Master 2 Informatique
Université de Bordeaux I

Projet d'**É**tude et de **R**echerche

Concurrent kernel execution on Graphic Processing Units

Adrien Cassagne, Aurélien George, Benjamin Lorendeau,
Jean-Charles Papin, Antoine Rougier

Supervised by S. Chaumette, J.C. Coquilhat and P. Desbarat
Articles proposed by Raymond Namyst

Bordeaux, January 23, 2013

Contents

List of Figures

I Introduction

General Purpose Graphic Processing Unit (GPGPU) are now used in high performance computing (HPC) for their massively parallel computing aspect and capabilities. Those devices integrate hundreds of computing unit (computing core). Usually, such a level of parallelism is used to solve simulation problems (heat transfer, ...) because of the numerical representation of simulated environment (matrices).

Those GPU can be programmed with specific programming languages like ***CUDA***[1] and ***OpenCL***[2] which provide a standard environment (C/C++ libraries).
Programs executed on a GPU (also called kernels) are executed sequentially. However, in order to maximize the usage of GPU resources, some advanced features (developed by *NVIDIA*) allow programmers to execute severals kernels in parallel on the GPU.

Unfortunately, concurrent kernels execution is only possible with *CUDA* on *NVIDIA* graphics cards. For other cards, *OpenCL* does not offer this functionality. That is why researchers from University of Virginia (USA) [2], tried to extend *OpenCL* standard by allowing execution of an "master kernel" which will launch other kernels. In fact, the "master kernel" is a mix of memory-bound and compute-bound kernels. By doing this, they could evaluate the advantage of this kind of solution.

Another group of researchers (from University of George Washington and from University of Arkansas), designed a software environment that allows different threads from the same process to share access to the GPU, which wasn't possible until the introduction of the "Automatic Context Funneling" [2] capabilities in *CUDA 4.0*.

For our PER (Projet d'Étude et de Recherche), we will analyse the benefits and limitations of concurrent kernel execution. We will also determine if parallel kernel execution can be used to avoid the cost of data transfers from the host to the GPU (by starting long computing time kernel before starting data transfers).

[1] *NVIDIA* Software development kit
[2] Open standard maintained by the consortium Khronos Group [1]

II Concurrent kernel execution

II.1 Issues

GPU programming is not simple. Indeed, it is based on the SIMD[3] model. That means one single instruction can be execute on different data at a time. It needs specifics algorithms which have to express lot of parallelism. As an example, we can imagine an algorithm which has to apply the same function to each pixel (block of pixels) in a picture. Executing each process at the same time on several processing units is a major process in order to gain performance.

The first step in GPU programming is to make an efficient kernel. When we talk about efficient kernel, it means we want to maximize the GPU resources usage. It is not always an obvious job because the kernel has to be fine-grained. Indeed, some kernels can't express enough parallelism because of their memory or computing bound.

In order to maximize the GPU resources usage, *NVIDIA* introduced a technology called "concurrent kernels". This new feature allows programmers to start several kernel instances on the same GPU.

II.2 State-of-the-Art

To design a GPU program, there are two major frameworks : *CUDA* [3] and *OpenCL*[4] [1]. *CUDA* is a Software Development Kit designed by *NVIDIA*, which allows to create programs that will run on *NVIDIA* GPU cards. *OpenCL* is an alternative framework to *CUDA* that can also execute kernels on GPU. *OpenCL* is an open standard maintained by Khronos Group gathering *Intel*, *AMD*, and *NVIDIA*.

As explained previously, CUDA allows concurrent kernel execution. Initially introduced into Fermi cards (see *NVIDIA's* architecture time line 1), this feature was a little bit restrictive: only one application can use GPU at a time, and for this application, only one thread can use the GPU at a time. Concurrent kernel was so limited by the ability of each thread to start concurrent kernels.

Year	Architecture
2007	*Tesla*
2010	*Fermi*
2012	*Kepler*

Table 1: Time line of *NVIDIA* architecture

The article "Exploiting Concurrent Kernel Execution on Graphic Processing Unit" [2], explains how we can avoid this limitation by using a "master" thread which will use the

[3]Single Instruction Multiple Data
[4]Open Computing Language

GPU: all other application threads will then use GPU ressources by submitting kernels to the master thread.

Later, in *CUDA* 4.0, *NVIDIA* introduced "Automatic Context Funneling" which allows application threads to use GPU independently. Finally, with latest *NVIDIA* graphic cards architecture (*Kepler*), and *CUDA* 5.*x*, it is now possible to use GPU across different applications and different threads.

For *OpenCL* programs, it's not possible to run concurrent kernels as in *NVIDIA* technology. However, it's possible to bypass this limitation by hijacking *OpenCL*. This technique is developped in the article "Fine-Grained Ressource Sharing for Concurrent GPGPU Kernels" [4] and will be aborder in the next chapter.

***NVIDIA CUDA* ToolKit**	**Architecture introduced**	***CUDA* Inovation**	***OpenCL* Inovation**
1.0 (2007)	*Tesla*	First *CUDA* release	
...	...	...	...
3.0 (2010)	*Fermi*	First *OpenCL* Implementation	
...	...	...	...
3.2 (2010)		Context Switching Manual Context Funneling [2]	Pseudo concurrent kernel [4]
4.0 (2011)		Automatic Context Switching	
...	...	...	...
5.0 (2012)	*Kepler*	-Hyper-Q (GPU shared among multiple applications)	

Table 2: Major *NVIDIA*/*OpenCL* dates

III Two points of view

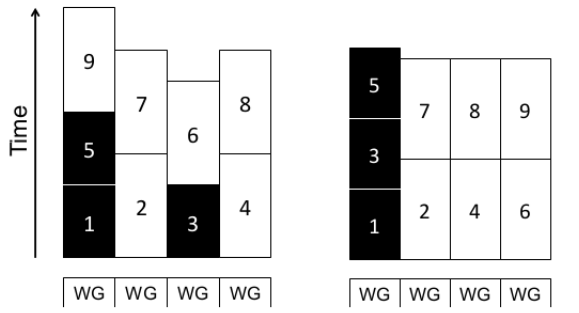
III.1 Approach based on *OpenCL*

With *OpenCL*, it is not possible to run concurrent kernels. But a method which consists in merging several kernels into one big kernel to execute them at the same time. This method has been implemented in *Fine-grained resource sharing for concurrent GPGPU kernels* [4] which we now explain.

