPU architecture: Nvidia Geforce 8800. This architecture is nine years old but provides a sufficiently sophisticated illustration.

Because of the huge number of cores available on the GPU (some GPUs now have more than 3500 cores), one can expect to divide the execution time by a big factor and speedup drastically the different simulations. However, this fact is not completely true. Indeed, according to Amdahl's law **for a fixed problem**, both the execution time $T(P)$ and the speedup $S(P)$ as functions of the number of processors P are given by

$$T(P) = T(1) \left(\alpha + \frac{1-\alpha}{P} \right), \quad S(P) = \frac{T(1)}{T(P)} = \frac{1}{\alpha + \frac{1-\alpha}{P}}, \quad (1.1)$$

where α is the fraction of the algorithm that is purely serial.

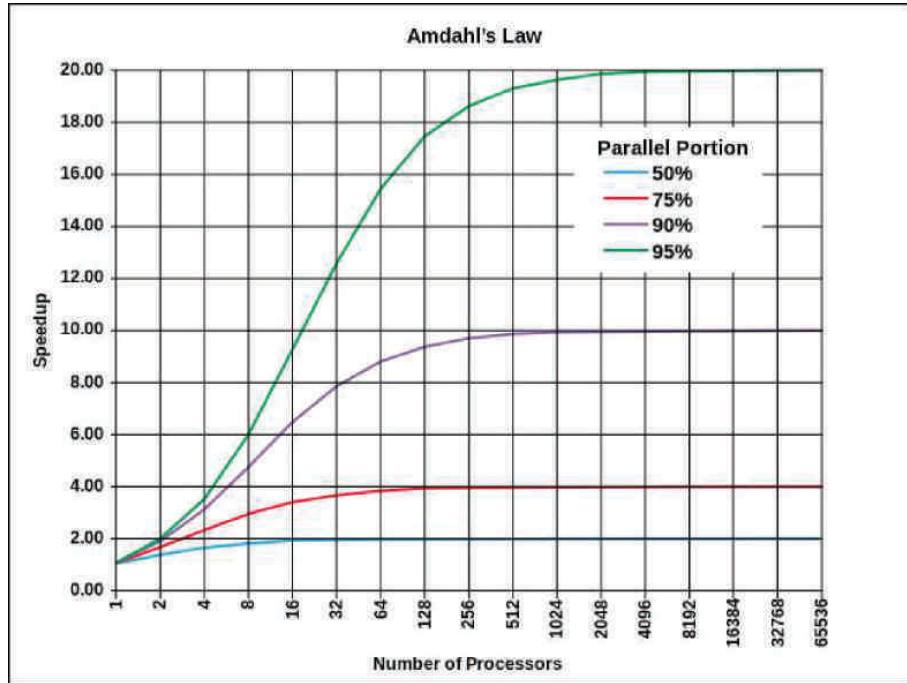


Figure 1.5: Wikipedia illustration for Amdahl's Law.

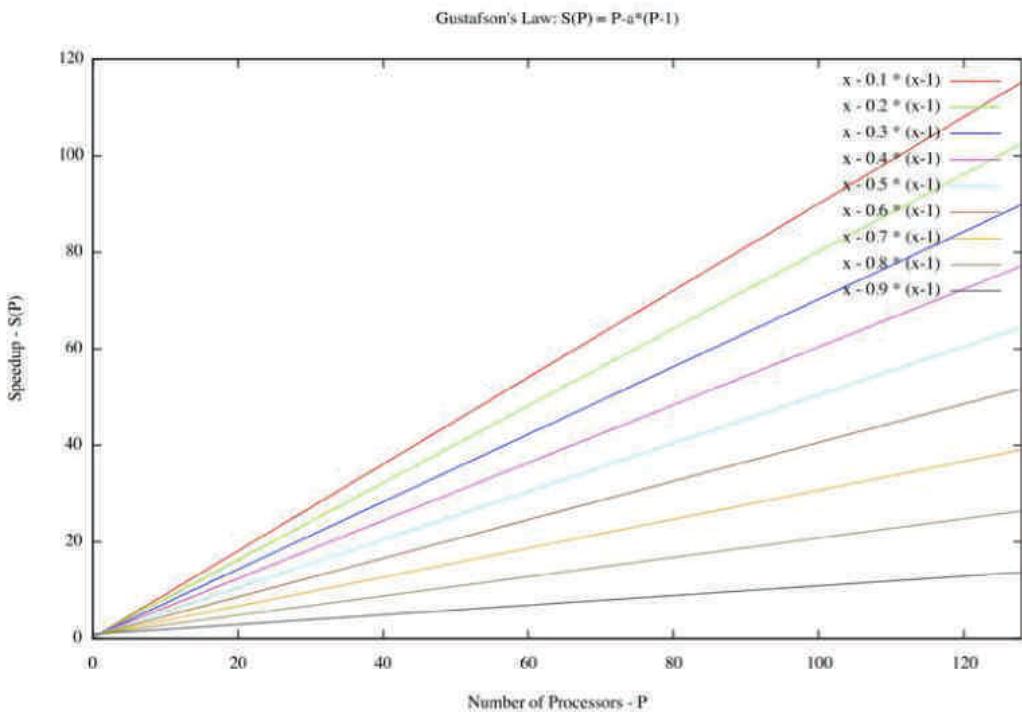


Figure 1.6: Wikipedia illustration for Gustafson's Law.

In Figure 1.5, different speedup curves are illustrated that prove the important limitation that we are confronted to. However, the good news come from the assumption that fixes the problem size. Indeed, when we have a quite big number of cores in our disposal it is, for example, more efficient to increase the accuracy of the original problem making it bigger. This behavior is expressed by the Gustafson's law which is established using $T(P)$ as a reference. Indeed, let us assume that $T(P) = 1U$, where U is a unit of time. Assuming α as before, $T(P)$ can be then decomposed into $T(P) = (\alpha + 1 - \alpha)U$ and thus $T(1) = (\alpha + [1 - \alpha]P)U$. The speedup is

$$S(P) = P - \alpha(P - 1), \quad (1.2)$$

An illustration of (1.2) is given in Figure 1.6.

