Understanding NVIDIA GPGPU Hardware

Ogier Maitre

Abstract This chapter presents NVIDIA general purpose graphical processing unit (GPGPU) architecture, by detailing both hardware and software concepts. The evolution of GPGPUs from the beginning to the most modern GPGPUs is presented in order to illustrate the trends that motivate the changes that occurred during this evolution. This allows us to anticipate future changes as well as to identify the stable features on which programmers can rely. This chapter starts with a brief history of these chips, then details architectural elements such as the GPGPU core structuration, the memory hierarchy and the hardware scheduling. Software concepts are also presented such as thread organization and correct usage of scheduling.

1 Introduction

General purpose graphical processing units (GPGPUs) were introduced to users a few years ago. They are new, complex and evolving chips that deliver high theoretical computation power at relatively low cost. As the hardware evolves rapidly, programmers using such architectures face various challenges in following this evolution and anticipating future trends. Indeed, due to the approach taken by this hardware, algorithm developments have to be low level and relatively adapted to the underlying hardware.

The architectural model of these chips is different from that of standard processors, yet common details were already implemented into past architectures. But this particular combination is new, and if standard recipes can be used, they have to be

O. Maitre (✉)
Chôros Laboratory
EPFL, BP 2130, Station 16, Lausanne, Switzerland
e-mail: ogier.maitre@epfl.ch

adapted to GPGPUs. Furthermore, modern programmers are used to more standard architectures, such as *x86* processors, with powerful compilers, that present a rather simple view of the underlying hardware.

High-computing-power architectures gained in abstraction level, over the years, and the new usage trend is to rely on an abstract view of the machine and let the software fill the gap to the concrete processor.

As GPGPUs have a complex hardware model, programmers need to understand the underlying hardware in order to use it to achieve interesting speedups. This can be difficult, as this architecture is not yet fixed and modifications are constantly occurring.

1.1 History and Origins

Graphical processing units (GPUs) emerged at the end of the 1990s, with the growth of 3D computing needs. They are fast and parallel chips that implement parts of the classical 3D software 3D rendering pipeline (OpenGL and DirectX). They can apply common rendering operations such as projection and lighting calculations. With the evolving needs of 3D rendering processes, shaders appeared, which allow the programming of certain steps of the rendering pipeline. Indeed, two steps become programmable, i.e. vertex and pixel calculations that allow us to customize the process before and after the rasterization stage. This allows developers to use these devices to apply custom algorithms, for example, for shadow calculation, by implementing custom vertex or pixel shaders.

These new capabilities inspired a number of works [4–6] which diverted the 3D rendering pipeline by inserting calculations that were not graphical, on non-graphical data.

These studies faced two major difficulties. The programming model of the languages used for pixel or vertex shader programming required a difficult translation to 3D rendering paradigms, and as the chips contained an architecture for each kind of shader, with a specific programming model (vertex or pixel related), these works generally used only one, leaving the other half of the processor idle.

At the end of 2006, the manufacturer NVIDIA introduced a new architecture, the G80, which led to major changes in this method. The pixel and vertex shader computing units are unified, and the G80 only has one type of core, which is more generic than before. This modification enhances programmability and eases load balancing between the two types of computing units.

Concurrently with this new architecture, CUDA appeared, a framework designed to allow the GPU to be programmed directly. CUDA is composed of a language (C-like, extended with keywords intended to manage GPU and CPU codes in the same source), a compiler for this language and libraries to handle GPU management.

Still, the chip contains limitations, compared to standard CPUs, such as:

- Lack of call stack management, requiring complete inlining at compile time and preventing recursive functions
- Lack of hardware support for double-precision calculation
- No communication with other devices, which prevents calls to functions like *printf*.

Double precision calculation will be introduced in GT200 chips and call stacks on the Fermi architecture. However, the freedom in programming offered by CUDA allows NVIDIA to claim their architecture as being GPGPU for "General Purpose Graphical Processing Unit". A large amount of scientific work takes advantage of this new architecture and its use spread quickly. The CUDA concept was therefore born, and some of its principles took the shape that we know nowadays.

1.2 Market Considerations

GPU computing is a hot area in scientific computing, as these devices allow us to obtain very interesting speedups for low costs, which compared to more conventional parallel hardwares may seem trivial.

After the first experiments that unofficially diverted 3D rendering cards, the emergence of specialized equipment has enabled many scientists to use these cards for their own parallel implementations.

The GPU market is specialized and relatively new, especially the GPGPU one. Indeed, these devices target two types of buyers, with the same products. Yet the needs of these two categories of users are not necessarily similar. Unfortunately for scientists, the sale volumes are not the same and 3D rendering necessarily has high priority compared to scientific computing. Yet this latter is an emerging market for these products and may become more important for manufacturers.

It has to be noted that the changes made to the GPUs, for scientific computing, cause few changes to 3D rendering. Ultimately, there is a change if it allows performance gain in both markets at the same time. The change that led to the emergence of GPGPUs (unification of pixel/vertex shaders) is a good illustration of this phenomenon. This may seem like a constraint, but it is important that these architectures continue to address these two markets together, from an economic point of view. Still, this point is not the only one to be taken into account, as high innovation also characterizes the development of these devices. Furthermore, the scientific computing market is much less competitive than the 3D rendering market. Tesla cards (graphics cards without a graphics output, such as the C-D-S870, C-M-S2050, K20-X) are an illustration of the drift that could lead to the separation of these two markets, leading to less affordable devices, targeted for a niche market.