A A software approach

In this article, researchers chose a software solution. Among *OpenCL*'s methods, there is a method called `clEnqueueNDRangeKernel()`, this is a method which executes a single kernel on the GPU. They created a scheduler called $KernelMerge$ which combines two invocations of `clEnqueueNDRangeKernel()` into a single kernel. In other words, the GPU executes a single kernel built by $KernelMerge$ which mixes smartly the two kernels to use maximum power of the GPU.

They implemented two algorithms in $KernelMerge$ which combine two kernels. The first is based on a round-robin work-stealing algorithm that assigns work from each kernel on a first-come, first-served basis. The second algorithm assigns a fixed percentage of ressources to each individual kernel. The figure 1 is a good view of these algorithms. We can see easily how the first algorithm distributes tasks between scheduler workgroups. The second algorithm assigns scheduler workgroups between both kernels and the number of scheduler workgroups is chosen between the two kernels depending on their percentage of work.



(a) Work-stealing Algorithm (b) Partitioning Algorithm

Figure 1: The black boxes represent one kernel and the white boxes represent the other. WG is a scheduler workgroup on a GPU.

B Limits and results

To evaluate their *KernelMerge* they ran pairs of kernels concurrently, then ran both kernels sequentially. Comparing both executions, they got a speedup which is upper to 1 for only 39% of the paired kernels tested, the best pair achieves 1.2 speedup, the worst pair is 3 times slower than the sequential version.

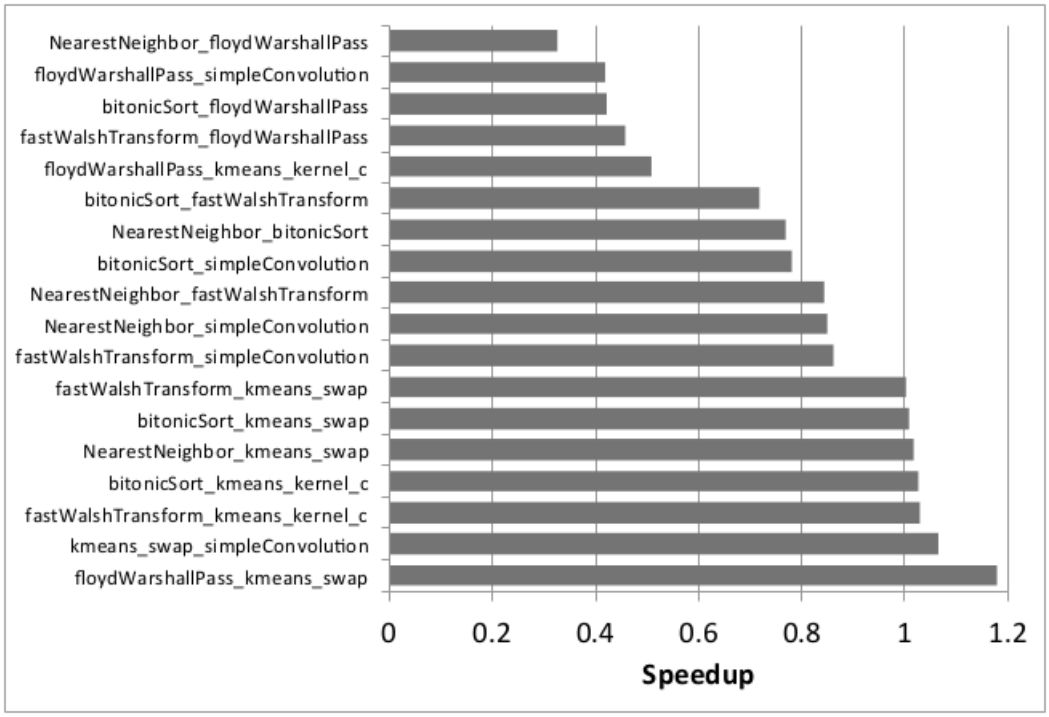


Figure 2: Merged runtimes to sequential runtimes
Gregg et al. [4]

KernelMerge is obviously not a good way to increase performance on the GPU. Scheduling pair of kernels is not a simple problem and requires a hard analysis to be efficient. Making a scheduler for a GPU is even more complicated and in this article, researchers had thought about heterogeneous kernels (a kernel which uses many computing ressources and few data ressources, and conversely for the other kernel) complement each other. The simple fact of scheduling produces an overhead, but it doesn't lose much time in most kernels. The main limit is that two kernels can't be associated in most cases, on the contrary if they ran concurrently, they use same ressources simulteneously and penalize each other.

III.2 Method using *CUDA*

Until recently kernels could only execute sequentially but concurrent multi-kernel execution over GPU has been recently introduced by *NVIDIA* in their previous GPU architecture called *Fermi*. While it is a strong improvement it has one peculiar drawback that only kernels of the same host thread context can execute in parallel[5]. Thus kernels from different contexts still have to execute sequentially, inducing context switching and overhead.

A Context switching

Before *Fermi* GPUs, context switching was the only option to access GPU ressources. This method only allows to have different application threads to be queued, thus **each application thread will be executed sequentially**. It is easy to see how this implementation is limited since if the application thread which has access to GPU does not express parallelism, it will use only one tiny percentage of theses resources while other threads could have the use of GPU left resources. Besides, context switching implies overhead due to the time needed to store and restore the state of different contexts.

