# Studying the Fermi architecture to improve the Integer capabilities for cryptology

Clément Humbert, Tristan Overney
Paolo Ienne, Andrea Miele, Ewaida Mohsen (Supervisors)

January 9, 2015

# Contents

# List of Figures

# Chapter 1

# Introduction

Due to their computing power, GPUs have been used in different scientific fields to perform non-graphical computations for a while. They are used for particle and molecular dynamic modeling, astrophysics and fluid mechanic modeling. More recently they've been used to investigate the possibility of assessing the security of cryptographic applications, in particular RSA and elliptic curves encryption [10]. Cryptology-related applications make heavy use of very large integer operations. Here's a building block of the algorithms used [10]:

**Algorithm 1** Large operands multiplication

**Input:** Integers x and $Y = \sum_{i=0}^{n-1} Y_i r^i$ such that $0 \leq x, Y_i < r$ for $0 \leq i < n$.
**Output:** $Z = x \cdot Y = \sum_{i=0}^{n} Z_i r^i$

1: mul.lo($Z_0$, x, $Y_0$)
2: mul.hi($Z_1$, x, $Y_0$)
3: mad.lo.cc($Z_1$, x, $Y_1$, $Z_1$)
4: mul.hi($Z_2$, x, $Y_1$)
5: **for** $i = 2$ to $n - 2$ **do**
6: madc.lo.cc($Z_i$, x, $Y_i$, $Z_i$)
7: mul.hi($Z_{i+1}$, x, $Y_i$)
8: madc.lo.cc($Z_{n-1}$, x, $Y_{n-1}$, $Z_{n-1}$)
9: madc.hi($Z_n$, x, $Y_{n-1}$, 0)
10: **return** $Z(= \sum_{i=0}^{n} Z_i r^i)$

Due to the nature of asymmetric cryptography, these applications are bound by integer computing capacity which is not the focus of GPU. GPUs are designed around floating-point number operations, for older GPUs it was single-precision floating-point number (which will be referred to as float from now on) operations and on more recent GPUs tendency is to focus on double precision floating-point number operations. Fermi generation is the last generation to have good integer multiplication performances compared to its float performance. It is one of the reasons it was chosen for this project.

## 1.1 Goals

The aim of this project is to identify the limitations of Fermi GPUs when used for cryptology and applications using large integers arithmetic in order to suggest hardware changes accordingly.

# Chapter 2

# First approach

The first idea to enhance the cryptographic capabilities of the Fermi GPU was to build crypto specific hardware into the original hardware to expose new operations (in the form of assembly opcodes) to the developers with the shortest possible latency.

Montgomery multiplication, a special kind of modular multiplication, is an algorithm that is constantly used in cryptology; implementing it in hardware can be done in many ways. One of the most efficient way is the rolled implementation described by Algorithm 2. Its operands $X$ and $Y$ are already prepared in software for the Montgomery multiplication. Algorithm 2 should be unrolled for the hardware implementation thus allowing to pipeline it.

**Algorithm 2** Hardware Montgomery multiplier [3]

**Input:** Integers $X$, $Y$ and $M$ with $0 \leq X, Y < M$.
**Output:** $P = (X * Y(2^n)^{-1}) \bmod M$.
$n$ is the number of bits in $X$.
$x_i$ is the $i^{th}$ bit of $X$.
$s_0$ is the least significant bit of $S$.

1: $S := 0$; $C := 0$; ▷ $S$ is the Sum and $C$ is the carry
2: **for** $i = 0$; $i < n$; $i++$ **do**
3: $S, C := S + C + x_i * Y$; ▷ $S, C := ...$ is a carry-save adder assignation
4: $S, C := S + C + s_0 * M$;
5: $S := S$ div 2; $C := C$ div 2;
6: $P := S + C$;
7: **if** $P \geq M$ **then**
8: $P := P - M$;
9: **return** $P$;

Another way to enhance cryptology performances is to have part of such an algorithm implemented in hardware, such as multiplications of numbers larger than 32bits, and the rest in software.

The impact of the new opcodes were to be measured with the assistance of GPGPU-Sim [4] a Fermi architecture simulator. It was determined that new opcodes can be added to GPGPU-Sim and included in an instruction group to modify its latency (an unused operation could have been chosen to set the new instruction's latency).

To determine the speed/size of the new components, the original cores have to be studied, as not much is known from Nvidia sources. Due to the lack of information this first approach was

put aside and to understand the cores structure and be able to suggest meaningful changes, a series of microbenchmarking experiments had to be done.

# Chapter 3

# Fermi's architecture

Before diving into the experiments and their results here's a quick-start guide to Nvidia's Fermi architecture and its vocabulary:

## 3.1 Streaming Multiprocessors

The largest building block inside the Fermi architecture is the *Streaming multiprocessor* (Figure 3.1 on page 7) abbreviated SM in this report.

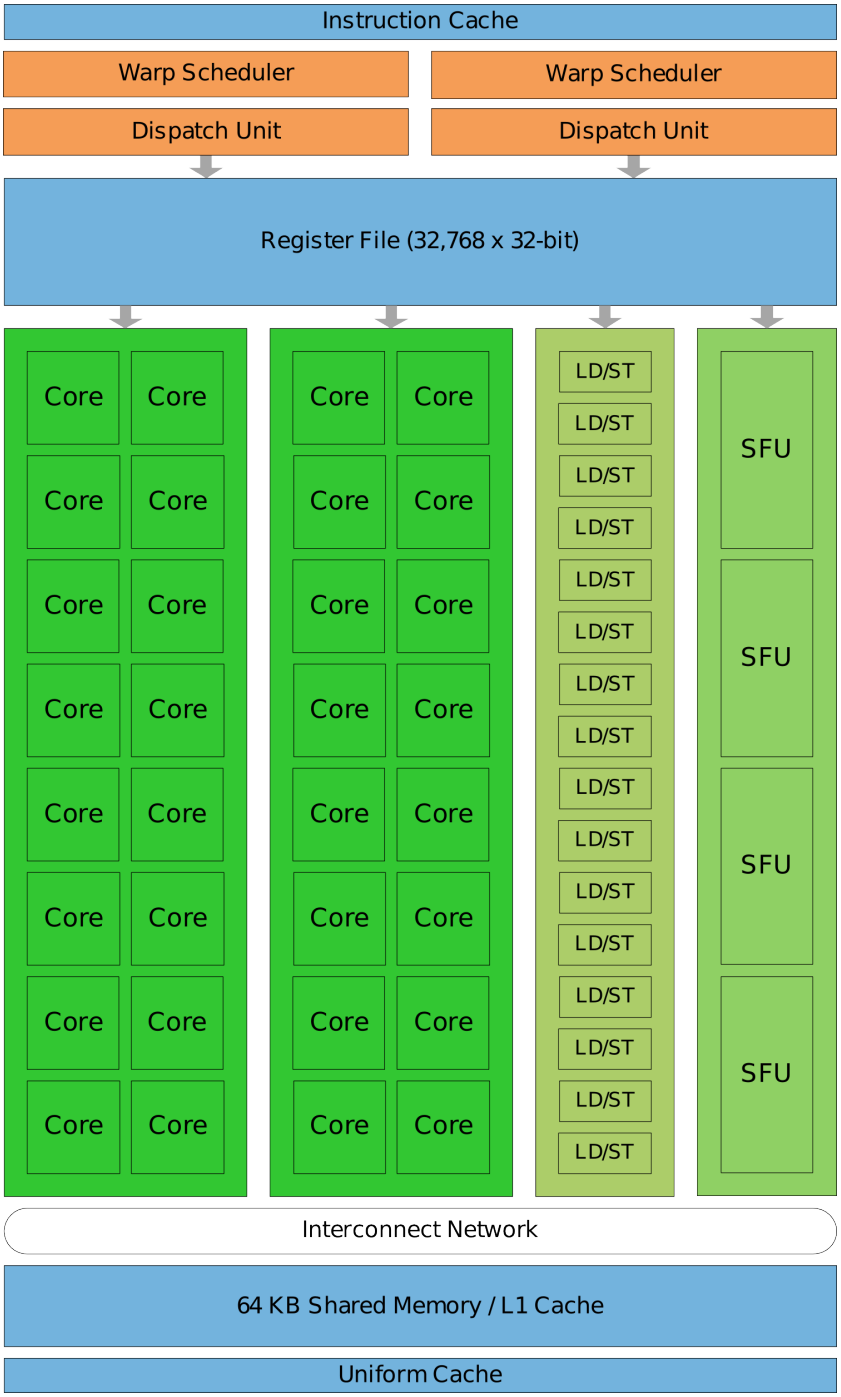


Figure 3.1: Fermi's SM schematic representation [6].