Table 1 Flynn's taxonomy with example architectures

	Instruction management	
Data management	Synchronous for several cores	Asynchronous for several cores
Synchronous for several cores	SISD	MISD
	Von Neumann (no parallelism)	Redundancy for critical systems
Asynchronous for several cores	SIMD	MIMD
	GPGPU, vectorial processor	Multi-processors, multi-cores

2 Generalities

2.1 Flynn's Taxonomy

Modern GPGPU processors, in particular those which concern us here (the NVIDIA GPGPUs), are SIMD/MIMD. By SIMD, we need to understand, according to Flynn's taxonomy detailed in Table 1, several calculation units performing the same task on different data (Single Instruction Multiple Data). As for the abbreviation MIMD, this is a set of computing elements that can perform different tasks on different data. Vectorial calculation units, for example, can be said to be SIMD. Indeed, in these units, the same operation is applied to all elements of a vector at the same time. A very widespread example of MIMD processors is the classic multi-core processor, where each core runs its own thread, without the constraints induced by neighbour cores at the instruction level. It should be noted that strict implementations of this taxonomy are rare. Indeed, modern processors, such as Intel multi-cores, have multiple cores (MIMD) but also implement a vectorial instruction set (SSE3, MMX, . . .) and several vector registers (SIMD). Later on, we shall see exactly how these SIMD/MIMD principles are implemented into GPGPU processors.

2.2 Warp Notion

We shall see below that GPGPUs have a large number of cores and run even more tasks. These tasks are represented by threads, which contain the instructions, the program counter, the memory context and the registers on which they work. The SIMD character of a GPGPU is expressed at the logical level through the concept of a warp, which is a set of threads that are always executed together. The warps are managed by the GPGPU, and the user has no control over the creation of these warps nor their organization. We will later see this concept in more detail.

Table 2 Compute capability *w.r.t* to chip and card versions

	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Chips	G80	G86, G9x	GT215	GT200	GF100	GF106	GK104	GK110
Cards	8800 GTX	8800 GT	GT 240	GTX 285	GTX 480	GTS 450	GTX 680	Tesla K20

2.3 Compute Capability

NVIDIA has released each of its GPGPUs under a certain compute capability. This starts at 1.0 for G80 and the latest chips have a 3.5 compute capability. This main idea is to assign to a certain feature a compute capability and to implement every feature of an $n - 1$ capability in capability n. It allows us to know that such features are available on devices with compute capability x.x and beyond. While, certain features disappear during GPGPU evolution, the vast majority of compute features follow this rule (Table 2).

Finally, compute capability does not strictly follow the raw computing power of the cards. For example, a GF100 chip with 2.0 compute capability is mounted on high-end GTX 480 cards, but a GF106 chip has 2.1 compute capability on GTS 450 cards. This is related to the fact that high-end cards were the first to be launched on the market and some minor modifications were made by the time the low-end cards were commercialized, leading to minor differences in terms of compute capability.

3 Hardware

The GPGPU has a complex hardware implementation. It is interesting to study the overall chip and the different elements that comprise it. GPGPUs have undergone large modifications during their evolution; it is interesting to study some of the major steps in their evolution to draw a clearer picture of the current state. The G80 was the first chip to be GPGPU, and as such, it is a solid base on which it is possible to understand GPGPU internal functioning. With time and programming needs, the architecture has evolved and we will introduce major features along with the chip where they first appear.

The GPGPU is quite a different chip compared to a conventional central processor, as used in work stations. It is generally mounted on an internal board, connected to the main memory by a PCI-E connector. Therefore, it generally has its own memory space (except in the case of small GPUs, as in some laptops), which is almost as large as the central memory of the work station. The power of GPGPU resides more in its large number of processing cores, rather than on a few powerful computing units. Therefore, it primarily copes with largely parallel problems.

Even if they are claimed as general, GPGPUs' main purpose is 3D rendering, for which most cards sold on the market are used. The architecture can be seen as mainly oriented towards 3D rendering and marginally towards scientific computing.
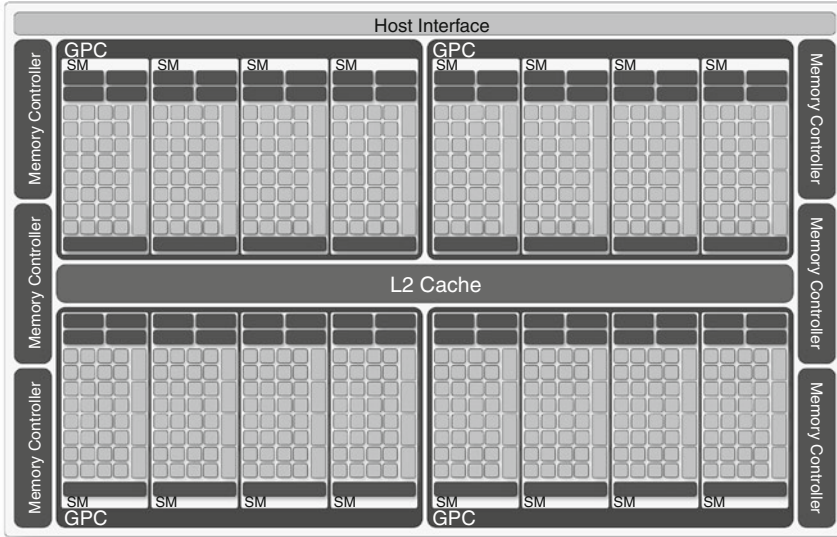
Fig. 1 A schematic view of an NVIDIA GF110 chip

The large number of cores present in these chips requires a strong and constraining structuring, which induces significant changes in the programming paradigm, especially for tasks distribution (Fig. 1).