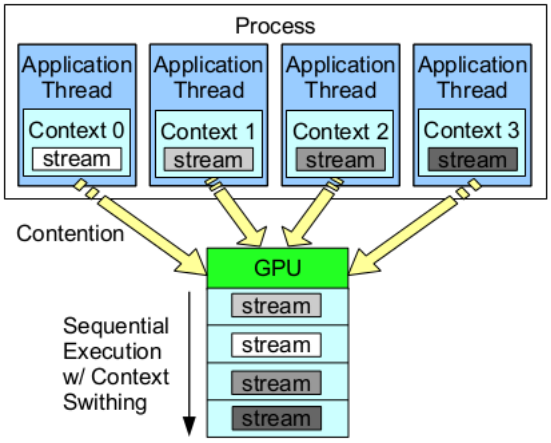


Figure 3: Context switching model. Wang et al. [2]

B Manual Context funneling

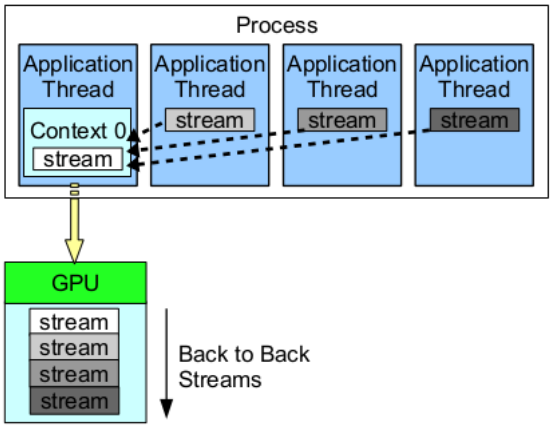


Figure 4: Manual context funneling model. Wang et al. [2]

To address this peculiar problem, Wang et al. have experimented manual context funneling to deal with the limitations of context switching. As we can see on the left-sided picture, there is only one context and each application thread must access GPU resources through a master thread. Thanks to this method, **each thread can be executed concurrently**. Thus, we remove the context switching overhead and let all threads have the possibility to access GPU resources through one master thread.

[5]This has been fixed with *Kepler* GPUs, the latest *NVIDIA* GPU architecture.

C Auto Context funneling

Since *NVIDIA CUDA 4.0*, there is no need for specific software implementation of the context funneling method. According to *NVIDIA*, the new model for runtime program is "one context per device per process". Thus, this provides auto context funneling and removes any need of context switching or manual context funneling. While it is quite similar to the manual context funneling designed by Wang et al. it provides two interesting features as it makes memory shared between application threads and removes the master thread mode, allowing all threads to access GPU resources.

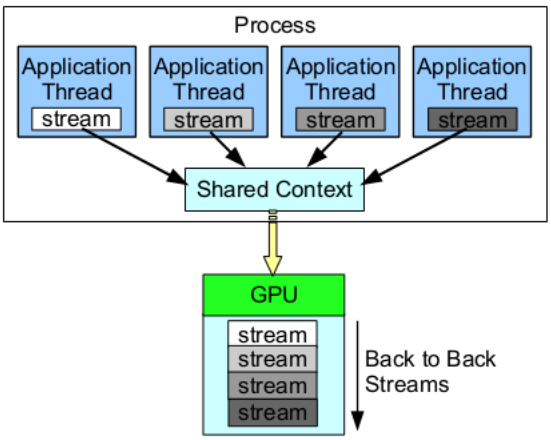


Figure 5: Auto context funneling model. Wang et al. [2]

D Limits and results

One of the limits of both manual and auto context funneling is that it is still not possible to have different applications be executed concurrently. *NVIDIA* has overcome this limitation with the latest GPU architecture : *Kepler*. Overall performances of both manual and auto context funneling are really close, and show performance improvements in comparison to context switching.

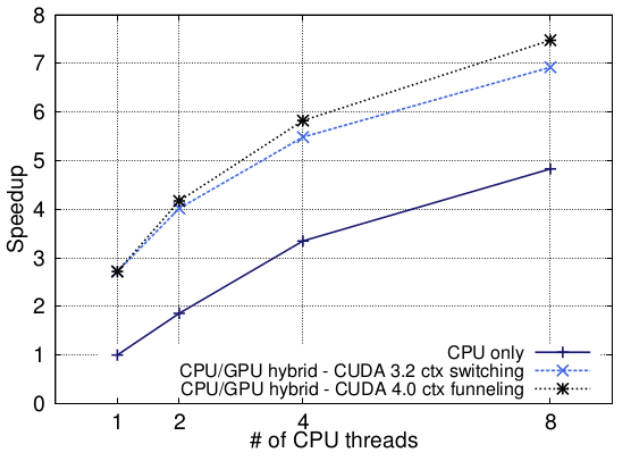


Figure 6: Context switching and funneling speedup. Wang et al. [2]

In figure 5 produced by Wang et al. in *Exploiting Concurrent Kernel Execution on Graphic Processing Units* [2], we can see the speedup gain produced by the use of context funneling. But we also can see, that in a real application, performance of context switching and context funneling are nearly the same when the number of cpu threads is low (2) but also that the difference becomes stronger as this number grows. It is one limitation of context switching.

In figure 4 produced by Wang et al. in *Exploiting Concurrent Kernel Execution on Graphic Processing Units* [2], we can see that auto and manual context funneling both perform the same way. While it can be seen as a good point for Wang et al. it lets us think that interest of using manual over auto funneling are limited. Indeed, if the same performances can be achieved using CUDA 4.0 auto funneling, why would we want to make the use of manual funneling. However, when control over data and task dependencies is important, it might be easier to do it using manual funneling.