Each SM is composed of the following computation blocks [6]:

- 32 CUDA cores in two groups of 16,

- 16 load/store units (LD/ST on the figure),
- 4 Special Functions Units (SFUs) dedicated to more complex arithmetic functions such as sines and logarithms.

As the SFU is dedicated to complex float operations, it can be replaced by integer computation components, thus it does not require any further research in this project. The focus of the experiments are the CUDA cores.

## 3.2 CUDA Cores

Figure 3.2 represents a CUDA core according to Nvidia [6].

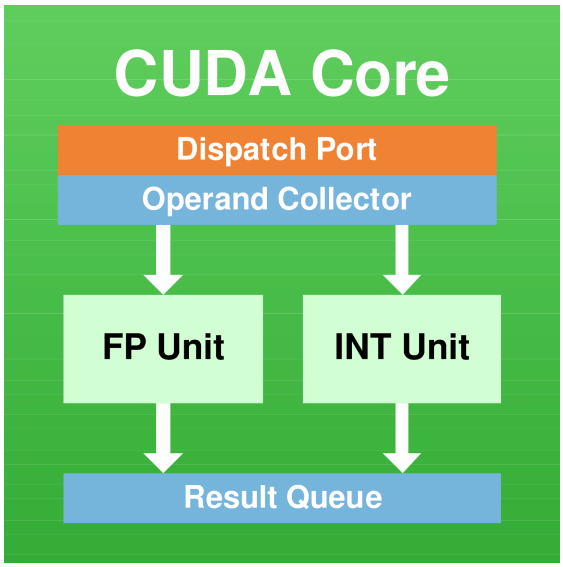


Figure 3.2: Schematic representation of Nvidia's CUDA Core.

From Figure 3.2 the notion of twice 16 cores can be adjusted to 2x16 integers ALUs and 2x16 float ALUs. Integer is supported with at most 32-bits in Fermi.

## 3.3 Basics of Fermi's scheduling

### 3.3.1 Grids and blocks

A CUDA kernel (i.e.: a section of code that will run on the GPU) is launched with two parameters, a grid size and a block size. The grid size is used to determine the number of SM on which the code will run. The block size determines how many threads will run the code inside each SM [8]. As the interesting properties are those of a single SM or even single core, every benchmarking experiment was done using a grid size of 1.

### 3.3.2 Warps

The second important scheduling unit is the warp. A warp consists of 32 threads. During each scheduling cycle, the scheduler selects two warps, for each of these it schedules 16 threads (making it 32 threads scheduled, coming from two warps) [8].

# Chapter 4

# Benchmarking

## 4.1 Methods

To get information about the microarchitecture of the CUDA cores, a series of specially crafted CUDA kernels were used. These usually contain large batches of instructions that were timed using the `clock64()` function offered by the CUDA API.

The benchmark programs have been run on a machine equipped with an Nvidia GeForce GTX 580 GPU used by Miele et al. [10].

## 4.2 Pipeline properties

This section contains the results obtained through the previously described methods using large batches of integer multiplications.

### 4.2.1 Benchmark running times against number of threads

#### Description of the experiment

The first experiment aims at outlining the relation between the running times of the benchmark program and the number of threads running in parallel in a single block (the threads reside on one SM).

#### Expectations

Integer multiplications overall performances are expected to be worse than float multiplication as indicated in [8]. The reason for it is what these experiments are trying to discover.

**Results and analysis**

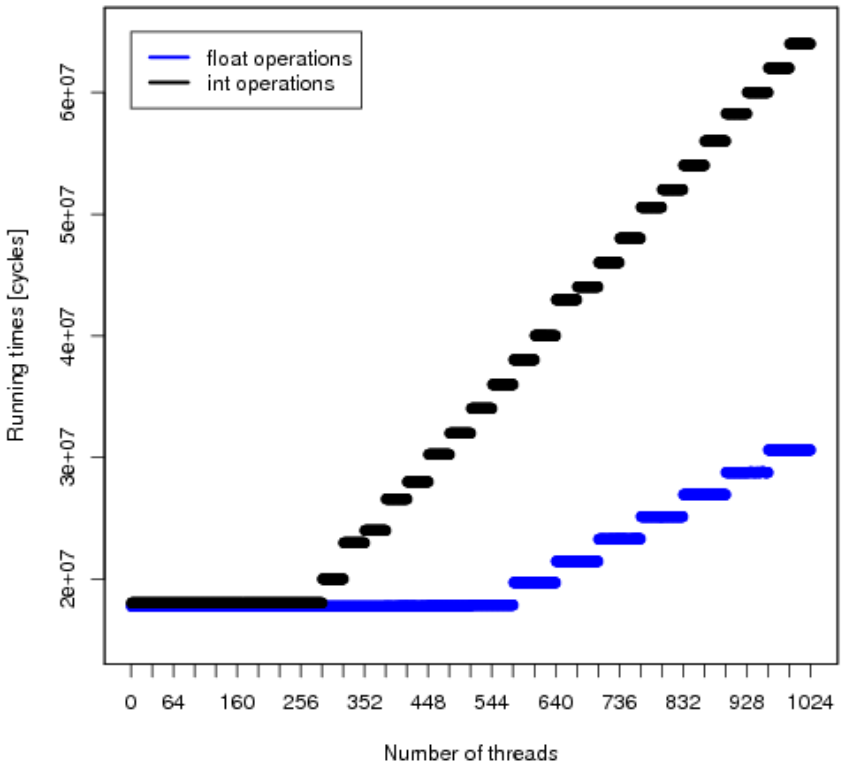


Figure 4.1: Running times of benchmark (in cycles) against number of threads.

Figure 4.1 depicts the results of the first experiment, and shows that the integer multiplication times start to grow at 289 threads in the block against 577 for the float multiplications.

The flat line of integer and float multiplication are superposed up to 288 threads, meaning that either both multiplication have the same latency or they're latencies are hidden by a pipeline. The flat line for the floating point pipeline would mean that up to 576 threads the pipeline is not filled. Since an SM has 32 cores, we compute the pipeline depth as $\frac{576}{32} = 18$ stages. This hypothesis seems to correlate with the running times of the integers multiplication.

Figure 4.2 represents the hypothesis made about the pipeline depth when 576 threads are running, $C_1$ to $C_{32}$ are the core of the single SM on which the experiment has been run. (1 to 16 is the first core group and 17 to 32 is the second one.) $t_{j,k}$ is the $k^{\text{th}}$ instruction of thread $j$.

| time since beginning | $C_1$ | $C_2$ | ... | $C_{16}$ | $C_{17}$ | ... | $C_{32}$ |
|---|---|---|---|---|---|---|---|
| 1 | $t_{0,0}$ | $t_{1,0}$ | ... | $t_{15,0}$ | $t_{32,0}$ | ... | $t_{47,0}$ |
| 2 | $t_{16,0}$ | $t_{17,0}$ | ... | $t_{31,0}$ | $t_{48,0}$ | ... | $t_{63,0}$ |
| 3 | $t_{64,0}$ | $t_{65,0}$ | ... | $t_{79,0}$ | $t_{96,0}$ | ... | $t_{111,0}$ |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 18 | $t_{528,0}$ | $t_{529,0}$ | ... | $t_{543,0}$ | $t_{560,0}$ | ... | $t_{575,0}$ |
| 19 | $t_{0,1}$ | $t_{1,1}$ | ... | $t_{15,1}$ | $t_{32,1}$ | ... | $t_{47,1}$ |
| ... | ... | ... | ... | ... | ... | ... | ... |

Figure 4.2: Hypothesized running scheme for 576 threads with float multiplication.