3.1 Core Organization

The G80 contains 128 scalar cores in a chip, for a quantity of transistors close to a current processor. Increasing the number of cores, compared to an Intel P4, is performed due to the lack of certain complex features that are usually implemented on standard processors, including out-of-order execution, cache memory, and call stack. In addition, the cores are organized into groups of eight, each with two texture units, a texture cache and shared memory. Core structuring changes regularly, therefore the number of groups and scalar cores depends on the general architecture of the processor but also on the model. Entry-level models generally contain a total of fewer groups and fewer cores.

3.1.1 Streaming Processors

Streaming Processors are the basic computing elements of a GPGPU. They correspond to the core and are capable of scalar calculations. They have similarities with

Fig. 2 MP schematic views, including cores, on-chip memories and instruction units. (**a**) The MP of a G80/GT200 chip, (**b**) the MP of a Fermi chip
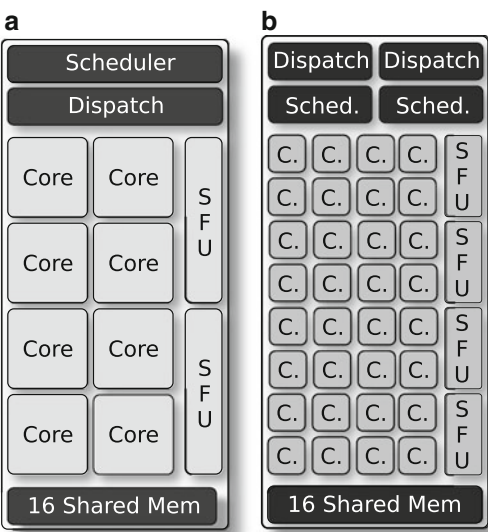

standard modern CPU cores, as they are able to handle several threads at the same time, also called simultaneous multi-threading (SMT) or Hyper-Threading (on Intel CPU). This allows the core to change its thread whenever it is stalled on a memory operation. But it is also different from a standard CPU core. For example, it does not contain computation registers or instruction units.

The cores have their own frequency called the shader clock, which can be different from the GPGPU clock. For example, on G80, the shader clock was four times the GPGPU frequency, it goes down to two times on the Fermi architecture, and this notion disappears on the Kepler architecture, where the shader is equal to the GPGPU clock.

Finally, the SP can hardly be considered as a real core, mainly because of its lack of instruction units. In order to be compatible with our usual understanding of a computing core, it has to be considered as a part of the multi-processor.

3.1.2 Multi-Processor

The multi-processor (MP) is a group of SPs, with one or several instruction units. The MP structure varies widely depending on the model. Figure 2a presents the first MP structure in G80/GT200 cards where only eight SPs were packed into an MP. The first great changes appeared with the Fermi architecture, as shown in Fig. 2b, where the number of SPs increased to 32. A new level of complication is reached with the Kepler architecture, as the number of cores rises significantly, compared to older architectures (256, 192 single precision and 64 double precision).
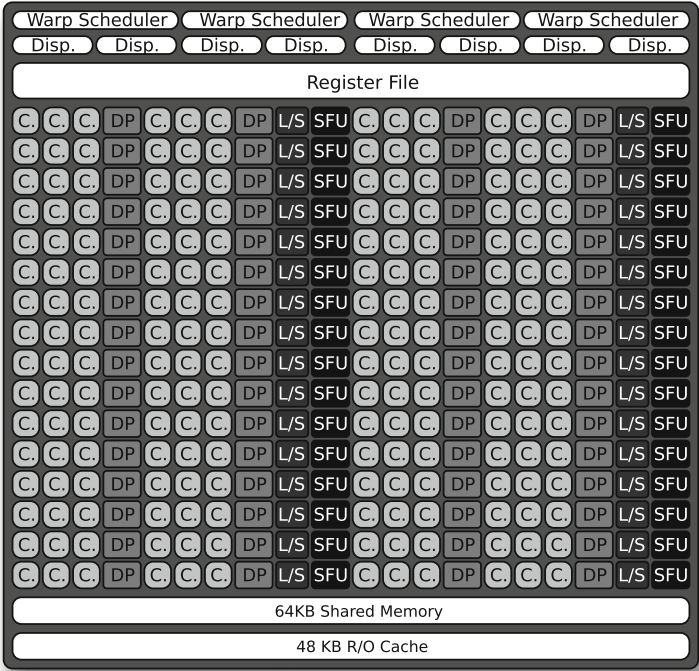
Fig. 3 A schematic view of an NVIDIA GK1xx chip (Kepler)

An MP has a set of registers that are distributed to the cores and to the sets of threads that run on it. On a G80, each MP can handle 768 threads at the same time (maximum 12,288 threads on the whole system). This number increases to 1,536 on the Fermi architecture and culminates in 2,048 on the architecture for Kepler (30,720 loaded threads; Fig. 3).