The emphasis of these two laws is on the size of data which is an important factor in parallel programming. At least, another point that a parallel programmer should take care of is the locality of these data. Said differently, if the problem is so big that the access to slower memory is needed, the speedup will be considerably affected. This latter fact makes some GPU implementations more efficient than the one based on a CPU cluster. The other point that makes the GPU implementation much more advantageous is the energy consumption during the applications. Figure 1.7 shows the evolution of computational capabilities per Watt during the last six years.

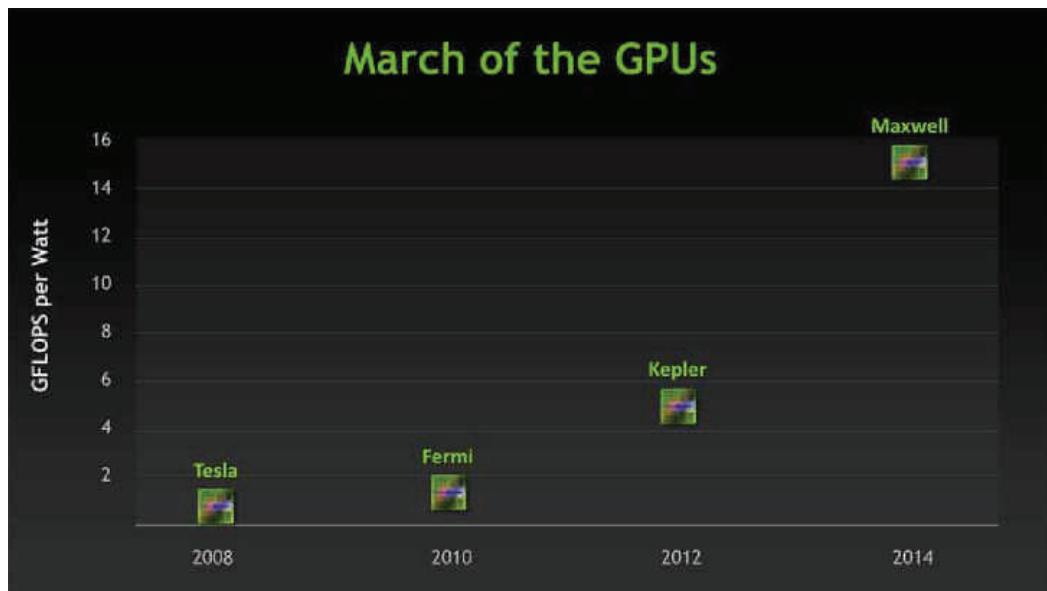


Figure 1.7: Nvidia GPUs computational capabilities

From a software point of view, programming GPUs has become increasingly simple and several solutions can be considered, as the use of:

- OpenCL: A low level language ¹⁴, that is proposed as a language that can be implemented on all cards.

¹⁴The efficiency depends on our good knowledge of the hardware.

- CUDA: Less low level than OpenCL, which was dedicated to Nvidia cards but it started to be implemented on the others.
- OpenACC (came from OpenHMPP): Is a directives¹⁵ language. Its use does not require to rewrite the CPU code, but only to localize the parts that are likely to contain parallelization.

Because this course is dedicated to mathematicians and to those interested by applied mathematics and simulation, we choose to program Nvidia GPUs using CUDA that is easier than OpenCL. However, knowing how to program on CUDA is an advanced starting point to OpenCL programming. Besides, we do not use OpenACC because it is not free and not as efficient as CUDA programming since it hides the GPU architecture details.

1.4 Organization of these lecture notes

Chapter 2

Like almost everything in life, it is important to have strong basis for the manipulation of the easiest problems. In parallel programming, this fact will not ensure to have an efficient program in more complex situations, but it will at least prevent a complete failure.

In Chapter 2, we simplify concepts as much as possible to keep them accessible even for beginners in C programming. One part of this simplification is the naive presentation of the GPU/CUDA hardware/software architecture. The other part is related to the parallel implementation of easy examples that helps a better understanding of GPU programming.

Chapter 3

In this part, we are interested by an advanced comprehension of the most effective ways to reduce the execution time of programs. To reach this goal, we explain the details of the hardware architecture. Then, we see how to propose software solutions that involve both texture memory and shared memory for sufficiently simple problems. We also present the communicative relation that exists between the GPU device and its hosting CPU. This communication could be implemented either asynchronously with the execution of some tasks on the GPU or employed to virtually extend the GPU memory using the CPU memory.

Chapter 4

In order to strengthen what is introduced in Chapter 2 & 3, in this part, we aim at implementing some standard simulations in mathematical finance. These examples are sufficiently general so to be, at least, reused for physical and biological simulations involving diffusions.

Once the hardware/software architecture comprehension of the used GPUs is sufficiently well digested, we begin in this chapter by a summary on the actual and future

¹⁵Like OpenMP.

evolutions of it. Then we re-employ the programming techniques studied before with concrete examples of simulating parabolic linear and nonlinear PDEs. We finish this lecture by seeing the different libraries, code examples, debuggers and forums that help to go beyond the scope of this course.

Chapter 2

First steps in parallel programming with CUDA/C

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

Donald E. Knuth “Structured Programming with goto Statements”

Chapter contents

2.1	Starting C and CUDA programming	11
2.2	Brief description of the GPU hardware/software architecture	17
2.3	Examples: Device query, adding vectors and memory speed comparison	19
2.4	Exercises	26
2.5	Solutions	30

2.1 Starting C and CUDA programming

In this section, we present the basics of C programming and provide some new tools that are specific to CUDA programming.

2.1.1 Preprocessor directives

These are code lines that begin with # and that will be used by the preprocessor before the compilation in order to enrich the code with needed information. For example, writing `#include <stdio.h>` will add to the compilation the file `stdio.h` that contains the prototype of various functions including the display function `printf`. `#include <stdlib.h>` is the other directive that we will usually use since the file `stdlib.h` contains the prototype of `malloc` and `free`.