On the other hand, the flat line of the integer experiment ends on 288 threads. Assuming a similar pipeline depth for the integer unit (18 stages) then we conclude that $\frac{288}{18} = 16$ cores only support integer multiplication.

Figure 4.3 represents the hypothesis made about the pipeline when 288 threads are running, $C_1$ to $C_{32}$ are the core of the single SM on which the experiment has been run. (1 to 16 is the first core group and 17 to 32 is the second one.). The first core group is provided in integer multiplication but the second. $t_{j,k}$ is the $k^{\text{th}}$ instruction of thread $j$.

| time since beginning | $C_1$ | $C_2$ | ... | $C_{16}$ | $C_{17}$ | ... | $C_{32}$ |
|---|---|---|---|---|---|---|---|
| 1 | $t_{0,0}$ | $t_{1,0}$ | ... | $t_{15,0}$ | - | ... | - |
| 2 | $t_{16,0}$ | $t_{17,0}$ | ... | $t_{31,0}$ | - | ... | - |
| 3 | $t_{32,0}$ | $t_{33,0}$ | ... | $t_{47,0}$ | - | ... | - |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 18 | $t_{272,0}$ | $t_{273,0}$ | ... | $t_{287,0}$ | - | ... | - |
| 19 | $t_{0,1}$ | $t_{1,1}$ | ... | $t_{15,1}$ | - | ... | - |
| ... | ... | ... | ... | ... | ... | ... | ... |

Figure 4.3: Hypothesized running scheme for 288 threads with integer number multiplication.

To understand these deterioration points the next experiment has been designed.

## 4.3 Understanding pipeline length and performance deterioration

### 4.3.1 The experiment

With the results of 4.2.1 in mind, it's clear that the pipelines depth must be 18. The goal of the following experiment is to determine the cost of the loop used in the benchmarks to adjust the running times found and see if they match with the hypothesis about pipeline length and scheduling.

#### Expectations

As seen in 4.2.1 the cost of float operations goes up at 576 threads. This would imply that every pipeline of the SM is perfectly filled with 576 threads. Dividing 576 threads by the 32 cores gives

a pipeline length of 18. In addition, the cost increase is suspected to be $1/9^{\text{th}}$ of the base cost as the scheduling is expected to be something like described in Table 4.4.

| time since beginning | $C_1$ | $C_2$ | ... | $C_{16}$ | $C_{17}$ | ... | $C_{32}$ |
|---|---|---|---|---|---|---|---|
| 1 | $t_{0,0}$ | $t_{1,0}$ | ... | $t_{15,0}$ | $t_{32,0}$ | ... | $t_{47,0}$ |
| 2 | $t_{16,0}$ | $t_{17,0}$ | ... | $t_{31,0}$ | $t_{48,0}$ | ... | $t_{63,0}$ |
| 3 | $t_{64,0}$ | $t_{65,0}$ | ... | $t_{79,0}$ | $t_{96,0}$ | ... | $t_{111,0}$ |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 18 | $t_{528,0}$ | $t_{529,0}$ | ... | $t_{543,0}$ | $t_{560,0}$ | ... | $t_{575,0}$ |
| 19 | $t_{576,0}$ | - | ... | - | - | ... | - |
| 20 | - | - | ... | - | - | ... | - |
| 21 | $t_{0,1}$ | $t_{1,1}$ | ... | $t_{15,1}$ | $t_{32,1}$ | ... | $t_{47,1}$ |
| ... | ... | ... | ... | ... | ... | ... | ... |

Figure 4.4: Hypothesized running scheme for 577 threads with float multiplication.

The $\frac{1}{9}$ can be derived from the following formula

$$\frac{t_{577\text{threads}} - t_{576\text{threads}}}{t_{576\text{threads}}}.$$

The latency of an instruction is equal to the pipeline length, plus $\frac{1}{18}$ of it for every additional warp after the $18^{\text{th}}$. Thus, the latency $l$ of an operation is $L + \max\left(0, \left\lceil \frac{N}{32} - 18 \right\rceil\right)$ and to get the total running time of a benchmark, this latency must be added as a constant to represent the time the last instruction takes to get through the pipeline.

The expansion of the previous formula gives

$$\frac{(18 \cdot l_{576} + l_{577} + l_{577}) - (18 \cdot l_{576} + l_{576})}{18 \cdot l_{576} + l_{576}}.$$

This, with numerical values and simplifications is

$$\frac{(18 \cdot 19 + 2 \cdot 19) - (18^2 + 18)}{18^2 + 18} = \frac{1}{9}.$$

To make sure the previous computations are not biased by the loop cost that is measured, the million of operation was broken down into $10 \cdot 100,000$ operations and $1000 \cdot 1000$ operations. The difference would expose the cost of the for loop if it's significant.

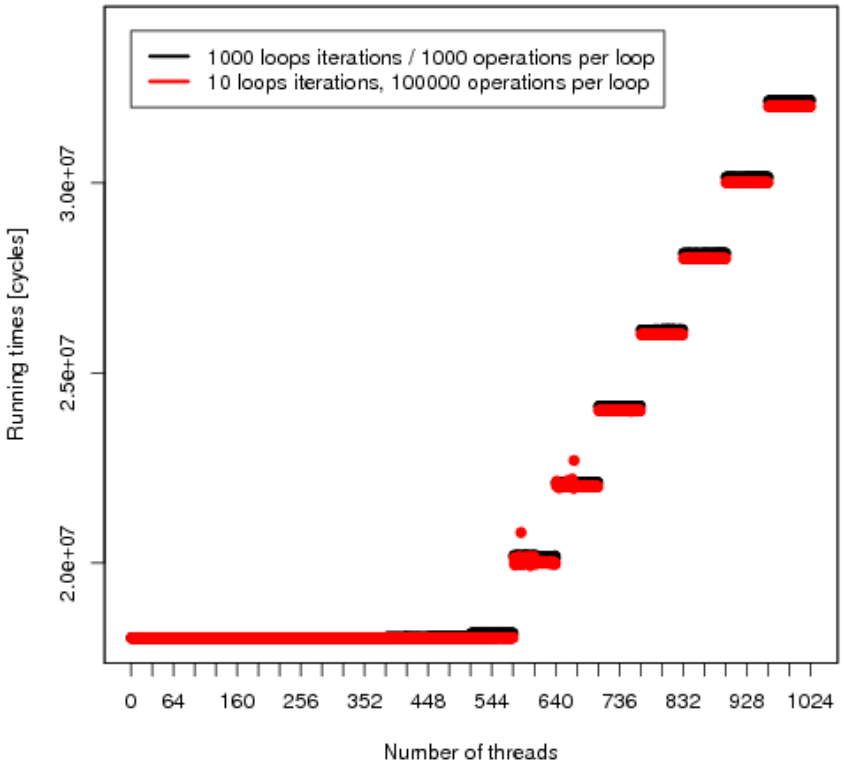


Figure 4.5: Running a million operations broken down in 10 and 1000 loop iterations.

Figure 4.5 suggests that only a insignificant amount of time is spent by the loop iterations' instructions.

## 4.4 Mixing float and integer multiplication

### 4.4.1 The experiment

The results of 4.2.1 seem to point out that the number of integer multiplications able to run in parallel on an SM is only half the number of float multiplications. Leading to the conclusion that only one of the two 16 core groups of an SM was equipped with integer. These experiments issue integer multiplications in parallel to float multiplications to confirm this hypothesis.

### 4.4.2 Benchmark running times, 1 float multiplication for 1 integer multiplication

If indeed only 1 out of 2 cores group can run integer multiplications, adding the same amount of float multiplications as there were integer multiplications should not increase the total time

spent executing the benchmark program as the float multiplications can be run on the other core group (the one that does not possess integer multipliers).

One million multiplications of each kind have been run on 1 to 1024 threads to see if the results were comparable to the integer multiplications only graph (Figure 4.1).