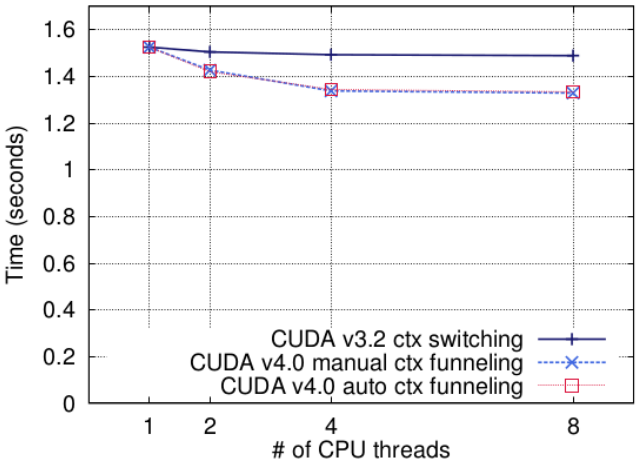


Figure 7: Execution time of auto and manual context funneling. Wang et al. [2]

IV Experimentation

Our experiments are focused on concurrent kernels. We'd like to analyse the advantages of executing several kernels at a time and to highlight some new programming mechanisms in order to maximize performances.

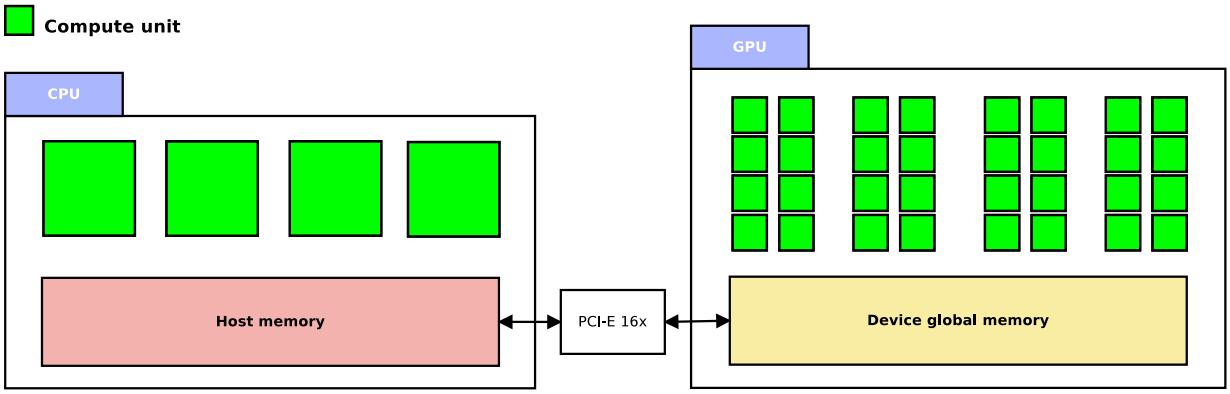


Figure 8: CPU and GPU architecture

Figure 8 shows you a classic architecture with a CPU and a GPU. Notice that you have to copy host memory to device memory when you want to execute a GPU kernel. This transfer can take time because it has to use the *PCI Express* port.

IV.1 Experimentation platform

For our experimentations, we use *CUDA 5* and machines from the CREMI (Centre de ressources pour l'Enseignement des Mathématiques et de l'Informatique) of the University of Bordeaux 1, and personal machines.

Quadro 4000	**GeForce 660 GTX**
Fermi architecture	*Kepler* architecture
256 *CUDA* cores	960 *CUDA* cores
2 GB GDDR5 at 89.6 GB/s	2 GB GDDR5 at 144.2GB/s
Compute capabilities 2.0 (up to 4 concurrent kernels)	Compute capabilities 3.0 (up to 16 concurrent kernels)

IV.2 Experimental kernels

A `kern_n2`

This kernel computes an n^2 iterations (imbricated loops) of an non-pertinent operation. We can tune the n numerical constant to change the computation time of this kernel. Even if this kernel is completely non-efficient (the kernel doesn't use all the GPU ressources), it offers us an easy way to compare different kernel sizes.

```
__global__ void kern_n2(unsigned int size)
{
  /* init variables */
  unsigned int i, j, k = 12, val = 0;

  /* n^2 computation */
  for(i=0; i < size; i++)
    for(j=0; j < size; j++)
      val += (i + j / k) / size;
}
```

B `kern_mem`

Unlike the previous to the previous kernel, this kernel uses data from the machine host. We must allocate and transfer data from the machine memory to the GPU memory and copy back GPU memory to host at the end. Like the previous kernel, we can tune the size of the total size transfered to the GPU. Due to PCI-E port limitations, the transfer time can be longer than the computation time on GPU. By combining this kernel with **`kern_n2`**, we hope to show that we can cover the transfer time by computation time of another kernel.

```
__global__ void kern_mem(const float* inBuffer, float* outBuffer)
{
  /* declare some shared memory */
  __shared__ float shData[BLOCK_SIZE];

  /* copy global memory into shared memory */
  __syncthreads();
  shData[threadIdx.x] = inBuffer[blockDim.x * blockIdx.x + threadIdx.x];
  __syncthreads();

  /* do an addition with shared memory (better than global memory) */
  float tmpVal;
  for(unsigned int i = 0; i < BLOCK_SIZE; ++i)
    tmpVal += shData[i];

  /* write value into global memory */
  outBuffer[blockDim.x * blockIdx.x + threadIdx.x] = tmpVal;
}
```

C `kern_mat_mult_perf`

This kernel computes the block matrix multiplication of the square matrix A by the square matrix B into the matrix C. The main advantage of this kernel is that we can launch several instances of this kernel on the GPU in order to solve different parts of the matrix multiplication. Thus, by executing many instances of our **`kern_mat_mult_perf`**, we hope to see an interesting speedup by using concurrent kernel execution.

```
1  __global__ void kern_mat_mult_perf(Matrix A, Matrix B, Matrix C)
2  {
3    /* declare some shared memory */
4    __shared__ float shDataA[BLOCK_SIZE][BLOCK_SIZE];
5    __shared__ float shDataB[BLOCK_SIZE][BLOCK_SIZE];
6
7    float valC = 0;
8
9    /* compute matrix multiplication per blocks */
10   for(unsigned int i=0; i < A.size; i += BLOCK_SIZE)
11   {
12     /* copy global memory into shared memory */
13     __syncthreads();
14     copyGlobalMatrixBlockIntoSharedBlock(A, shDataA, ...);
15     copyGlobalMatrixBlockIntoSharedBlock(B, shDataB, ...);
16     __syncthreads();
17
18     /* final computation, matrix multplication on shared memory */
19     for(unsigned int j=0; j < BLOCK_SIZE; ++j)
20       valC += shDataA[threadIdx.y][j] * shDataB[j][threadIdx.x];
21   }
22
23   /* write result into output C matrix */
24   setMatrixValue(valC, C, ...);
25 }
```

IV.3 Benefits of concurrent kernel execution

This first experimentation wants to demonstrate that we can cover up the computation time of a specific kernel by executing a bigger kernel.