2.1.2 Variable types and manipulations

In C language, one can define various kind of variables to represent a quantity. Some of the most important ones are:

- **int**: used for integers $\in \{-2^{31}, \dots, 2^{31} - 1\}$ which are then either positive or negative
- **float**: used to represent real numbers in 32 bits (single precision)
- **unsigned int**: used for positive integers $\in \{0, \dots, 2^{32} - 1\}$
- **long int**: used for integers $\in \{-2^{63}, \dots, 2^{63} - 1\}$ which are then either positive or negative
- **unsigned long int**: used for positive integers $\in \{0, \dots, 2^{64} - 1\}$
- **double**: used to represent real numbers in 64 bits with a better precision than **float**
- **char**: used for characters: a, b and so on

For example, if the variable **x** that we want to declare is a signed integer, then it is sufficient to write: **int x;**. Afterwards, **x** can take any integer value for example **x = 1;**. Besides, there are so many ways for a variable to be known by a function, including:

- **global variable**: this variable is known by all the functions defined in the programming file. In this case, we declare the variable at the beginning of this file.
- **local variable**: this variable is known by only one function. In this case, we declare the variable only in the body of one function.
- **shared variable**: this variable is known by more than one function. This can be found in different situation and the most general is when we declare a variable “**x**” in a function that calls other functions with “**x**” as an argument.

2.1.3 Functions: Declaration and use

Functions are the heart of procedural programming. In C language, the declaration of a function should always be done before calling it. For example, the compiler will detect an error if we write:

```
float x;
void f() { int i = 1; x = g(i);}
float g(int j) {return (float)j; //Change the type of the variable }
```

In the previous example, we declare a function **f** using a function **g** which is not yet known. In order to correct it, we should rather write

```
float x;
float g(int j) {return (float)j; //Change the type of the variable }
```

```
void f() { int i = 1; x = g(i); }
```

In this program, `x` is a global variable and `i` a shared variable. From the definition of `g`, we notice that the definition of a function requires at least two information:

- The arguments (or the input) of the function, here `int j`.
- The type (or the output) of the function, here `float`.

Like `f`, we can have functions without input or output. If we do not have an output, we must put `void`. If we do not have an input, we either put `void` between the brackets or nothing.

There are some cases when functions return `void` like:

- When a global variable will contain the result, like in the previous example.
- When manipulating pointers and arrays which will be seen later.
- When the result will not be saved on the CPU memory, like in the case of using CUDA kernels.

From our example, we should point out two other remarks:

- Although we defined the functions `f` and `g` on one line, we can jump lines without causing any problem, **provided that the lines that have to be compiled finish by ";"**.
- In the definition of `g`, we introduced a commentary after `//` that will be ignored by the compiler. Another way of making commentaries is to put them between `/* */`.

One important question remains to be answered:

What is the starting function of a C program?

The starting function of a C program is always called `main`. This means that a C program will start by executing this function before going to any other function called within this `main` function or indirectly called within it through intermediary functions. The convention that we take to declare the `main` function is the following:

```
int main() { ...; ...; ...; ..... return 0; }
```

2.1.4 Displaying

In all our applications, we will use the displaying function `printf` that is declared in the file `stdio.h` which has to be included by the directive `#include <stdio.h>`. Hereafter, we give some examples that are sufficiently explicit to understand how to use this function.

We write the following function in the file `main.c`

```

int main (void){

    // Here we define and allocate values
    int a = 2;
    float A = 2.1f;

    // Here we display some values
    printf("The number %i is bigger than 1\n", a);
    printf("\n"); /* We jump one line */
    printf("it is however smaller than %f\n", A);
    return 0;

}

```

after a compilation with: nvcc main.c -o PRINT and an execution with ./PRINT, we obtain:

The number 2 is bigger than 1

it is however smaller than 2.100000

2.1.5 CUDA new kind of functions and variables

Very often during CUDA/C programming, we use the documentation provided by NVIDIA, in particular the following files:

- **CUDA_C_Programming_Guide**: Necessary document for the CUDA language handling and global understanding of the hardware architecture of the GPU.
- **CUDA_Runtime_API**: Document describing the CUDA functions that allow to program the GPU.

At every session, it is required to open these files and to consult them as often as possible. A good programmer, specialist in converting C code into a parallel CUDA/C code, is the one that masters both the documentation and the use of the "Timer" introduced in Section 2.3.2.

According to the first document, we learn that the kernel definition returns always void and must be preceded by the qualifier `_global_` in the following way:

```
_global_ void myKernel (...) { ...; }
```

The kernel functions will be executed on the GPU but will be almost always called by a CPU function. There is also another category of functions that will be both executed on the GPU and called by a GPU function. The latter category must be preceded by the qualifier `_device_` in the following way:

```
_device_ void myDivF1 (...) { ...; }
```

or also

```
_device_ int myDivF2 (...) { ...; }
```

This means that, in contrast to kernel functions, a device function can return any output including `void` because it will stay on the GPU memory.

CUDA introduces also the `_device_` variables which are sometimes helpful to define global GPU variables, like putting in the beginning of a programming file:

```
_device_ int A[10000]; // There is no size restriction
```

2.1.6 Starting pointers and arrays manipulation

When programming CUDA for parallel applications, it is impossible to not use pointers and arrays. The term pointer comes from the memory address of some variables. To get the memory address of a variable of type `int`, we need a pointer `int*` and to get the memory address of a variable of type `float`, we need a pointer `float*` and so on.

Although we can operate on variables without pointers, using them becomes mandatory when operating on arrays. Indeed, the only way of giving an array to a function, in order to use its content or changing it, is by using pointers. There are two kinds of arrays, the static like:

```
int A[10]; //Small because static, must be known at the compilation
A[0] = 4;
or
double B[15];
B[1] = 2.0;
```

and the dynamic like:

```
unsigned long int length = 1e8 // Length only limited by the memory size
float *C; // Definition
C = (float*)malloc(length*sizeof(float)); // Memory allocation
C[1e8-1] = (float)B[1] + (float)A[0]; // Operations
free(C); // Free it because we do not need it,
// otherwise it stays and occupies a place in the memory
```

In the previous example `A`, `B` and `C` are three pointers to the first place (place 0) of different array types. Indeed, writing `A[0] = 4;` is equivalent to `*A = 4;` and writing `B[1] = 2.0;` is equivalent to `*(B+1) = 2.0;.`

In order to give the array `C` as an argument to a function `f`, it is sufficient to write `f(C)` or `f(&C[0])`, provided that in the definition of `f` we have as argument, for example, `f(float* Ar)`. Finally the operator `&` is the getting address operator.