An important feature of MPs is the integration of several SFUs (Special Function Units). These units are inherited from 3D rendering and are helpful in computing fast approximations of transcendental functions (EXP2, LOG2, SIN, COS). The number of SFUs varies across the different generations of GPGPUs.

Finally, the MPs contain an on-chip memory that will be detailed in Sect. 3.2.2, as well as warp schedulers detailed in Sect. 3.3.

3.1.3 Warp

An MP always executes its threads in an SIMD way. Due to hardware constraints and in order to allow modification of the core structuration among the models, the number of threads executed together stays the same on all models of NVIDIA

GPGPUs. The *warp* is a bundle of threads that are always executed together.

Several strategies exist in executing the warps on MPs. Indeed, the G80 has eight cores and one instruction unit, which leads to a warp being executed, quarter by quarter, until the next instruction comes up. This is directly connected to the speed of the instruction unit, which is four times slower than the core speed.

The GF100 (GTX480, etc.) has 32 cores per MP and two instruction units, so each MP runs two half warps at each time. Then, two warps are performed every two clock cycles and the SPs run at two times the instruction unit clock rate.

On Kepler, 192 single precision cores are packed into an MP as well as eight instruction units. In this case, two instructions can be executed on four different warps. Here, the instructions are executed in one go, on a whole warp, due to the high number of cores in an MP.

3.1.4 Divergence

As seen in Sect. 3.1.2, several cores of the processor are structured around the same instruction unit. Executing different instructions at the same time on these cores is impossible. In order to manage code that still requires this kind of behaviour, such as in Code Snippet 1, the cores have to diverge, i.e. some cores run their common instructions (in this example the "then" part (Fig. 4b)), while the others remain idle, then the first do nothing, while the second group run their instructions (the "else" code (Fig. 4c)).

```
if( threadId/2==0 ){ // then part }
else { // else part }
```

Code Snippet 1: Code producing divergences between four cores

Obviously, these divergences lead to performance loss because some cores are idle for a moment. At high doses, these divergences slow down the execution of the code and should be avoided.

A good example of code that could produce divergences is code using the thread number in a control structure. It is possible to have a number of divergences on such an operation, up to the number of threads running in SIMD (that is, the warp size, see Sect. 3.1.3).

The definition of divergence is based on the warp notion. Indeed, only the threads within a warp are likely to create conflicts. Two threads that do not belong to the same warp therefore cannot create divergences. This means that the minimum degree of parallelism of an NVIDIA GPGPU is 32.
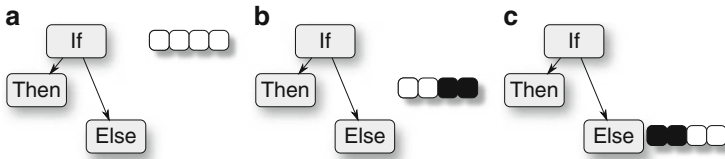
Fig. 4 The evolution of the available amount of shared memory during GPGPU evolution. (**a**) All thread tests the conditions, (**b**) first threads execute the "then part", (**c**) other threads execute the "else part"
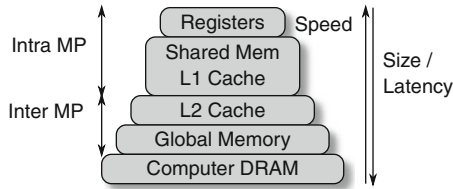
Fig. 5 Schematic view of the memory hierarchy of a Kepler GPGPU

3.2 Memory Hierarchy

As with the core organization, the GPGPU memory hierarchy is very specific to this kind of architecture. This is mainly due to its 3D rendering inheritance. The memory hierarchy is deeper than in standard processors as shown in Fig. 5, and it contains several disjoint memory spaces.

3.2.1 Cache Memory

The cache memory implementation has varied greatly during GPGPU evolution. The G80 has no R/W cache. It only has two R/O caches, one for constant data and the other for textures.

From the Fermi architecture onwards, an L1 cache is implemented of size 16 or 48 K/MP. It has to be noted that due to the large number of threads potentially running on each MP (up to 2,048), the available cache memory per thread is greatly reduced compared to a conventional processor (64 KB for two threads of a core on an Intel Core i7).

From a programming point of view, G80 can use cache memory to store constant data. Programmers can also transform data into textures, in order to use their caches to speedup memory accesses (by using spatial and temporal locality).

The Fermi architecture allows the programmer to use the L1 and L2 caches, in the same way as with a standard processor (i.e. transparently), keeping in mind that the size per thread will be small. The cache system is built on top of the device memory and uses the same memory address space. Furthermore, using the largest

setting for L1 cache sizes (48 KB) will reduce the size of the shared memory, as we will see in the next section. L2 cache is common to all the cores of the processors, and its maximum size is 768 KB.

In the Kepler architecture, global variables are not cacheable in L1. At best they are stored in the L2 cache. Here, L1 is reserved for local variables and register backups. L2 cache on Kepler grows to a maximum of 1.5 MB.

Programming with constant and texture caches can be beneficial on G80, but it can prove to be counterproductive on newer models ([3]). Considering the age of this kind of GPU, and the complexity in implementing this technique, this should disappear quickly from GPGPU programming techniques.