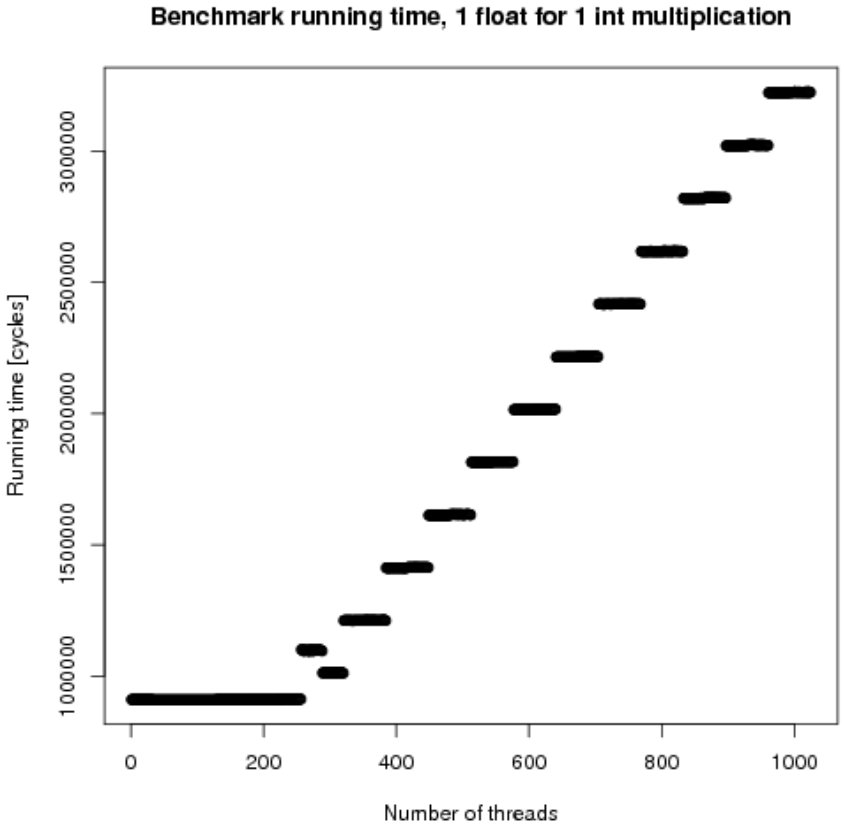


Figure 4.6: Integer/Float multiplication ratio: 1.

### 4.4.3 Benchmark running times with mixed float multiplications and integer multiplications

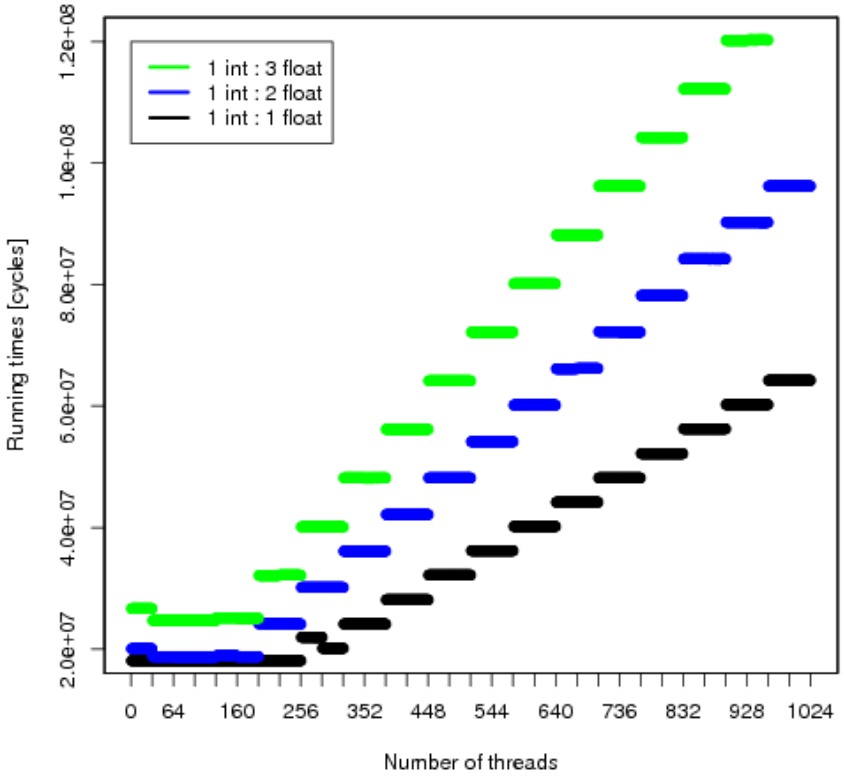


Figure 4.7: Running times of benchmarks with a mix of float multiplications and integers multiplications.

### 4.4.4 Results

The benchmark running times appear to be bound by those of the integer multiplications but is no higher than when only integer multiplications are run, this confirms the hypothesis that only 16 of the 32 CUDA core are equipped with integer multipliers.

## 4.5 Results

### 4.5.1 Pipeline structure

As seen during the experiments, the CUDA core's pipeline appears to be an 18 steps pipeline. The fact that integer multiplications and float multiplications both take the same amount of time (on a machine that's supposed to be a float computation optimized device) until the pipelines are filled suggests a simple, no-dependency-check, scheduler that fires up new instructions every 18 cycles.

It also appears rather clearly that the throughput for integer multiplications is only 16 multiplications per clock cycle per SM [8] due to the fact that, on the contrary of what Nvidia advertises, not all CUDA core (Figure 3.2) have a full integer unit and only 16 are equipped with integer multipliers; allowing only 16 integer multiplications to be scheduled every 18 cycles.

### 4.5.2 Prospects

From the previous observations the following ideas are expected to drastically improve the integer computation performance while maintaining a stable (if not lower) cost in transistors:

- Any new instructions (e.g. Montgomery multiplication, larger integer multiplication, etc.) can use up to 18 cycles without needing to modify any aspect of the scheduler.
- A large amount of integer computation power can be added at low-cost as a whole 16-cores group can be totally replaced by cores dedicated to integer arithmetic.

## 4.6 Instruction level parallelism

To check if the SM scheduler can detects dependencies, a benchmark with two streams of independent instructions has been run with an execution scheme as described by Figure 4.8. As there is no dependency between the even and odd instructions and there isn't any "new" thread to run, the scheduler looks for the next instruction of the threads belongings to the first two warps.

| time since beginning | $C_1$ | $C_2$ | ... | $C_{16}$ | $C_{17}$ | ... | $C_{32}$ |
|---|---|---|---|---|---|---|---|
| 1 | $t_{0,0}$ | $t_{1,0}$ | ... | $t_{15,0}$ | $t_{32,0}$ | ... | $t_{47,0}$ |
| 2 | $t_{16,0}$ | $t_{17,0}$ | ... | $t_{31,0}$ | $t_{48,0}$ | ... | $t_{63,0}$ |
| 3 | $t_{0,1}$ | $t_{1,1}$ | ... | $t_{15,1}$ | $t_{32,1}$ | ... | $t_{47,1}$ |
| 4 | $t_{16,1}$ | $t_{17,1}$ | ... | $t_{31,1}$ | $t_{48,1}$ | ... | $t_{63,1}$ |
| 5 | - | - | ... | - | - | ... | - |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 18 | - | - | ... | - | - | ... | - |
| 19 | $t_{0,2}$ | $t_{1,2}$ | ... | $t_{15,2}$ | $t_{32,2}$ | ... | $t_{47,2}$ |
| ... | ... | ... | ... | ... | ... | ... | ... |

Figure 4.8: Hypothesized running scheme for 64 threads with interleaved dependent float multiplication.