2.1.7 Loops and if clauses

We have already seen how to access to a special value in an array. When an array is big, it is possible to access to all its coordinates in a synthetic way using an index. Indeed, thanks to loops one can write elegantly the memory access and use it in recursions and iterative algorithms. There are three different syntaxes for writing a loop: The `for` loop, the `while` and the `do while`. In this course, we use only the two first syntaxes. In a C

code once we declare an index *i* and an array *A*:

```
int i;
int length = 10;
double A[10];
```

we can initialize the values of *A* using the **for** loop in the following way

```
for(i=0; i<length; i++) {A[i] = 0.0;}
```

i++ means that we are increasing the value of *i* by one. Thus, going from the first coordinate *i*=0 till the last one *i*=*length*-1, we set the values of *A* to 0.0. An equivalent way of doing that using the **while** loop:

```
i=0;
while(i<length) {A[i] = 0.0; i++;}
```

In addition to loops, the C language allows to write conditions on some operations using **if** and **else**. The **if** clause can be used without an **else**. Let us see the following example:

```
i=length-1;
while(i>-1) {if(i<length/2) {A[i] = 0.0;} else {A[i] = 1.0;} i--;}
```

which initializes to 0.0 or 1.0. *i--* means that we are decreasing the value of *i* by one. There is also the **switch case** clause that will not be used in this course.

Let us show a simple example of the use of loops for the addition of two dynamic arrays *A* and *B* where the results is stored in the dynamic array *C*. Provided that all arrays are linear of size *length*, a simple addition function is given by

```
void addVect(int *a, int *b, int *c, int length) {
    // Variable definition
    int i;

    for(i=0; i<length; i++) {
        c[i] = a[i] + b[i];
    }
}
```

In order to execute this function, we need a **main** function that: Defines the different pointers, allocates arrays, sets a value for each one of them, executes the addition and print it using **printf** function in order to check the result. Finally, It is generally mandatory to free the memory space when it is not used.

```

int main (void){

    // Variables definition
    int *a, *b, *c;
    int i;

    // Length for the size of arrays
    int length = 1000000;

    // Memory allocation of arrays
    a = (int*)malloc(length*sizeof(int));
    b = (int*)malloc(length*sizeof(int));
    c = (int*)malloc(length*sizeof(int));

    // Setting values
    for(i=0; i<length; i++) {
        a[i] = i;
        b[i] = 9*i;
    }

    // Executing the addition
    addVect(a, b, c, length);

    // Displaying the results to check the correctness
    for(i=length-50; i<length-45; i++){
        printf(" (%i) : %i\n", a[i]+b[i], c[i]);
    }

    // Freeing the memory
    free(a);
    free(b);
    free(c);

    return 0;
}

```

2.2 Brief description of the GPU hardware/software architecture

Although we introduced the CPU and the GPU architectures in Chapter 1, we give now a better description of the memory hierarchical organization in the GPU and some specification of CUDA programming

2.2.1 Some hardware aspects

The GPU is a many-core device that contains a very large number of processors that work together. As shown on Figure 2.1, these processors form various groups of Streaming Multiprocessors (SMs). Each SM can execute various independent sequences of instructions called **threads**. To have an efficient implementation, one has to choose a number of threads that is equal to the power of two: 1, 2, 4 and so on. Moreover, to prevent an un-

derutilization of the processors in each SM, one should launch a number of threads that is at least twice the number processors per SM. Otherwise, some processors are likely to do no computation at all or finish fast and wait for the others. However, in some situations, one has to find a tradeoff between the number of threads and the number of **blocks**. Indeed, each SM executes blocks of threads. Thus, if the number of blocks is smaller than the number of SMs, some SMs will not work at all. Like for the number of threads, an optimal number of blocks is at least twice the number of SMs.

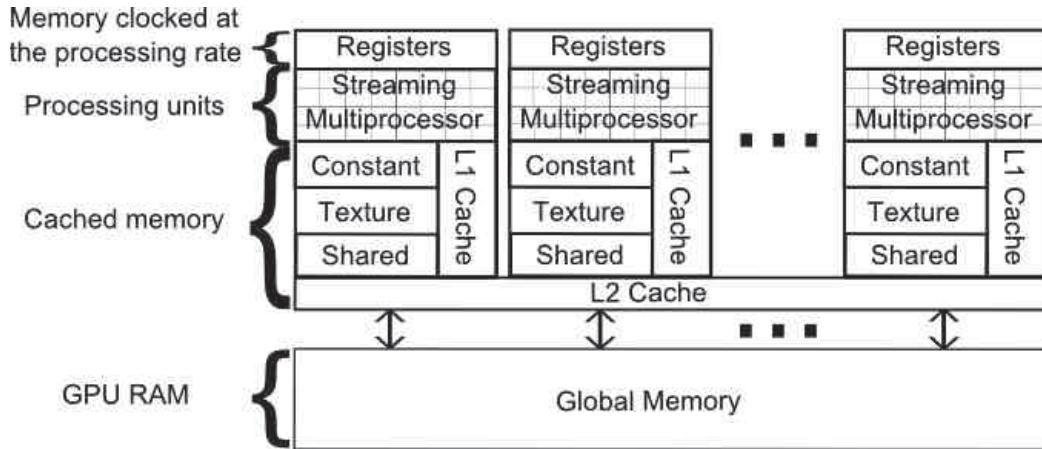


Figure 2.1: Some details on the NVIDIA GPU architecture.

Also according to Figure 2.1, like on the CPU, there are also different memory levels on the GPU. The size of each memory as well as the number of SMs and the number of processors per an SM are specific to each card. Moreover, both the speed of the processor and of the memory vary according to the model.

The register memory is the fastest memory that is used by each SM. Unfortunately, this memory is quite small and it is homogeneously shared by the number of threads per SM. Consequently, for each thread the number of available registers is equal to: (The number of registers per SM)/ (The number of threads per block). Provided that there is sufficient register memory space, one has to use registers for local variables in the program.