3.2.2 Shared Memory

Each MP has a memory called "shared", by which we must understand that it is shared between threads running on the same block. G8x and GT2xx architectures have 16 KB of this shared memory on each MP. Shared memory has to be directly accessed in the code, as it has a different memory space from standard GPU memory. This memory is very fast (within an order of magnitude comparable to registers) and has complex access modes.

In fact, it is organized in different banks, which are served by a bus allowing very fast access, with particular configurations. The general idea is that the access is the fastest if each bank serves one thread. Otherwise, the accesses are serialized, and the access time is multiplied by the number of threads that access the bank. This behaviour is called a *bank conflict*.

Firstly, on G8x and GT2xx, one-to-one accesses are allowed without conflict. This architecture uses 16 banks and the threads are served by half warps (16 threads at a time). The address space is spread over all banks by 32 bits. Finally, the G8x/GT2xx shared memory bus allows values to be read that are broadcasted from a bank to all the cores, i.e. if all threads of a half warp read a variable that is in the same word of 32 bits.

From Fermi onwards, several threads of a (half) warp can access the same bank, without conflict if they access the same 32-bit word. In this case, the word is sent to the requesting threads, which extract the needed part. Conflict persists if two threads try to access two bits that do not fall within the same 32-bit word, but still belong to the same bank.

In Kepler, the number of banks rises to 32, allowing the warps to be served entirely (and no longer by half warps). The same operation as above is extended to words of 64 bits.

The shared memory is an important feature of NVIDIA GPGPUs. It is very fast to access, because it is close to the cores, but it is also difficult to use, as its size is small compared to standard DRAM devices. Furthermore, other complications have appeared during GPGPU evolution. The increase in number of cores and manageable threads has side effects on the use of this memory.
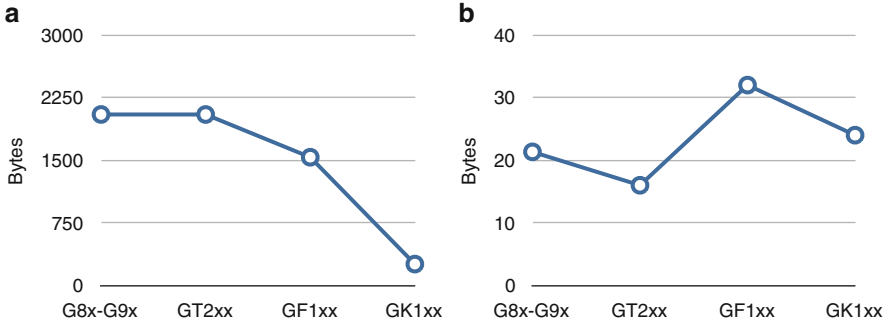
Fig. 6 The evolution of the available amount of shared memory during GPGPU evolution. (**a**) SHM per core, (**b**) SHM per thread

Implementations that use shared memory to store thread-specific data are difficult to maintain. Indeed, the amount of shared memory available for each thread will vary depending on the model, and an adaptation to the processor is necessary to port the algorithm to different processor generations.

Indeed, Fig. 6a, b shows the evolution of the shared memory ratio per core and thread on different important architectures. It has to be noted that the available shared memory per core tends to decrease with time, mainly due to the sharp increase in the number of cores. Available memory per thread also undergoes a contrasting trend, by going down when the number of schedulable threads increases, and finally going up on the Fermi architecture when shared memory size is increased to 48 KB. Finally, the last generation card causes a further decrease of the value, which returns to a value close to what it was originally.

However, shared memory is the privileged place for communication between threads of the same process. It allows for quick exchange of information between the cores, if communications are small sized. The amount of shared memory used by a block can be determined statically at compile time or dynamically at runtime. As we will see in Sect. 3.3, the amount used by a block must be suitable, in order for that block to be placed on an MP.

3.2.3 Registers

As discussed in the section dedicated to software 4, a large number of threads are loaded on a core at a given time. This leads to the use of a large number of registers. To maximize the capabilities of the register allocation, registers are not dedicated to a core in particular but are potentially common to all the cores of an MP and are assigned to a particular thread when the thread execution starts. Registers are installed in the "register file" to be used by all the threads loaded onto an MP.

Table 3 Evolution of register characteristics during time

	G8x-G9x	GT200	GF1xx	GK104	GK110
Registers per thread	128	128	63	63	255
Registers per MP (K)	8	16	32	64	64

It has to be noted that the number of registers in the case of GPGPU is more important than in a conventional processor. Indeed, the lack (or low size) of cache can cause a memory access to be expensive for each register backup. Therefore, a large number of registers are important for the proper functioning of a program on GPGPU, in order to avoid access to the main memory or an L1 cache saturation.

The number of registers that can be used by a thread, and the number of registers implemented into MPs, has varied over generations of cards, as presented in Table 3. In Sect. 4, we shall see how the number of registers can affect the behaviour of a GPGPU program.

3.2.4 Device Memory

The memory device is the widest memory of the card. It is directly accessible by the GPGPU and has a separate memory space, which is common with L1 and L2 caches. The local and global variables are stored in this memory. It is a fast memory, with high latency. However, the memory bus is relatively large and can be used by multiple threads in a single operation if access patterns are correct. This memory is accessed by a transaction system, allowing the loading of several variables per transaction.