### 4.6.1 Results

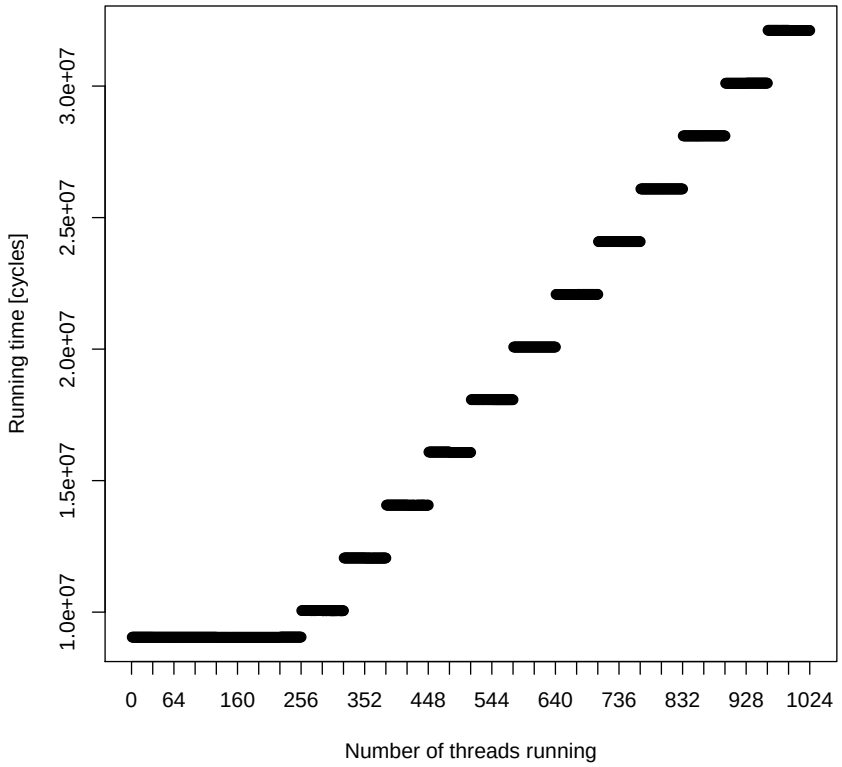


Figure 4.9: Running times of benchmarks with two independent instruction streams.

The running time for a million operations goes down to 9 million cycles in Figure 4.9 against 18 million without instruction level parallelism. This shows that the scheduler has the ability to detect and manage instructions dependencies to exploit the cores to the maximum.

# Chapter 5

# Beyond the current architecture

The results of those experiments give a much clearer picture of the internal functioning of a Fermi SM, and it is now possible to determine which changes can be done.

## 5.1 Proposed enhancement

The most obvious upgrade that can be implemented is to have two identical core groups in order to boost the issue rate of integer multiplications up to a float multiplication-like performance.

| $1^{st}$ CUDA core group | $2^{nd}$ CUDA core group |
|---|---|
| float unit and 32-bit multiplier | float unit |

Figure 5.1: Current CUDA core configuration.

There are two distinct approaches to this modification which are described as plan A and plan B in the following part of the report.

### 5.1.1 Plan A: 32-bit integer multiplier for every core

In this first scenario, the change would be to add a 32-bit multiplier to every core that doesn't have one yet. Consequently, all 32 cores would have a full float unit and a 32-bit multiplier.

| $1^{st}$ CUDA core group | $2^{nd}$ CUDA core group |
|---|---|
| float unit and 32-bit multiplier | float unit and 32-bit multiplier |

Figure 5.2: First plan CUDA core new configuration.

The expected outcome in term of issue rate would stay the same for float multiplications, that is 32 per clock cycle per SM. But the integer multiplication issue rate, which is assumed to have become the same as the float multiplication issue rate, is also expected to be 32 per clock cycle per SM. The advantage of this plan is that there is few changes.

### 5.1.2 Plan B: Every core with only integer support

In this second plan, instead of just adding 32-bit multiplier as in plan A; the change would be to have no more float support but instead have 32 integer only units with 32-bit multiplier.

| $1^{st}$ CUDA core group | $2^{nd}$ CUDA core group |
|---|---|
| 32-bit multiplier | 32-bit multiplier |

Figure 5.3: Second plan CUDA core new configuration.

The expected outcome for that scenario would then be an issue rate of 0 per clock cycle per SM as the support for such operand type would be removed, and the integer multiplication issue rate is expected to be close to the current float multiplication issue rate which is 32 per clock cycle per SM. The advantage of this plan is that the new SM will be smaller than in plan A.

## 5.2 Estimated performance increase

To estimate the increase in integer multiplications capability, GPGPU-Sim is used. The base configuration is the one for GTX480 shipped with GPGPU-Sim. With the default configuration, GPGPU-Sim simulate integer multiplication as any other operations (with two pipelines) but with a slower initiation rate. The first step to measure the performance is to simulate a one-pipeline multiplication with the following configuration changes in gpgpusim.config. Configuration options details can be found on the GPGPU-Sim wiki [2]:

| Option | Value |
|---|---|
| pipeline_widths | 1,1,1,1,1,1,1 |
| num_sp_units | 1 |
| ptx_opcode_initiation_int | 1,2,1,1,8 |
| operand_collector_num_in_ports_sp | 1 |
| operand_collector_num_out_ports_sp | 1 |

Figure 5.4: 16 multipliers simulation configuration.

To simulate the changes, the following configuration is used (Table 5.5 illustrates the changes from Table 5.4):

| Option | Value |
|---|---|
| pipeline_widths | 2,1,1,2,1,1,2 |
| num_sp_units | 2 |
| operand_collector_num_in_ports_sp | 2 |
| operand_collector_num_out_ports_sp | 2 |

Figure 5.5: 32 multipliers simulation configuration.

Launching the Algorithm 1 on both configurations yields the running times illustrated in 5.6.

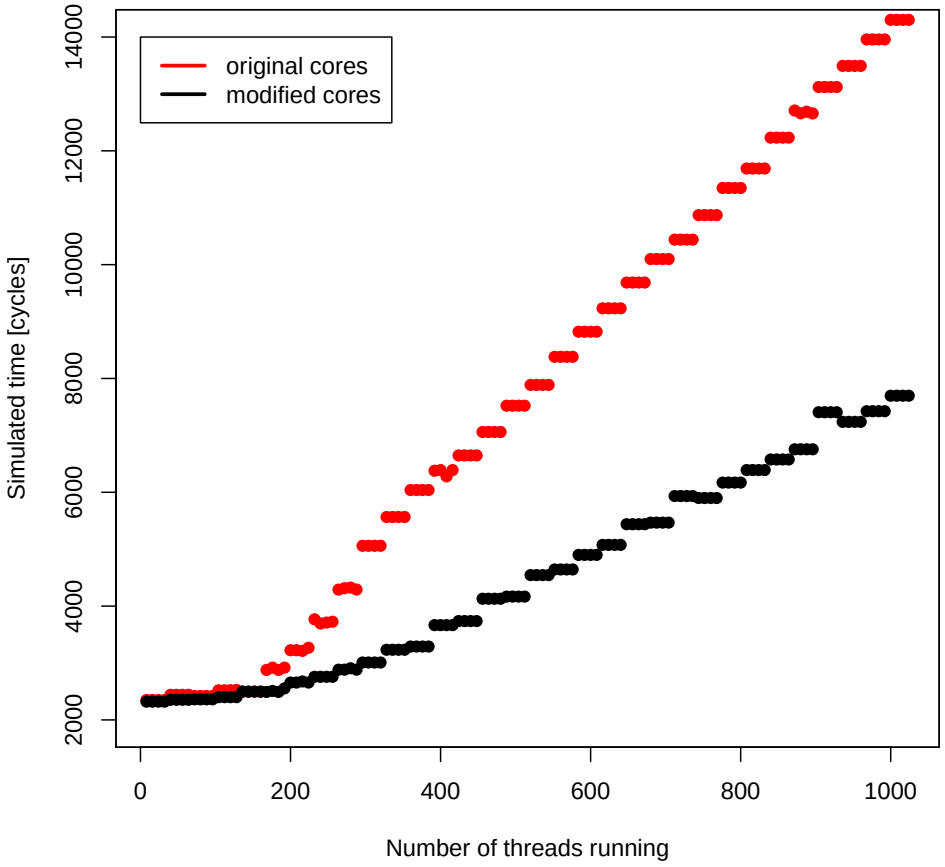


Figure 5.6: Running times on the default pipeline and on the improved architecture.