The shared memory is the second fastest controllable memory on the GPU. In contrast, the L1/L2 cached memory is a pure uncontrollable with L1 being as fast as the shared memory. The L2 cache memory is only available on GPUs newer than Tesla (Fermi, Kepler, Maxwell). Using the shared memory is justified either because we saturate registers or because some threads need to collaborate. The latter reason, will be studied in Chapter ??.

The other cached memories are: constant memory and texture memory that are both read only memory. The values in the constant memory have to be common to the threads of the same block and should be transferred once to the GPU employing `cudaMemcpyToSymbol`. The constant memory is really fast if all the threads of each block read the same value that is constant. Regarding the texture memory, it is advantageous to use it if we want that each thread accesses to the values of its neighbors. The texture memory access is justified for both graphics and some simulations. Nevertheless, it is not standard and will be studied in Chapter ??.

Finally, there is the global memory which plays the role of a GPU RAM. Although faster than the CPU RAM, it is much more slower than the cache. The size of this memory is of the order of 1Gbytes and the threads of all blocks can access to it in a parallel way explained in the next subsection.

2.2.2 Some software aspects

This part deals with how to lunch and specify a memory access in a kernel function. We present two important points:

1. How to specify the number of threads and the number of blocks.
2. How to use threads and blocks to access to the global, shared and constant memory.

As we saw previously, the GPU hardware includes various SMs that execute blocks. Each processor of an SM executes some threads of some blocks. When ordering the CPU to launch a kernel on a GPU, we need thus to specify the number of blocks and the number of threads per block in the following way:

```
kernel<<<numBlocks, threadsPerBlock>>>();
```

When accessing to the shared memory, we index the threads using `threadIdx.x` which is the index of a thread in each SM. As far as the global memory is concerned, we should consider the fact that this memory is common to all SMs. Consequently, to access to it in parallel we use the following index:

```
idx = threadIdx.x + blockIdx.x*blockDim.x;
```

where `blockIdx.x` is the index of the block and `blockDim.x` is the number of blocks. To use efficiently the constant memory, we should access to a common value to all threads in a block. Subsequently, one can only use `blockIdx.x` for this access.

2.3 Examples: Device query, adding vectors and memory speed comparison

2.3.1 Empty kernel and device query

The simplest function executed on the GPU is given by the following empty kernel that takes `void`:

```
--global__ void empty_k(void){ }
```

Calling a kernel, defined by `void empty_k(void)`, from another function is done by:
`empty_k<<<numBlocks, threadsPerBlock>>>();`

With this example, we can set `numBlocks = 1 = threadsPerBlock` and call this kernel in the `main` function defined in the same file of extension `.cu`. This file is compiled with `nvcc` then executed.

```
#include <stdio.h>

// Function that catches the error
void testCUDA(cudaError_t error, const char *file, int line) {
    if (error != cudaSuccess) {
        printf("There is an error in file %s at line %d\n", file, line);
        exit(EXIT_FAILURE);
    }
}

// Has to be defined in the compilation in order to get the correct value of the
// macros __FILE__ and __LINE__
#define testCUDA(error) (testCUDA(error, __FILE__, __LINE__))

__global__ void empty_k(void){
}

int main (void){
    int count;
    cudaDeviceProp prop;

    empty_k<<<1,1>>>();
    testCUDA(cudaGetDeviceCount(&count));
    printf("The number of devices available is %i GPUs \n", count);
    testCUDA(cudaGetDeviceProperties(&prop, count-1));
    printf("Name: %s\n", prop.name);
    printf("Global memory size in octet (bytes): %ld\n", prop.totalGlobalMem);
    printf("Shared memory size per block: %ld\n", prop.sharedMemPerBlock);
    printf("Number of registers per block: %i\n", prop.regsPerBlock);
    printf("Number of threads in a warp: %i\n", prop.warpSize);
    printf("Maximum number of threads that can be launched per block: %i\n",
           prop.maxThreadsPerBlock);
    printf("Maximum number of threads that can be launched: %i X %i X %i\n",
           prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
    printf("Maximum grid size: %i X %i X %i\n", prop.maxGridSize[0],
           prop.maxGridSize[1], prop.maxGridSize[2]);
    printf("Total constant memory size: %ld\n", prop.totalConstMem);
    printf("Major compute capability: %i\n", prop.major);
    printf("Minor compute capability: %i\n", prop.minor);
    printf("Clock rate: %i\n", prop.clockRate);
    printf("Maximum 1D texture memory: %i\n", prop.maxTexture1D);
    printf("Could we overlap? %i\n", prop.deviceOverlap);
    printf("Number of multiprocessors: %i\n", prop.multiProcessorCount);
    printf("Is there a limit for kernel execution? %i\n",
           prop.kernelExecTimeoutEnabled);
    printf("Is my GPU a chipset? %i\n", prop.integrated);
    printf("Can we map the host memory? %i\n", prop.canMapHostMemory);
    printf("Can we launch concurrent kernels? %i\n", prop.concurrentKernels);
    printf("Do we have ECC memory? %i\n", prop.ECCEnabled);
    return 0;
}
```

Except for the empty kernel, it is quite important to know the specification of the used GPUs before any application. First, one must know the number of available GPUs

thanks to `cudaGetDeviceCount`. Afterwards, we get the specification of each GPU on our machine using the function `cudaGetDeviceProperties`.

Besides, we remark that `cudaGetDeviceCount` and `cudaGetDeviceProperties` as well as all functions prefixed by `cuda` return `cudaError_t`. Thus, we also propose a function that receives `cudaError_t` as an input and catches the error. The final solution is given above.

2.3.2 Adding vectors

It is important to mention that a real comparison, between a CPU and a GPU implementation, has to be based on a parallel implementation on the CPU. However, we are only interested by the parallel implementation on the GPU, but we should keep in mind that a good CPU parallelization speedups the execution time by a factor that can reach the number of the available computing cores (+ hyper-threading).