We made this experimentation in two different modes: synchronously (sequential) and asynchronously (parallel). For each experiments we start two kernels: the first with a fixed number of iterations (1024 here), and the second with a variable number of iterations (from 1 to 2048).

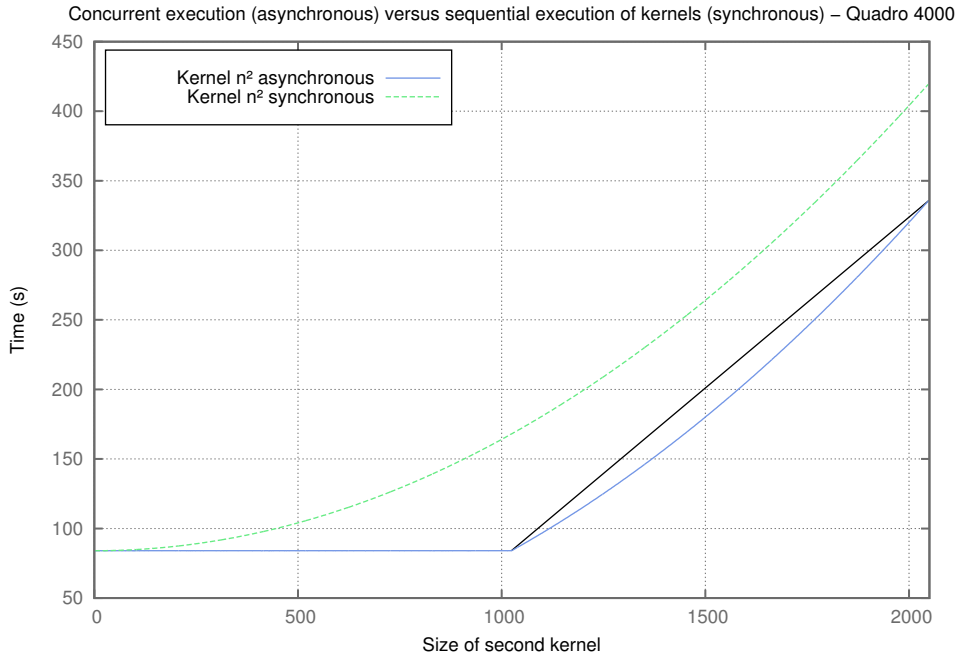


Figure 9: Concurrent execution versus sequential execution of kernels n^2

The blue curve shows the concurrent kernel execution. We can see that when the second kernel is executed with a number of iterations inferior to **1024**, the total time of execution (of both kernels) is limited by the execution time of the kernel with a fixed number of iterations. When the second kernel goes over 1024 iterations, then we can note that the total execution time increase. Thus, the computation time of the second kernel is hidden for a number of iterations inferior to 1024.

The green curve corresponds to the total execution time of the two kernels synchronously. We can see that the execution time increases with the number of iterations of the second kernel.

In this experimentation, we want to show that we can cover up the cost of a transfer from the host to the GPU. We launch concurrent execution of two different kernels (**kern_n2** and **kern_mem**).

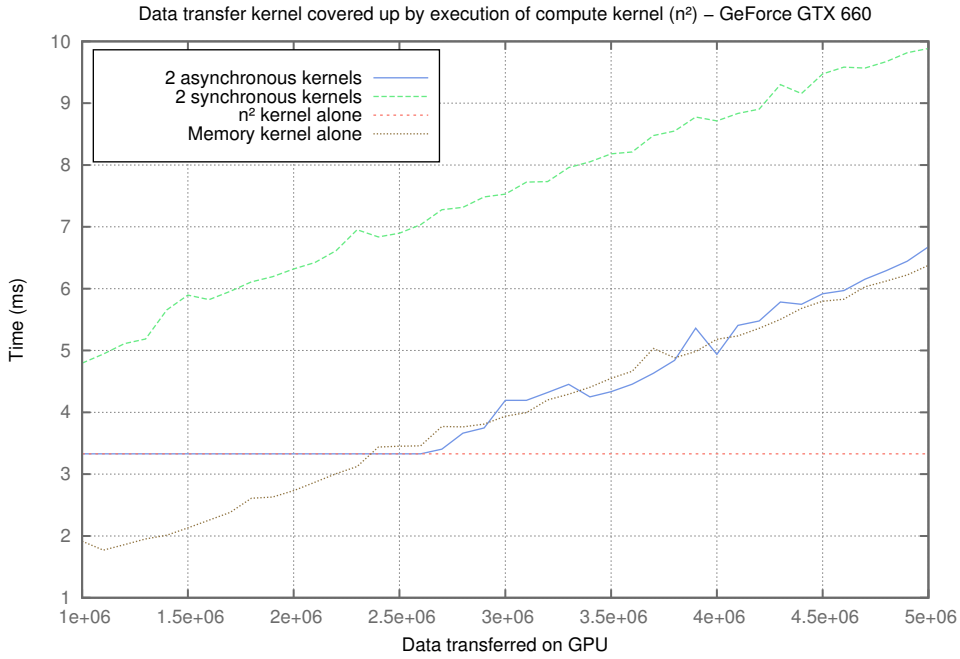


Figure 10: Data transfer kernel covered up by execution of compute kernel n^2

The red curve (**kern_n2**) corresponds to our reference curve: the execution time is steady because of two main reasons. Firstly, the number of executions is constant, and secondly `kern_n2` doesn't depend on transferred data. The brown curve corresponds to our **kern_mem** execution: the execution time increase with the transfer size.