With higher occupancy, the improved architecture runs the large multiplication in 53% of the original time, an 85% improvement.

## 5.3 Estimated cost variance

In order to estimate the cost change, a 24-bit and a 32-bit multipliers have been synthesized with the $65nm$ library GPSVT, which will have to be adapted as the Fermi's architecture is manufactured in $40nm$ [7]:

$$\text{24-bit multiplier} = mul_{24} = 10,354\mu m^2 \text{ (in } 65nm) \Rightarrow 6,471\mu m^2 \text{ (in } 40nm).$$

$$\text{32-bit multiplier} = mul_{32} = 24,052\mu m^2 \text{ (in } 65nm) \Rightarrow 14,801\mu m^2 \text{ (in } 40nm).$$

The size of Fermi's component is also needed and for that a high resolution picture of the chip (Figure 5.7) is used in addition to the known total die size which is $520mm^2$ [1].

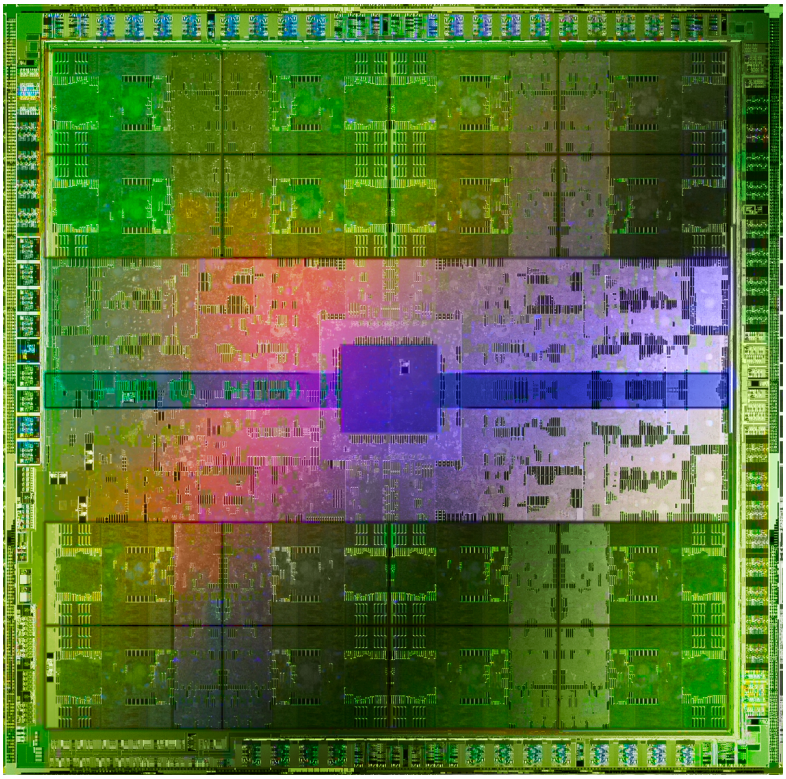

Figure 5.7: Picture of the full GTX580's chip.

### 5.3.1 Plan A: 32-bit integer multiplier for every core

First thing is to find out the size of a single CUDA core. To do so, the full chip picture (Figure 5.7) has been analyzed pixel by pixel with its original size of $1200p \times 1200p$ where Figure 5.8 brings out the different part of the chip that were identified and used. Then it was put in correspondance to the known $520mm^2$ area of the full chip and the Table 5.9 has then been filled.

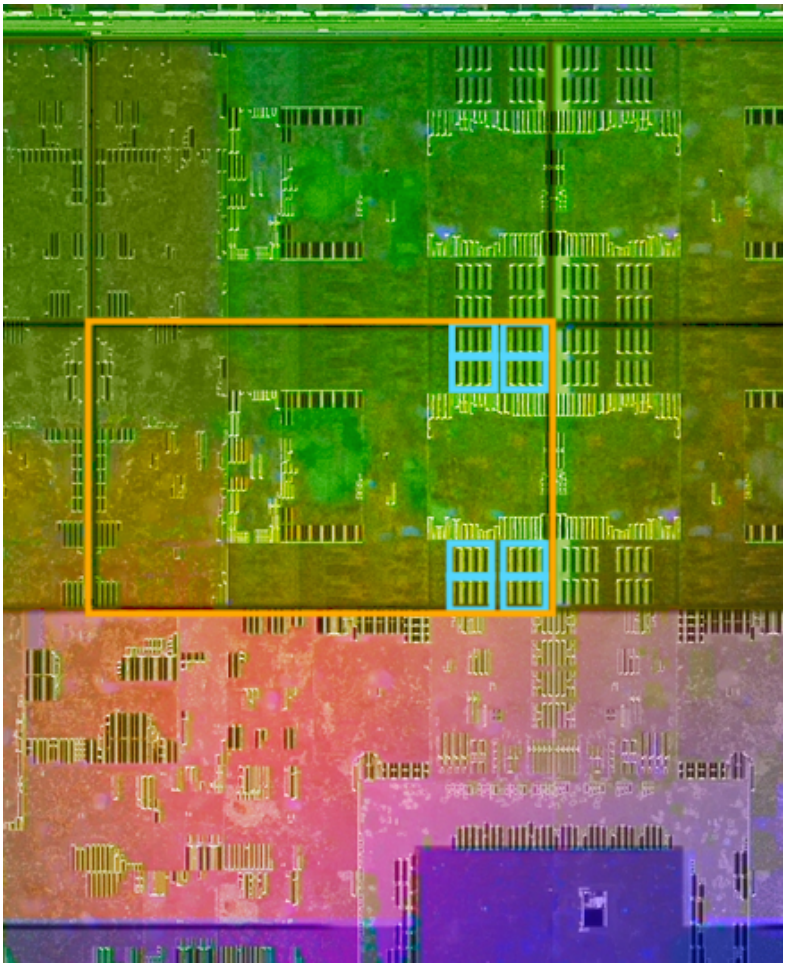

Figure 5.8: Inside the orange frame is an SM; inside the blue frame are the 32 CUDA cores of that SM.

| Component | Size in $\mu m^2$ | Size in $px^2$ |
|---|---|---|
| Full chip | $5.20 \times 10^8$ | 1,440,000 |
| Streaming multiprocessor (SM) | $1.50 \times 10^7$ | 41,600 |
| 32 CUDA cores | $9.71 \times 10^5$ | 2688 |
| Average CUDA core | $3.03 \times 10^4$ | 84 |

Figure 5.9: Size chart between pixels ($px$) and $\mu m^2$ of Fermi's elements.

The size, in the chart, for a single CUA core is an average of a "float unit and 32-bit multiplier" and a "float unit" CUDA core, in order to find the new core size only half a 32-bit multiplier has to be added.

$$\text{new core size} = \text{average CUDA core size} + \frac{mul_{32}}{2} = 3.77 \times 10^4 \mu m^2.$$

Thus, the changes would lead to a size increase of 24.4% of the average core size.

The new SM size can be computed by adding 16 times the 32-bit multiplier cost (once for each cores that doesn't have one already.)

$$\text{new SM size} = \text{SM size} + mul_{32} \times 16 = 1.53 \times 10^7 \mu m^2.$$

Which represents an increase of 1.5% of a SM size.

And eventually, the new size for the whole chip equals 16 times the difference between the old and the new SM size.

$$\text{new chip size} = \text{full chip size} + (\text{new SM size} - \text{old SM size}) \times 16 =$$

$$\text{full chip size} + (mul_{32} \times 16) \times 16 = 5.24 \times 10^8 \mu m^2.$$

Which represents an increase of 0.73% of the full chip size.

### 5.3.2 Plan B: Every core with only integer support

Here the task is to remove the float unit/ float and integer fused unit and replace it with an integer only unit.