Let us start with `numBlocks = 1 = threadsPerBlock` and write a kernel that adds two integers sent by the CPU and stores the result in the GPU memory.

```
// Adds two integers sent by the CPU
// and copies the value on the global memory
__global__ void kernel_one(int a, int b, int *c) {
    c[0] = a + b;
}
```

We call this kernel in the `main` function or using a wrapper like

```
// The wrapper that executes kernel_one
int addOne(int a, int b){

    int c, *cGPU;

    // Memory allocation on the GPU
    testCUDA(cudaMalloc(&cGPU, sizeof(int)));

    // Launching the operation on the GPU
    kernel_one<<<1,1>>>(a, b, cGPU);

    // Copying the value from one ProcUnit to the other ProcUnit
    testCUDA(cudaMemcpy(&c, cGPU, sizeof(int), cudaMemcpyDeviceToHost));

    // Freeing the GPU memory
    testCUDA(cudaFree(cGPU));

    return c;
}
```

The function `cudaMalloc` allocates the result variable in the GPU RAM, `cudaMemcpy` copies the addition result to the CPU RAM and `cudaFree` deallocates the GPU RAM. We point out that we re-used the function `testCUDA` introduced previously to catch the errors of type `cudaError_t`.

Let us see how things change when dealing with vectors of length L. One can compute the execution time of the GPU addition using the following wrapper

```

// The wrapper that executes either kernel_vect or kernel_vect_big
void addVect(int *a, int *b, int *c, int L){

    int *aGPU, *bGPU, *cGPU;

    int count;
    cudaDeviceProp prop;
    testCUDA(cudaGetDeviceCount(&count));
    testCUDA(cudaGetDeviceProperties(&prop, count-1));

    float TimerV;                                // GPU timer instructions
    cudaEvent_t start, stop;                      // GPU timer instructions
    testCUDA(cudaEventCreate(&start));           // GPU timer instructions
    testCUDA(cudaEventCreate(&stop));            // GPU timer instructions
    testCUDA(cudaEventRecord(start,0));           // GPU timer instructions

    testCUDA(cudaMalloc(&aGPU, L*sizeof(int)));
    testCUDA(cudaMalloc(&bGPU, L*sizeof(int)));
    testCUDA(cudaMalloc(&cGPU, L*sizeof(int)));
    testCUDA(cudaMemcpy(aGPU, a, L*sizeof(int), cudaMemcpyHostToDevice));
    testCUDA(cudaMemcpy(bGPU, b, L*sizeof(int), cudaMemcpyHostToDevice));

    // Launching the operation on the GPU
    if((L+NTPB-1)/NTPB<prop.maxGridSize[0]){
        printf("No need for a while loop\n");
        kernel_vect<<<(L+NTPB-1)/NTPB,NTPB>>>(aGPU, bGPU, cGPU, L);
    }else{
        printf("A while loop is needed\n");
        kernel_vect_big<<<NTPB,NTPB>>>(aGPU, bGPU, cGPU, L);
    }
    // Copying the value from one ProcUnit to the other ProcUnit
    testCUDA(cudaMemcpy(c, cGPU, L*sizeof(int), cudaMemcpyDeviceToHost));

    testCUDA(cudaEventRecord(stop,0));             // GPU timer instructions
    testCUDA(cudaEventSynchronize(stop));          // GPU timer instructions
    testCUDA(cudaEventElapsedTime(&TimerV,
                                 start, stop));      // GPU timer instructions
    printf("Execution time: %f ms\n", TimerV);    // GPU timer instructions

    // Freeing the GPU memory
    testCUDA(cudaFree(aGPU));
    testCUDA(cudaFree(bGPU));
    testCUDA(cudaFree(cGPU));
    testCUDA(cudaEventDestroy(start));            // GPU timer instructions
    testCUDA(cudaEventDestroy(stop));             // GPU timer instructions
}

```

These GPU timer instructions will be the one used in all the GPU applications studied in this course. Indeed, they are preferred to the CPU timers since they do not take into account the different latencies induced by tasks executed on the CPU.

We already know that `threadsPerBlock` has to be a power of two, for example `threadsPerBlock = NTPB` with

```
#define NTPB 1024
```

set at the beginning of the source file. Subsequently, the maximum number of blocks needed for the addition is given by: $(L + NTPB - 1)/NTPB$. We call then the kernel

function

```
kernel_vect<<<(L+NTPB-1)/NTPB,NTPB>>>(aGPU, bGPU, cGPU, L);
```

that performs the following addition

```
// Adds two integer vectors on the global memory
__global__ void kernel_vect(int *a, int *b, int *c, int L){

    int idx = threadIdx.x + blockIdx.x*blockDim.x;

    if(idx<L){
        c[idx] = a[idx] + b[idx];
    }
}
```

Another solution would be to call

```
kernel_vect_big<<<NB,NTPB>>>(aGPU, bGPU, cGPU, L);
```

with $NB = NTPB$ or any other value smaller than `maxGridSize[0]`. The latter kernel is defined by

```
// Adds two big integer vectors on the global memory
__global__ void kernel_vect_big(int *a, int *b, int *c, int L){

    int idx = threadIdx.x + blockIdx.x*blockDim.x;

    while(idx<L){
        c[idx] = a[idx] + b[idx];
        idx += blockDim.x*gridDim.x;
    }
}
```

2.3.3 Memory speed comparison: Registers, shared, constant and global

To simplify the comparison, we fix the size `length` of vectors to be proportional to the number of blocks `NB` times the number of threads per block `NTPB`. Both `NB` and `NTPB` are known during compilation thanks to `#define` directive. For instance:

```
#define NB 16384
#define NTPB 1024
```

Previously, we saw a dynamic way of declaring variables in the GPU RAM (global memory). We can also use a static declaration like

```
_device_ int aGlob[NB*NTPB];
_device_ int bGlob[NB*NTPB];
```

Now, let us analyse the following kernel

```
__global__ void kernel_vect(int *a, int *b, int *c) {

    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    int i;

    for(i = 0; i<20; i++){
        aGlob[idx] = a[idx] + 1;           // Global variable solution
        bGlob[idx] = b[idx] + 1;           // Global variable solution
        a[idx] = aGlob[idx];              // Global variable solution
        b[idx] = bGlob[idx];              // Global variable solution
    }
    c[idx] = aGlob[idx] + bGlob[idx];    // Global variable solution
}
```