Processors with compute capabilities 1.0 and 1.1 (G8x, G9x) allow memory access by multiple cores on the same half warp at the same time, if these accesses fall into a common segment, which should be 64 bytes for 4-byte words, 128 words of 8 bytes and 128 bytes for $2x$ 16-byte words (a half warp here is served by two transactions; thus, a warp is served by four). In addition, the words must be accessed in sequence (the nth thread using the nth word of x bytes). For processors with compute capability 1.2 and 1.3 (GT2xx), the sequential access constraint is released, and the words can be accessed in any order by the threads of a half warp, as long as every word fits into the corresponding segment. If these constraints are not met, 32-byte transactions are performed separately for each thread of the half warp.

For higher compute capability processors, the use of the cache simplifies the transactions, as they are made directly through the cache or used by the threads, allowing more complicated schemes than with previous versions. It is also possible to reduce the number of transactions if the data can be loaded by the first half of the warp and retained until the execution of the second half warp. With previous generations, the two half warps caused data loading, even if one transaction could have brought all of them in one load. Indeed, no storage allows the data to wait until the execution of the second half warp.

Ind$_0$ G$_0$	Ind$_1$ G$_0$	Ind$_2$ G$_0$	Ind$_3$ G$_0$	Ind$_0$ G$_1$	Ind$_1$ G$_1$	Ind$_2$ G$_1$	Ind$_3$ G$_1$	Ind$_0$ G$_2$	Ind$_1$ G$_2$	Ind$_2$ G$_2$	Ind$_3$ G$_2$	Ind$_0$ G$_3$	Ind$_1$ G$_3$	Ind$_2$ G$_3$	Ind$_3$ G$_3$

Fig. 7 A gene-wise population organization

This type of well-shaped, sequential access is called coalescent memory access, but the term is mainly used for the first processor models, and it tends to disappear from CUDA literature due to recent cache improvement.

The main memory is usually widely used, due to its impressive size. It may be important to make proper use of it, and in particular of its transaction system, which reduces up to $32\times$ the number of memory accesses. The reorganization of the data can help improve this behaviour. In artificial evolution, the organization of a population of individuals by genes rather than by individuals as in Fig. 7 allows the threads of a warp to use the result of one transaction several times.

While the implementation of cache memory on Fermi architectures allows caching of local variables, the small amount of cache memory should motivate programmers to focus on optimizing the SMT behaviour (Sect. 3.3) and the memory transaction behaviours.

3.2.5 Host Memory

One last memory deserves to be mentioned here. It is of course the standard memory of the host machine. This space is not accessible directly by the GPU, but a large part of the data are loaded from this memory. They should indeed be sent by the program running on the CPU to the GPU memory, in order to be used by the program running on it.

This space is often the largest memory space of a machine and it allows access to other devices of the host machine (including other GPGPU memories). It is therefore essential to access it in order to load the initial data, export results, etc. The exchanges between the CPU and the GPU memory are executed through DMA transfers that are initiated by the CPU program, which indicates the address of the source and destination buffer, these addresses being located in different spaces (GPU vs CPU).

These DMA transfers have significant latency, but a high speed. It is important to reduce the number of transfers as much as possible, even if larger transfers should be made. The flattening of the tree structures can allow a reduction in transfers and facilitates the calculation of addresses on the GPGPU side.

3.3 Simultaneous Multi-threading

The GPGPU has a relatively high memory latency and relatively little cache memory to compensate (especially compared to the number of loaded threads). The optimization of the memory behaviour is based on a multi-level scheduling
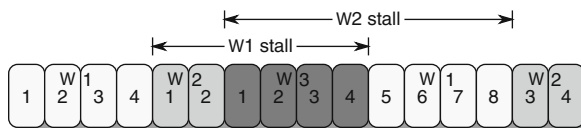
Fig. 8 An example of warp scheduling

mechanism. Indeed, we have seen that SPs are able to execute several threads "at a time", thanks to SMT.

This means that a GPGPU will handle a great deal of threads at a time, and the schedulers will map threads to SPs whenever they are ready to be executed. As threads are always executed with the other threads of their warp, the schedulers handle warps as their scheduling unit. Similar to SMT, a warp is replaced by another whenever it is terminated or it attempts a memory operation that provokes a latency.

The warps are placed on an MP and stay on it until termination, i.e. there is no warp migration between MPs, as the execution environment (registers, shared memory, PC, etc.) is not shared across the whole chip. Figure 8 presents an example of scheduling. The first warp executes four instructions before accessing memory and being stalled; it is replaced by warp 2 for two instructions, until it is also replaced by warp 3. The number of operations that the SPs perform can vary, depending on the architecture.

4 Software

As we have remarked in the hardware part, GPGPU architectures vary greatly with each generation. Moreover, even within a generation, the configuration of the chip (in particular, the number of cores) may change. This can cause complications when porting programs to different target architectures. To overcome this problem, NVIDIA has developed a software framework called CUDA that provides some abstraction of the underlying hardware, which facilitates portability even across different architectures and understanding of the underlying processor.

CUDA in particular defines the logical organization of threads in a program, provides an intermediate language compiler compatible with all the cards that support it and defines a set of computing capacities that the cards can support. A debugger is also provided, as well as a set of libraries, which are useful for scientific computing.