In this section $u(\text{FP}_{24})$ represents the CUDA core size that possesses a float unit with only a 24-bit multiplier; $u(\text{FP}_{24}\text{INT}_{32})$ represents the CUDA core size that possesses a float and integer fused unit with a 32-bit multiplier; $u(\text{INT}_{32})$ represents the CUDA core size that only possesses a integer unit with a 32-bit multiplier. It is known that:

$$16u(\text{FP}_{24}) + 16u(\text{FP}_{24}\text{INT}_{32}) = 32 \text{ CUDA core size} = 9.71 \times 10^5 \mu m^2.$$

With the previous equation it is found that:

$$u(\text{FP}_{24}\text{INT}_{32}) = 6.07 \times 10^4 \mu m^2 - u(\text{FP}_{24}).$$

By an approximation it can be determined that:

$$\text{FP}_{24} = mul_{24} * 120\% = 0.78 \times 10^4 \mu m^2.$$

And then:

$$\text{FP}_{24}\text{INT}_{32} = \text{FP}_{24} - mul_{24} + mul_{32} = 1.61 \times 10^4 \mu m^2.$$

The size of the different CUDA cores should be relative to their units, it follows that:

$$\alpha = \frac{u(\text{FP}_{24})}{\text{FP}_{24}} = \frac{u(\text{FP}_{24}\text{INT}_{32})}{\text{FP}_{24}\text{INT}_{32}}.$$

As the value of $u(\text{FP}_{24}\text{INT}_{32})$ in function of $u(\text{FP}_{24})$ is known. It is found that:

$$u(\text{FP}_{24}) = 1.97 \times 10^4 \mu m^2 \text{ and, more interestingly : } \alpha = 2.52.$$

And finally, it is possible to find the size of the new CUDA core size for plan B:

$$\text{new CUDA core size} = u(\text{INT}_{32}) = \alpha \times \text{INT}_{32} = 3.73 \times 10^4 \mu m^2.$$

Which in this case would represent a 22.9% increase for the size of a CUDA core. The new SM size can be computed by removing the 32 old CUDA cores and putting 32 updated ones which are stripped from float support.

$$\text{new SM size} = \text{SM size} + 32 \times (u(\text{INT}_{32}) - \text{average CUDA core size}) = 1.52 \times 10^7 \mu m^2.$$

Which represents an increase of 1.48% of a SM size.

And eventually, the new size of the whole chip is 16 times the difference between the old and the new SM size.

$$\text{new chip size} = \text{full chip size} + (\text{new SM size} - \text{SM size}) \times 16 = 5.23 \times 10^8 \mu m^2.$$

Which represents an increase of 0.68% of the full chip size.

# Chapter 6

# Conclusion

In this project it has been made obvious that the integer performances (and a fortiori the crypto performances) of Nvidia GPUs are due to an intentional economy on integer hardware but that spares little area for a consequent drop in performances.

While GPU cards have a computation model well adapted to cryptology, it seems that Nvidia cares less and less about integer performances, the integer multiplication throughput compared to the number of cores dropped in Kepler compared to Fermi and is even translated as multiple instructions in Maxwell [8].

Nvidia's policy to keep everything closed source and secret made the original goal of the project (designing a crypto-specific architecture based on Nvidia GPUs architectures) impossible to attain and forced the realization of micro-benchmarking in order to find more information on the architecture we have chosen to base our work on. We went from an initial plan of designing our own many-cores crypto processor to trying to reverse-engineer the work of a full team of engineers.

# Appendix A

# Benchmarking program

This code sample is the typical code that was run for an experiment. There were variations on the operations inside the for loop (type, numbers of operations) and some experiments were also run without the for loop and just the operations.

```
#include <stdio.h>
#include <cuda.h>

__global__ void loop_mult(unsigned int* a, unsigned int* b, int64_t*
    times_bef, int64_t* times_aft, int n) {
    int u = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int op_a = a[u];
    unsigned int op_b = b[u];

    times_bef[u] = clock64();
    for(int i=0; i<1000; i++) {
        asm volatile("mul.lo.u32 %0, %0, %1;" : "+r"(op_a) : "r"(op_b
            ));
        /* 999 more multiplications */
    }
    timeis_aft[u] = clock64();

    a[u] = op_a;
}

int main(int argc, char** argv) {
    int n_threads = 1;
    int n_loops = 1;
    int n_grids = 1;

    if(argc == 2) {
        n_threads = atoi(argv[1]);
    }
    if(argc == 3) {
        n_threads = atoi(argv[1]);
        n_loops = atoi(argv[2]);
```

```
}
if(argc == 4) {
    n_threads = atoi(argv[1]);
    n_loops = atoi(argv[2]);
    n_grids = atoi(argv[3]);
}

dim3 dimBlock(n_threads, 1, 1);
dim3 dimGrid(n_grids, 1, 1);

unsigned int *a, *b;
unsigned int *d_a, *d_b;
int64_t *times_bef, *times_aft;
int64_t *d_times_bef, *d_times_aft;

a = (unsigned int *) malloc(n_threads*sizeof(unsigned int));
b = (unsigned int *) malloc(n_threads*sizeof(unsigned int));
times_bef = (int64_t *) malloc(n_threads*sizeof(int64_t));
times_aft = (int64_t *) malloc(n_threads*sizeof(int64_t));

for (int i = 0; i < n_threads; i++) {
    a[i] = (unsigned int) 3.1;
    b[i] = (unsigned int) 245321.0;
}

cudaMalloc ( &d_a,n_threads*sizeof(unsigned int));
cudaMalloc ( &d_b,n_threads*sizeof(unsigned int));
cudaMalloc ( &d_times_bef,n_threads*sizeof(int64_t));
cudaMalloc ( &d_times_aft,n_threads*sizeof(int64_t));

cudaMemcpy(d_a, a, n_threads*sizeof(unsigned int),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, n_threads*sizeof(unsigned int),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_times_bef, times_bef, n_threads*sizeof(int64_t),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_times_aft, times_aft, n_threads*sizeof(int64_t),
    cudaMemcpyHostToDevice);

loop_mult<<<dimGrid, dimBlock >>>(d_a, d_b, d_times_bef,
    d_times_aft, n_loops);

cudaMemcpy(a, d_a, n_threads*sizeof(unsigned int),
    cudaMemcpyDeviceToHost);
cudaMemcpy(times_bef, d_times_bef, n_threads*sizeof(int64_t),
    cudaMemcpyDeviceToHost);
cudaMemcpy(times_aft, d_times_aft, n_threads*sizeof(int64_t),
    cudaMemcpyDeviceToHost);
```

```
    for( int i=0; i < n_threads; i++) {
        fprintf(stderr, "%lu %lu\n",times_bef[i],times_aft[i]);
    }

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_times_bef);
    cudaFree(d_times_aft);
    free(a);
    free(b);
    free(times_aft);
    free(times_bef);
    return EXIT_SUCCESS;

}
```

# Appendix B

# Additional benchmarking data

## B.1 Graphics intersteps data

The following tables describe the steps between running times in the graphics presented previously. Analyzing them may allow to deduce properties of:

- the cores' pipelines, if it represents the delay between dependencies checks;
- the scheduling mechanism, if it represents the delaying of threads operations in favor of the launch of other threads.

| # | Time delta | Ratio of base execution time |
|---|---|---|
| 1 | 1,992,038 | 0.110518 |
| 2 | 2,972,214 | 0.164899 |
| 3 | 1,012,084 | 0.056151 |
| 4 | 2,577,818 | 0.143018 |
| 5 | 1,422,160 | 0.078902 |
| 6 | 2,256,334 | 0.125182 |
| 7 | 1,743,568 | 0.096733 |
| 8 | 2,016,076 | 0.111852 |
| 9 | 1,984,078 | 0.110077 |
| 10 | 2,024,116 | 0.112298 |
| 11 | 1,978,718 | 0.109779 |
| 12 | 2,943,966 | 0.163331 |
| 13 | 1,065,326 | 0.059104 |
| 14 | 2,011,174 | 0.111580 |
| 15 | 1,982,664 | 0.109998 |
| 16 | 2,537,828 | 0.140799 |
| 17 | 1,468,682 | 0.081483 |
| 18 | 2,005,218 | 0.111250 |
| 19 | 1,985,786 | 0.110172 |
| 20 | 2,256,680 | 0.125201 |
| 21 | 1,750,700 | 0.097129 |
| 22 | 2,007,560 | 0.111380 |
| 23 | 1,985,942 | 0.110180 |

Table B.1: Intersteps between integer multiplications benchmarking.

| # | Time delta | Ratio of base execution time |
|---|---|---|
| 1 | 2,147,856 | 0.119163 |
| 2 | 1,934,380 | 0.107320 |
| 3 | 1,997,060 | 0.110797 |
| 4 | 2,026,158 | 0.112411 |
| 5 | 2,009,622 | 0.111494 |
| 6 | 2,007,910 | 0.111399 |
| 7 | 2,018,118 | 0.111965 |

Table B.2: Intersteps between float multiplications benchmarking.