The unique goal of this simple kernel is to perform a large number of global memory access. If we have to replace, as much as possible, the global memory access by the register memory access, `kernel_vect` becomes

```
__global__ void kernel_vect(int *a, int *b, int *c) {

    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    int i;

    int aLocal1, bLocal1, aLocal2, bLocal2; // Local variable solution
    aLocal1 = a[idx] + 1;                  // Local variable solution
    bLocal1 = b[idx] + 1;                  // Local variable solution
    for(i = 0; i<19; i++){
        aLocal2 = aLocal1 + 1;            // Local variable solution
        bLocal2 = bLocal1 + 1;            // Local variable solution
        aLocal1 = aLocal2;                // Local variable solution
        bLocal1 = bLocal2;                // Local variable solution
    }
    c[idx] = aLocal1 + bLocal1;           // Local variable solution
}
```

Finally, if we want to replace, as much as possible, the global memory access by the shared memory access, `kernel_vect` becomes

```
__global__ void kernel_vect(int *a, int *b, int *c) {

    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    int i;

    int idxShared = threadIdx.x;          // Shared solution
    __shared__ int sA1[NTPB], sA2[NTPB]; // Shared solution
    __shared__ int sB1[NTPB], sB2[NTPB]; // Shared solution
    sA1[idxShared] = a[idx] + 1;          // Shared solution
    sB1[idxShared] = b[idx] + 1;          // Shared solution
    for(i = 0; i<19; i++){
        sA2[idxShared] = sA1[idxShared] + 1; // Shared solution
        sB2[idxShared] = sB1[idxShared] + 1; // Shared solution
        sA1[idxShared] = sA2[idxShared];    // Shared solution
        sB1[idxShared] = sB2[idxShared];    // Shared solution
    }
    c[idx] = sA1[idxShared] + sB1[idxShared]; // Shared solution
}
```

In order to compare the three options, we use the following, already usual, wrapper

```

void wrapper_vect(int *a, int *b, int *c){

    int *aGPU, *bGPU, *cGPU;

    float TimerV;                                // GPU timer instructions
    cudaEvent_t start, stop;                      // GPU timer instructions
    testCUDA(cudaEventCreate(&start));           // GPU timer instructions
    testCUDA(cudaEventCreate(&stop));            // GPU timer instructions
    testCUDA(cudaEventRecord(start,0));           // GPU timer instructions

    testCUDA(cudaMalloc(&aGPU, NB*NTPB*sizeof(int)));
    testCUDA(cudaMalloc(&bGPU, NB*NTPB*sizeof(int)));
    testCUDA(cudaMalloc(&cGPU, NB*NTPB*sizeof(int)));
    // Copying the value from one ProcUnit to the other ProcUnit
    testCUDA(cudaMemcpy(aGPU, a, NB*NTPB*sizeof(int), cudaMemcpyHostToDevice));
    testCUDA(cudaMemcpy(bGPU, b, NB*NTPB*sizeof(int), cudaMemcpyHostToDevice));

    // Launching the operation on the GPU
    kernel_vect<<<NB,NTPB>>>(aGPU, bGPU, cGPU);
    // Copying the value from one ProcUnit to the other ProcUnit
    testCUDA(cudaMemcpy(c, cGPU, NB*NTPB*sizeof(int), cudaMemcpyDeviceToHost));

    testCUDA(cudaEventRecord(stop,0));             // GPU timer instructions
    testCUDA(cudaEventSynchronize(stop));          // GPU timer instructions
    testCUDA(cudaEventElapsedTime(&TimerV,
                                 start, stop));      // GPU timer instructions
    printf("Execution time: %f ms\n", TimerV);   // GPU timer instructions

    // Freeing the GPU memory
    testCUDA(cudaFree(aGPU));
    testCUDA(cudaFree(bGPU));
    testCUDA(cudaFree(cGPU));
    testCUDA(cudaEventDestroy(start));            // GPU timer instructions
    testCUDA(cudaEventDestroy(stop));             // GPU timer instructions
}

```

The result depends on the GPU used as well as on the size of vectors. Nevertheless, the register solution always outperforms the shared one which turns to be much faster than the global solution.

Like programming a CPU application, to get an optimal GPU implementation we need a sufficiently good knowledge of the memory architecture. We introduce now the constant memory which is only better than the global memory when the constant value is common to all threads of a block. In this case, only 1 thread among a group of 16 accesses to the constant value and broadcast it to its neighbors. These reduced accesses are quite fast since there are only few of them.

The constant memory array has to be declared global, for instance:

```
__constant__ int vConst[NB];
```

Thus, the addition of a global array with a constant one can be performed by the following kernel:

```
// Kernel that performs vector addition using constant memory
__global__ void kernel_vect_const(int *a, int *c) {

    int idx = threadIdx.x + blockIdx.x*blockDim.x;

    c[idx] = a[idx] + vConst[blockIdx.x];
}
```

and the wrapper becomes

```
// Wrapper that launches kernel_vect_const
void addVectConst(int *a, int *bC, int *c) {

    int *aGPU, *cGPU;

    float TimerV;                                // GPU timer instructions
    cudaEvent_t start, stop;                      // GPU timer instructions
    testCUDA(cudaEventCreate(&start));           // GPU timer instructions
    testCUDA(cudaEventCreate(&stop));            // GPU timer instructions
    testCUDA(cudaEventRecord(start,0));            // GPU timer instructions

    testCUDA(cudaMalloc(&aGPU, NB*NTPB*sizeof(int)));
    testCUDA(cudaMalloc(&cGPU, NB*NTPB*sizeof(int)));
    testCUDA(cudaMemcpy(aGPU, a, NB*NTPB*sizeof(int), cudaMemcpyHostToDevice));
    testCUDA(cudaMemcpyToSymbol(vConst, bC, NB*sizeof(int)));

    // Launching the operation on the GPU
    kernel_vect_const<<<NB,NTPB>>>(aGPU, cGPU);
    // Copying the value from one ProcUnit to the other ProcUnit
    testCUDA(cudaMemcpy(c, cGPU, NB*NTPB*sizeof(int), cudaMemcpyDeviceToHost));

    testCUDA(cudaEventRecord(stop,0));             // GPU timer instructions
    testCUDA(cudaEventSynchronize(stop));          // GPU timer instructions
    testCUDA(cudaEventElapsedTime(&TimerV,
        start, stop));                          // GPU timer instructions
    printf("Execution time: %f ms\n", TimerV);   // GPU timer instructions

    // Freeing the GPU memory
    testCUDA(cudaFree(aGPU));
    testCUDA(cudaFree(cGPU));
    testCUDA(cudaEventDestroy(start));            // GPU timer instructions
    testCUDA(cudaEventDestroy(stop));             // GPU timer instructions
}
```

We urge the reader to check that: The constant memory is much less efficient than the global one if the access to the constant memory is performed in the same way that is done for the global.