4.1 Logical Thread Organization

The warp defines an interface notion between the hardware and the software. Yet this is not the only existing thread structure. CUDA terminology defines two levels

Table 4 Block and grid constraints

|	Compute capability			
	1–1.1	1.2–1.3	2.x	3–3.5
Max block grid dimension	2		3	
Max size of x dimension	2^{16}			$2^{31} - 1$
Max size of y, z dimension	2^{16}			
Max thread block dimension	3			
Max size of x, y dimension	512		1,024	
Max size of z dimension	64			
Warp size	32			
Max number of threads per block	512		1,024	
Max number of threads per MP	768	1,024	1,536	2,048
Max number of blocks per MP	8		16	
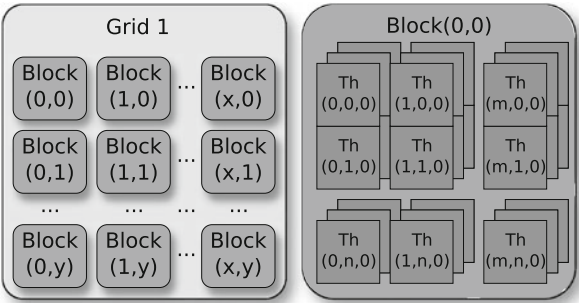
Fig. 9 Thread organization per block and grid

of structures above the threads. Indeed, threads are grouped in a block and the blocks themselves are grouped in a grid, which forms a kernel, i.e. a GPGPU program. A grid is assigned to a GPGPU, and each block is assigned to an MP. In addition, the MP can be assigned to several blocks, if the total number does not exceed the thread hardware scheduling limits. Similarly, multiple grids can be assigned to a GPGPU.

These groups (grid and block) take a multidimensional form (detailed in Table 4), which is used to map threads to data tables used in the calculations. Figure 9 shows the logical structuring of threads per block, then blocks per grid. The links with 3D rendering are obvious here.

Possible configurations vary greatly depending on the compute capability of the GPU. For example, only blocks of arrays are possible for compute capability below 2.x, while cubic blocks become possible above. Yet limitations exist in the matter of size, as a limit on the number of threads per block, even if dimensional limits should allow greater thread sets.
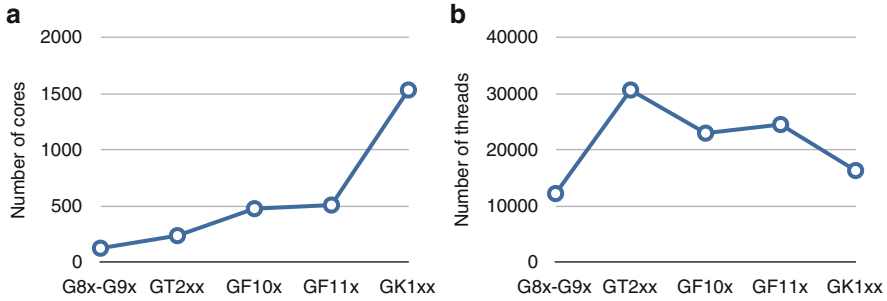
Fig. 10 The evolution of the number of cores and manageable threads per GPGPU over generations. (**a**) Cores per processor, (**b**) manageable threads per processor

4.2 Scheduling Usage

Scheduling greatly optimizes the access time to the memory. If scheduling is done at two levels that are linked to thread organization, in blocks and grids, it has also to be considered in a global manner. Indeed, an algorithm to be parallelized can be divided into a given number of tasks, and it is often difficult to make an arbitrary distribution. For example, in the case of the parallelization of an evolutionary algorithm with the master–slave model (where only the evaluation of the population is parallelized), the number of tasks is fixed (i.e. the number of individuals at maximum). The overall population needs to be considered and distributed into a set of blocks while taking into account the number of threads in each block, which should be set to the most suitable configuration. Here, this problem remains simple, because the evaluation of an individual is independent of other individuals in the population. An individual may be placed in one block or another, without affecting the algorithm.

It is also important to consider the SIMD behaviour of a group of threads in order to create the blocks. The first approximation is that a set of tasks in a block must execute the same instruction at any time. It is possible to refine this statement, because different threads located in different warps can execute different instructions, without loss of performance. Finally, it is also acceptable for threads in a warp to execute different instructions, but divergences will appear and performance will drop.

To ease the scheduling process, it is important for the number of tasks to be as large as possible. The scheduling process is described by the manufacturer as very fast, and no significant overhead should be considered here. We have seen in Fig. 10b that the number of threads loaded onto a processor is high. If the application allows, it should be divided into a large number of independent tasks, and these tasks should avoid synchronization, which is expensive here, as on most parallel architectures.

If the tasks are completely independent and can be grouped freely, it is easy to create blocks and threads that will maximize the use of schedulers. The blocks must have, preferably, a number of threads which is a multiple of 32 (warp size). The grid

Table 5 Number of threads per block or MP

	Compute capability			
	1.0–1.1	1.2–1.3	2.x	3.0–3.5
Max number of threads per block	512		1,024	
Max number of threads per MP	768	1,024	1,536	2,048

must contain a number of blocks that is a multiple of the number of MPs per card. This information can be found in the technical documentation of the card but also dynamically thanks to the function detailed in the Device Management section of the CUDA Runtime API [2] or in the Driver API [1].