The blue curve corresponds to the execution of both kernels asynchronously. We can see that when the time of the transfer does not exceed the computation time of **kern_n2**, data can be transfered freely.

For the last curve (in green), we execute both kernels synchronously. We can see that the total execution time is equal to the computation time of our **kern_n2** plus the transfer time of our **kern_mem**.

With these first experimentations, we have demonstrated that we can execute, transfer data and cover up the execution of a kernel by a bigger kernel.

IV.4 Core occupation improvement by concurrent kernel

This second experimentation aims to evaluate the advantages of parallel execution of kernels. This is why we use our **`kern_mat_mult_perf`** in that it can operate on different parts of the initial matrix in an independent way.

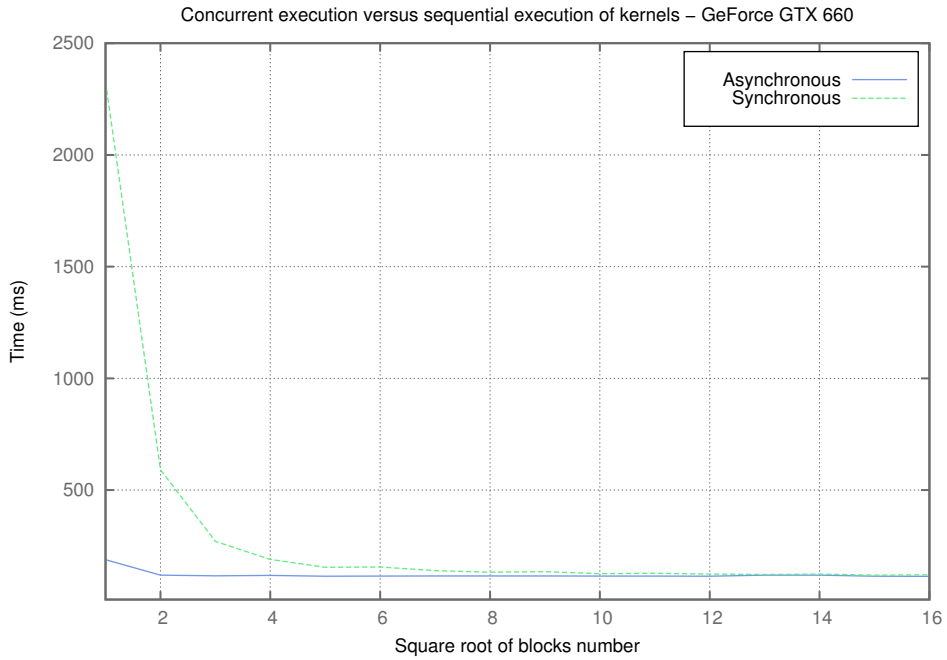


Figure 11: Concurrent vs sequential execution of kernels (`kern_mat_mult_perf`)

For the matrix multiplication, we divide the three whole matrix in blocks. Each group of block (from matrix A, B and C) is then associated to a kernel execution of **`kern_mat_mult_perf`**.

Results, presented in figure 11, shows that a parallel execution of all kernels instances is much more efficient than sequential execution. The difference between both types of execution falls down when the number of blocks increases, because of the number of threads allocated by each kernel execution: in each kernel execution, we set a fixed number of threads. This number of threads is the same for each execution of our kernel. Thus, when the number of blocks increases, the work size of each kernel execution decrease and the execution time too.

To be be more specific, we execute our program on a *GeForce GTX 660* which have 5 streaming multiprocessors (*SMX*). Be aware, in the figure 11, the number of blocks is a square root. When the number of blocks exceeds the number of *SMX*, the time of the synchronous version become closer of the asynchronous version.
However there is always a little speedup with asynchronous version until approximately 32 *CUDA* blocks.
Nevertheless, when we send a lot of kernels to GPU (asynchronous version), there is a little overhead (blocks one by one).

IV.5 About *CUDA 5.0* scheduling kernels

In this section, all figures are extracted from *Nvidia Visual Profiler 5* (*CUDA Toolkit 5*). We used the most up to date version of the software because it is the only one to provide visualisation of concurrent execution on GPU.

Case of matrix mutiplication by blocks (`kern_mat_mult_perf`)
Figure 12 shows a scheduling without concurrent mecanisms. Green bars are the execution of kernels. Each kernel executes the same code (SIMD code). Red bar show the profiling overhead. You have to consider the figure without this overhead.

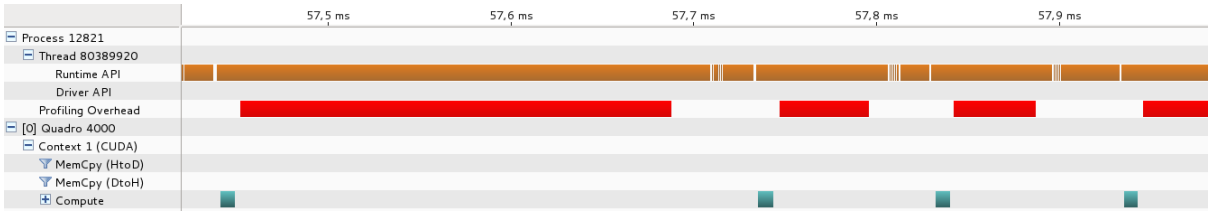


Figure 12: Sequential kernels (call of `kern_mat_mult_perf` 4 time)

Figure 13 exposes the benefit of using concurrent kernels. Let's specify that we used limited kernels. Each kernel doesn't exploit the full ressources in order to determine potential GPU concurrency.
In the figure 12 and 13, one kernel take one *CUDA* block and one *CUDA* block contain 256 threads.