## B.2 Integer multiplication: 1024 threads starting times

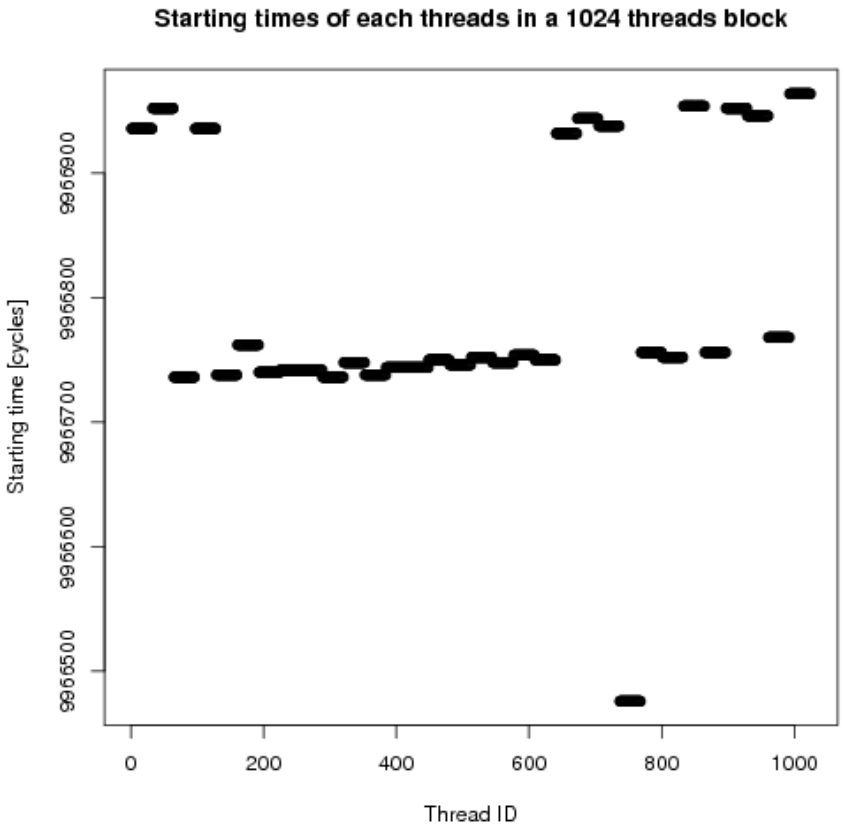


Figure B.1: Order in which thread are started.

Figure B.1 shows the starting times of threads by their id. It confirms the fact that the scheduler starts warps half by half (16 threads by 16 threads).

# Appendix C

# GPGPU-Sim faithfulness

To make sure GPGPU-Sim was usable with the benchmarks used in this project, the basic benchmarks were ran on the GTX 580 and on GPGPU-Sim with its default GTX 480 configuration (in which the only difference between the GTX 580 and GTX 480 is the number of SMs). The benchmark we have run are:

- Float multiplication benchmark;
- Integer multiplication benchmark;
- Float matrix multiplication;
- Integer matrix multiplication.

The running times yielded are plotted in Figure C.1.

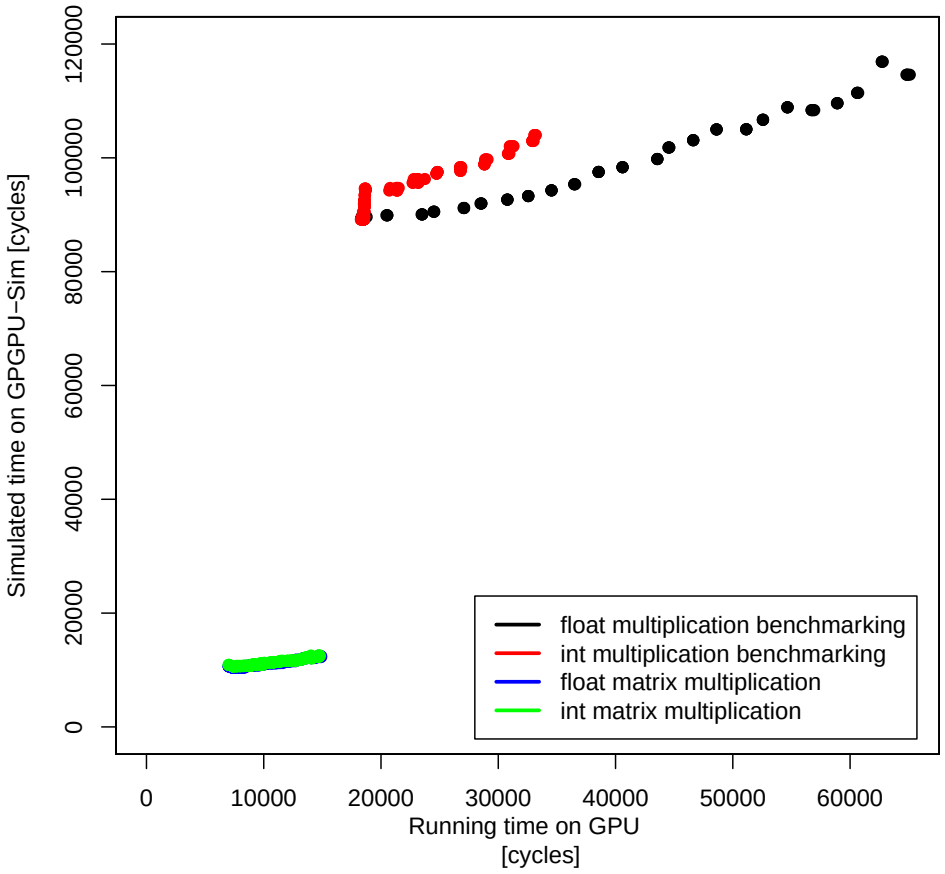


Figure C.1: Running times on the GTX 580 plotted against GPGPU-Sim running times.

The Pearson's correlation for each experiments are in Table C.2.

| Benchmark type | Running times correlation |
|---|---|
| Float multiplication benchmark | 94% |
| Integer multiplication benchmark | 98% |
| Float matrix multiplication | 98% |
| Integer matrix multiplication | 97% |

Figure C.2: Correlation between GPU running times and GPGPU-Sim simulated times.

While GPGPU-Sim has a higher clock count than the GTX 580 and its evolution is quite different, the running times are still strongly correlated.

# Bibliography


[1] GeForce 500 series. http://en.wikipedia.org/wiki/GeForce_500_series#GeForce_500_.285xx.29_series, 2009. Accessed: winter 2014.

[2] GPGPU-Sim. http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual, 2015.

[3] David Narh Amanor. Efficient Hardware Architectures for Modular Multiplication. 2005.

[4] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. 2009.

[5] Tom M. Bruintjes, Karel H. G. Walters, Sabih H. Gerez, Bert Molenkamp, and Gerard J. M. Smit. Sabrewing: A Lightweight Architecture for Combined Floating-Point and Integer Arithmetic. 2012.

[6] Nvidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2009.

[7] Nvidia Corporation. NVIDIA GeForce GTX 580 GPU Datasheet. 2010.

[8] Nvidia Corporation. CUDA C Programming Guide: Design Guide. 2014.

[9] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. 2009.

[10] Andrea Miele, Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra. Cofactorization on Graphics Processing Units. 2014.

[11] P. L. Montgomery and R. D. Silverman. An FFT extension to the p-1 factoring algorithm. 1990.