It is still necessary to allocate more blocks than there are MPs on the card, as a block cannot contain enough threads to saturate the scheduler. Table 5 summarizes the maximum number of allocatable threads in a block or an MP.

4.3 Scheduling Evolution

A block uses some resources of the MP on which it is placed. Before loading a block, it is necessary for these resources to be available on the target MP. These resources are the number registers and the shared memory needed by each thread of a block.

Since all threads in a block are executed, potentially at the same time, and the SMT mechanism allows the thread data to stay in the registers, a thread uses its resources from when it is loaded, until it is cleaned (its block is terminated). If the number of registers used by threads multiplied by the number of threads in the block exceeds the number of available registers on the MP, the execution of this block is not possible. When placing a second block at the same time as another, the amount of registers left available by the first block should be sufficient to execute the second, with the same mechanism as above.

The shared memory has to be distributed by a very similar process. The only difference comes from the fact that shared memory is assigned to a block and not to a thread. Otherwise, the shared memory consumed by a block is considered in the same way as the number of registers by the allocation mechanism.

The number of registers used by a thread and the amount of shared memory used by a block can generally be seen at compile time. It can also be dynamically calculated by CUDA library functions such as cudaFuncGetAttributes. This method takes into account both software consumption and hardware resources and gives the maxThreadPerBlock value directly. Information about these functions can be found into the section "Execution Control" of the Runtime or Driver CUDA API [1, 2].

It is interesting to design the thread groups of a GPGPU program in order to fulfil the scheduler, in particular to reduce the number of registers and shared memory used by a block, so that two can be placed at the same time on an MP.

Reducing the amount of shared memory in order to increase the number of loadable threads can have a negative effect on performance, especially if the change is accompanied by a massive increase in the device memory usage. Conversely, increasing the use of shared memory can reduce performance if the reduction of the number of schedulable threads reduces the memory performance of the program. It is thus a balance that depends mainly on the current GPGPU program.

4.4 Stream

Due to the high-level management character of blocks, it is possible to place multiple grids on a chip. Indeed, the blocks from different grids having their own resources, as well as blocks of the same grid, can therefore coexist on an MP or on a single chip. From Fermi onwards, it is possible to run 16 grids simultaneously and a transfer host/device in each direction at all times.

This method allows programmers to harness the power of a chip, using several small programs, rather than with a single more power-consuming one. In addition, an algorithm can be cut into several sub-algorithms, allowing better load balancing and concurrent execution of steps.

However, tasks must be independent, which is typically not the case, if an algorithm is cut into several sub-algorithms. This introduces a notion of dependency between tasks, which can be difficult to satisfy.

CUDA introduces the notion of stream, which is used to indicate the existence of a dependency between grids. A stream thus allows a flow organization of calculations. Different flows may run on the same chip at the same time, but not different stages of the same flow.

The standard execution of a program uses standard streams (number 0), but programmers can use other streams to add dependant tasks and to ensure one will be launched after another. The memory transfers (in and out of the GPGPU memory) will take place automatically before and after their respective kernel.

5 Conclusion

GPGPUs have a complex architecture which is very different from that of standard processors. The hardware model is different, and the provided software does not yet entirely cover the gap between these two types of architectures. Indeed, the core architecture is unusual (by its vastness and constraints) but so too is the memory hierarchy, which contains more levels than traditional CPUs. Furthermore, GPGPU is a processor that does not run any operating system and is attached to the host as a stand-alone device.

GPGPU programming has undergone significant changes in recent years. Starting from a diversion of rendering hardwares to become an official solution

developed by manufacturers, it has become much easier to use. GPGPU programmers have a modern interface to the hardware, at a much lower level than previously (avoiding an OpenGL or DirectX layer), which is therefore more efficient and generic. However, the programmer is therefore more exposed to changes in the processor architecture. In addition, it is also increasingly linked to a manufacturer, particularly using the CUDA framework.

Shared memory is a good illustration of this principle with the drastic changes that have been made during GPGPU evolution. But these cards still have a cost-to-performance ratio (whether to learn or to buy) that remains interesting. In addition, it is a very common feature which will allow the use of a parallel port on a large number of machines. The development of large GPGPU clusters also motivates a part of the scientific community to port and develop algorithms targeted at these architectures.

References

1. CUDA 5.0 driver API documentation. http://docs.nvidia.com/cuda/cuda-driver-api/index.html
2. CUDA 5.0 runtime API documentation. http://docs.nvidia.com/cuda/cuda-runtime-api/index.html
3. CUDA v5.0 Kepler tuning guide. http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html
4. Fok, K.-L., Wong, T.-T., Wong, M.-L.: Evolutionary computing on consumer graphics hardware. IEEE Intell. Syst. **22**(2), 69–78 (2007)
5. Kedem, G., Ishihara, Y.: Brute force attack on Unix passwords with SIMD computer. In: Proceedings of the 8th Conference on USENIX Security Symposium, vol. 8, p. 8. USENIX Association, Berkeley (1999)
6. Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Proceedings of Advances in Natural Computation ICNC 2005, Part III, Changsha, 27–29 August 2005. Lecture Notes in Computer Science, vol. 3612, pp. 1051–1059. Springer, Berlin (2005)