2.4 Exercises

Exercise 1

This exercise concerns a straight floating point vector addition. We will use a `timer.h` file that contains

```
#ifndef SEEK_SET
#define SEEK_SET 0
#endif
#ifndef CLOCKS_PER_SEC
#include <unistd.h>
#define CLOCKS_PER_SEC _SC_CLK_TCK
#endif

class Timer {

private:
    double _start;
    double sum;

public:
    Timer(void) : _start(0.0), sum(0.0) {}

public:
    inline void start(void) {
        _start = clock();
    }

    inline void add(void) {
        sum += (clock() - _start)/(double)CLOCKS_PER_SEC;
    }

    inline double getstart(void) const {
        return _start;
    }

    inline double getsum(void) const {
        return sum;
    }

    inline void reset(void) {
        sum = 0.0;
    }
};

};
```

1. What is the maximum number `maxGridSize` of blocks that can be launched?
2. What is the maximum number `maxThreadsDim` of threads and `maxThreadsPerBlock` of threads per block that can be launched?
3. With a vector that has a fixed size, propose a trick to have a number of threads per block always equal to a power of 2.
4. Using `timer.h`, compare a floating vector addition on GPU with the one on CPU.
5. What are the disadvantages of using this timer for the GPU implementations?
6. Using `cudaEvent_t`, evaluate the execution time of the different steps of the floating addition implementation on the GPU.

Exercise 2

The cumulative distribution function of the one-dimensional normal law can be approximated using the function NP defined by

```
/*One-Dimensional Normal Law. Cumulative distribution function. */
float NP(float x)
{
    float p = 0.2316419f;
    float b1 = 0.3193815f;
    float b2 = -0.3565638f;
    float b3 = 1.781478f;
    float b4 = -1.821256f;
    float b5 = 1.330274f;
    float one_over_twopi = 0.3989423f;
    float t;

    if(x >= 0.0f) {
        t = 1.0f / ( 1.0f + p * x );
        return (1.0f - one_over_twopi * expf( -x * x / 2.0f ) * t *
            ( t *( t * ( t * ( t * b5 + b4 ) + b3 ) + b2 ) + b1 ));
    }else{/* x < 0 */
        t = 1.0f / ( 1.0f - p * x );
        return ( one_over_twopi * expf( -x * x / 2.0f ) * t *
            ( t *( t * ( t * ( t * b5 + b4 ) + b3 ) + b2 ) + b1 ));
    }
}
```

The purpose of this exercise is to compare the shared, register, constant and global memories used for the variables p, b1, b2, b3, b4, b5 and one_over_twopi. To perform this comparison, we need to compute the cumulative distribution function of each coordinate of a sufficiently big array of floating values.

1. What is the maximum shared, register, constant and global memory sizes?
2. Find a function in **CUDA_Math_API** that can be used to compute this cumulative distribution.
3. Write a kernel for each solution and compare the execution times.

Exercise 3

Let us consider the following incomplete program

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

__global__ void LDLt_k(float *a, float *y, int n)
{
    // Thread identifier in the grid
    int tidx = threadIdx.x + blockIdx.x*blockDim.x;
    // Shared memory
    extern __shared__ float sA[];

    ////////////////////Fill with the right instructions
}
```

```

int main() {

    int i, j, k;
    // The rank of the matrix
    int Dim = 16;
    // The minimum number of threads per block
    int minTB = 32;
    // The number of blocks
    int NB = 16384;
    // The number of matrices to invert
    int size = NB*minTB;
    // Parameter to fill the matrices
    float rho;
    // The matrix and the value vector
    float *A, *AGPU, *Y, *YGPU;

    float TimerAddOne;                                // GPU timer instructions
    cudaEvent_t start, stop;                          // GPU timer instructions
    cudaEventCreate(&start);                         // GPU timer instructions
    cudaEventCreate(&stop);                          // GPU timer instructions

    // Memory allocation
    A = (float *)calloc(size*Dim*Dim,sizeof(float));
    Y = (float *)calloc(size*Dim,sizeof(float));
    cudaMalloc(&AGPU, size*Dim*Dim*sizeof(float));
    cudaMalloc(&YGPU, size*Dim*sizeof(float));

    // Setting values
    for (i=0; i<size; i++){
        rho = 1.0f/(1.1f+i);
        for (j=0; j<Dim; j++){
            for (k=0; k<Dim; k++){
                if(j==k){A[i*Dim*Dim+j*Dim+k] = 1.0f;
                }else{A[i*Dim*Dim+j*Dim+k] = rho;}
            }
            Y[j+i*Dim]=0.5f*j;
        }
    }

    cudaMemcpy(AGPU,A,size*Dim*Dim*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(YGPU,Y,size*Dim*sizeof(float),cudaMemcpyHostToDevice);
    cudaEventRecord(start,0);                        // GPU timer instructions

    // Resolve the systems AX=Y using LDLt factorization
    LDLt_k<<<NB,minTB,minTB*((Dim*Dim+Dim)/2+Dim)*sizeof(float)>>>(AGPU, YGPU, Dim);

    cudaEventRecord(stop,0);                         // GPU timer instructions
    cudaEventSynchronize(stop);                     // GPU timer instructions
    cudaEventElapsedTime(&TimerAddOne,start, stop); // GPU timer instructions
    cudaMemcpy(Y,YGPU,size*Dim*sizeof(float),cudaMemcpyDeviceToHost);

    // Check a numerical value
    i = 79;
    for (j=0; j<Dim; j++) printf("%f \n",Y[j+i*Dim]);

    printf("GPU Timer: %f ms\n", TimerAddOne);
    // Memory free
    free(A);
    cudaFree(AGPU);
    free(Y);
    cudaFree(YGPU);
    cudaEventDestroy(start);                      // GPU timer instructions
    cudaEventDestroy(stop);                       // GPU timer instructions
    return 0;
}

```