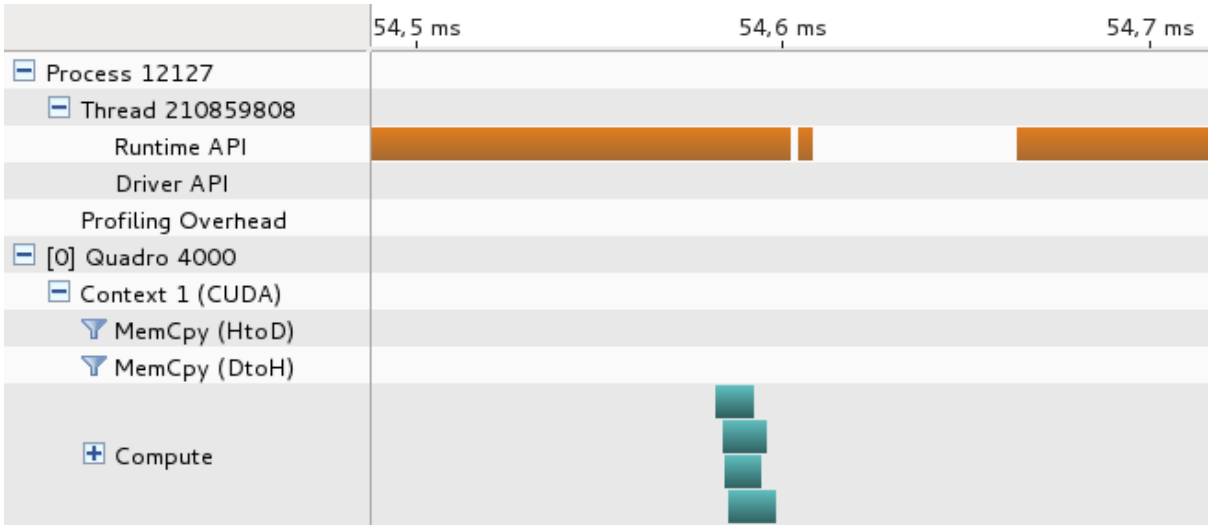


Figure 13: Concurrent kernels (call of `kern_mat_mult_perf` 4 time)

As expected, we notice a good level of parallelism with concurrent kernels (figure 13). But, we may also point out that kernels don't start at the same time. There is here a waste of time and a little limitation of concurrency on the GPU.

Case of concurrents `kern_mem` and `kern_n2`
For these measurement, we used two different kernels in order to represent a different use case (not strictly SIMD executions). The first kernel is `kern_mem` (purple bar and yellow bar) which does a little computation and an important transfer of memory (yellow bar). Unlike the previous kernel, the second kernel (`kern_n2`, green bar) only does a consequent computation.

We notice a visualisation problem with figure 14, 15, 16 and 17: we don't see transfer time between CPU memory and GPU memory (for `kern_mem`). However our tests (code part) demonstrate that there is a copy. We don't know if there is an *NVIDIA Visual Profiler 5* issue or a bug in our program.

Figure 14 shows sequential version: GPU execute `kern_n2` and `kern_mem` separately. We notice a little waste of time between the first and second execution. This is due to a return back to the CPU.

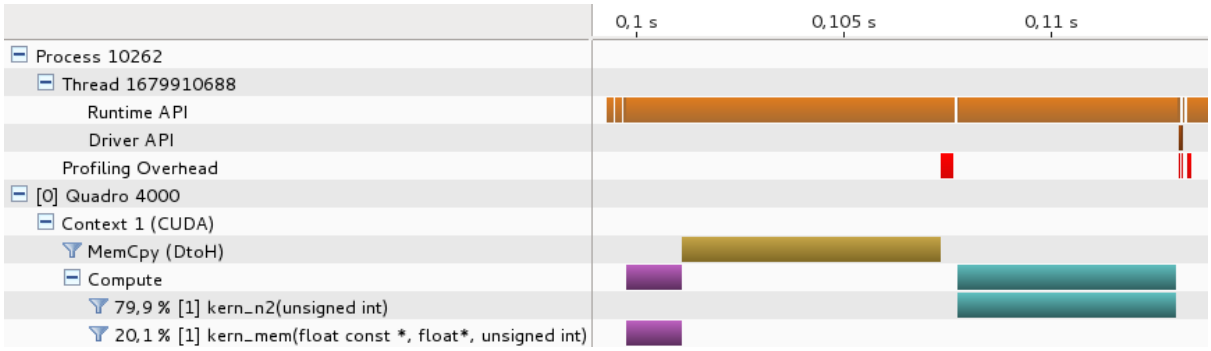


Figure 14: Sequential kernels (call of `kern_mem` before `kern_n2`)

In figure 15, we execute concurrently these two kernels. We see that the `kern_n2` execution start only when the execution part of `kern_mem` is over. Nevertheless, there is a parallelism because `kern_n2` is computing during `kern_mem` memory transfer.
Figure 15 was extract from a *Fermi* GPU (*Quadro 4000*).

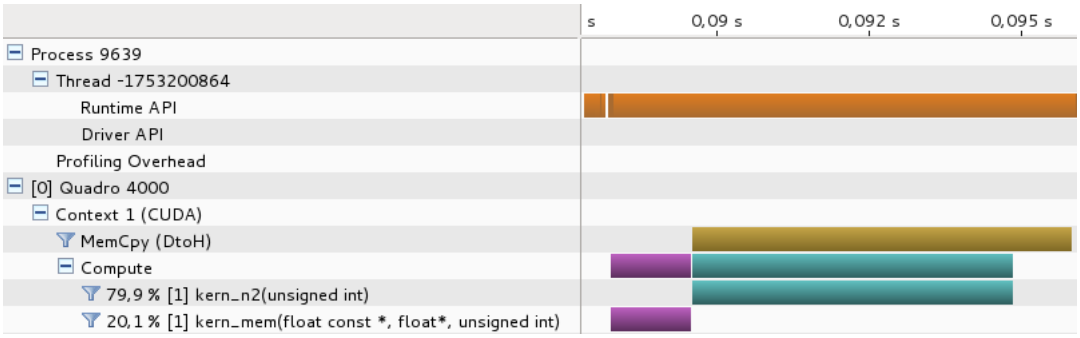


Figure 15: Concurrent kernels (call of `kern_mem` before `kern_n2`)

In figure 16, we use the same program as in figure 15 but on another GPU (*Geforce GTX 660*) with *Kepler* architecture. We don't notice the anomaly anymore. We conclude that the *Kepler* scheduler is better than the *Fermi* scheduler. We tried the program on another *Fermi* GPU (*Quadro 600*) with computation capability 2.1 and the problem stayed the same.

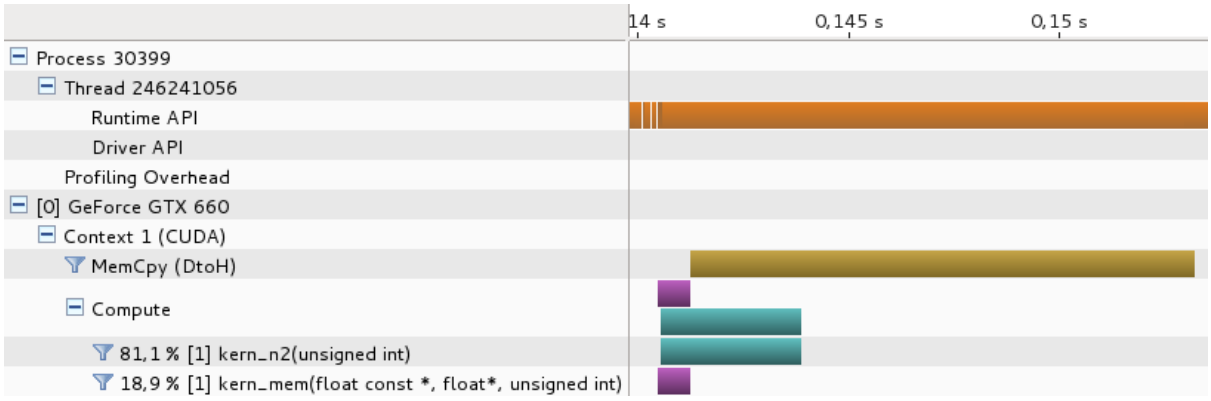


Figure 16: Concurrent kernels (call of `kern_mem` before `kern_n2`) on *Kepler*

We succeeded in by-passing *Fermi* problem with inter changing calls between `kern_mem` and `kern_n2` (we call `kern_n2` before `kern_mem`). With this solution, there is a perfect concurrency (figure 17).
We also notice that kernel execution order on device (GPU) follow the order of kernels calls on host (CPU).

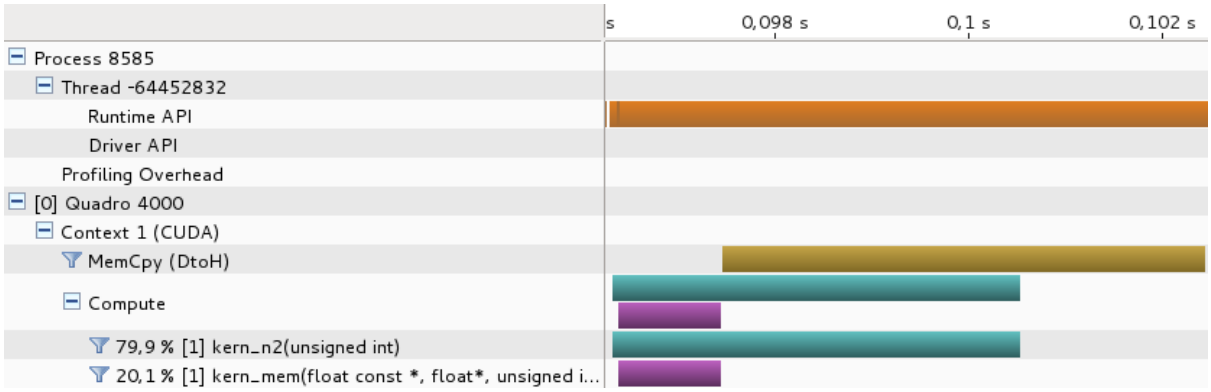


Figure 17: Concurrent kernels (call of `kern_n2` before `kern_mem`)

V Conclusion

According to the articles [2] [4] that we analysed, concurrency on GPU is a real issue. Today *NVIDIA Fermi* and *Kepler* GPUs are ahead in the race to Exascale.

Our tests demonstrate the capacity and the efficiency of CUDA and especially of concurrent kernels. Even if there are still some problems with concurrency on GPU (limitations on number of kernels, black-box scheduling, etc.) we are optimistic with this kind of programming.

Before the existence of a concurrency mechanism, GPU was too limited on particular situations. Today we can consider and design large range problems on GPU. Moreover, we see that some works begin to appear with *OpenCL* API. The solution brought by the article is less convincing than the *CUDA* approach but there is hope that *OpenCL* will evolve soon.

Even if we are very optimistic with kernels concurrency, we want to remind that a barrier on GPU is the programming itself. APIs are a too complex (two very distinct programming languages for a unique program) and we hope that this point will evolve in the future (*NVIDIA* has already done a lot of work by supporting a *C++* like language for kernels).

Anyway, we can't forget new alternatives on vector processing. For example, the *Xeon Phi*, provided by *Intel*, also brings up a lot of parallelism with the advantage of being fully x86/x64-compatible. Thus, actual parallel applications can be natively offloaded on *Intel Xeon Phi* processor (even if to reach performance, code-tunning is still needed).

Bibliography

[1] Khronos web site. http://www.khronos.org/.

[2] Lingyuan Wang, Miaoqing Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 24 –32, july 2011.

[3] Cuda web site. https://developer.nvidia.com/category/zone/cuda-zone.

[4] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.