Benchmarking Fermi Microarchitecture

Paolo Ienne, Andrea Miele
Ewaida Moshen, Clément Humbert, Tristan Overney

November 20, 2014

1 Goals

The goals of this research is to expose the Fermi microarchitecture details as implemented in Nvidia Fermi cards (pipeline length, instructions latency, scheduling patterns, etc.) in order to know what can and cannot be changed to create an integer computation oriented device.

2 Methods

To achieve the aforementioned goals, a serie of specially crafted CUDA kernels were used. These usually contain large batches of instructions that were timed with the assistance of the `clock64()` function offered by the CUDA API.

The benchmark programs have been ran on a machine equipped with an Nvidia GeForce GTX 580 GPU.

3 Terminology

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 1 on page 1) abbreviated SM in this report. Fermi cards are equipped with 16 of these SMs.

Figure 1: Fermi's streaming multiprocessor schematic representation

Each SM is composed of the following computation blocks:

- 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 floating-point operations it's already an area that can be reused for integer computation components, thus it's not part of the following research. The focus of the experiments are the CUDA cores.

3.2 CUDA Cores

Figure 2 is the representation of what a CUDA core is, according to Nvidia.

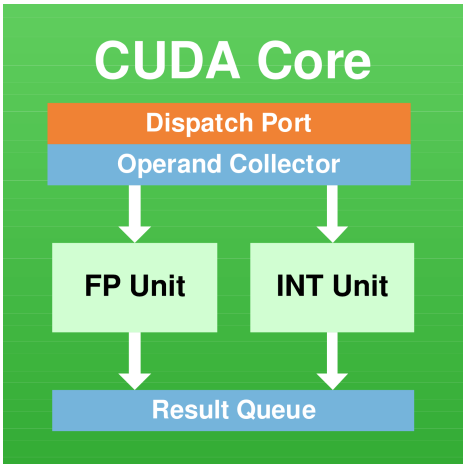

Figure 2: Schematic representation of Nvidia's CUDA Core.

From figure 2 the notion of twice 16 cores can be adjusted to 2x16 integers ALUs and 2x16 simple-precision floating-point ALUs.

3.3 Basics of Fermi's scheduling

3.3.1 Grids and blocks

A cuda kernel (that is: 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. As we're interested in the properties of single SMs or even, single cores, every benchmarking experiment was done using a grid size of one.

3.3.2 Warps

The second important scheduling unit is the warp. A warp consists of 32 threads. 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).

4 Pipeline properties

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

4.1 Benchmark running times against number of threads

4.1.1 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 parallely in a single block (the threads reside on one SM).

4.1.2 Expectations

The running times are expected to be slightly higher for the integer multiplications, as GPUs are optimized for floting point computation but to deteriorate in a similar fashion (begin to deteriorate at the same point, at the same rate) due to each core being equipped with integer and single-precision floating-point ALUs.

4.1.3 Results and analysis

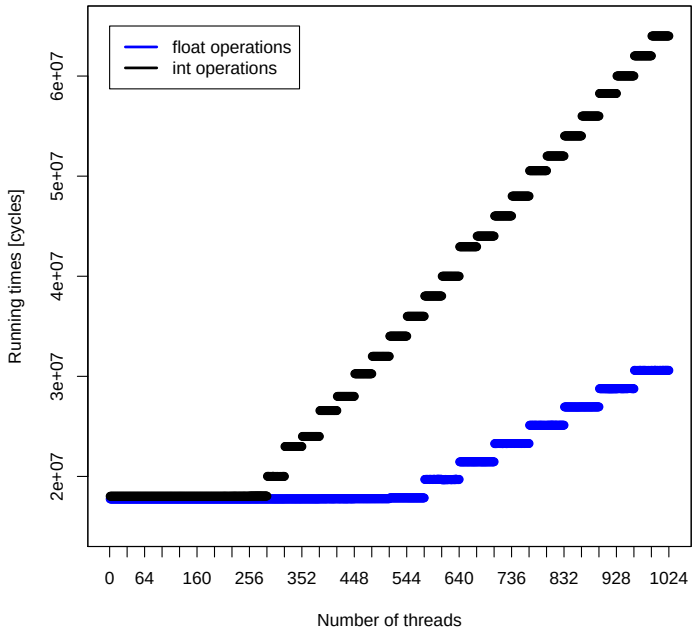

Figure 3: Running times of benchmark (in cycles) against number of threads.

The results are against the expectations as it's easy to see the integer multiplication times starting to grow at 257 threads in the block against 513 for the simple-precision floating-point multiplications.

As the running time is constant up to 512 threads running in parallel and that we have 32 cores available, the hypothesis has been formulated that the pipeline depth for the single-precision floating-point multiplication is 16 ($512/32 = 16$). Which seems to correlate with the running times of the single-precision floating-points multiplication.

Figure 4 represents the hypothesis made about the pipeline depth when 512 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-eme instruction of the thread number 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	$t_{64,0}$	$t_{65,0}$	...	$t_{79,0}$	$t_{96,0}$	...	$t_{111,0}$
...	...	...	...	...	...	...	...
16	$t_{464,0}$	$t_{465,0}$	...	$t_{479,0}$	$t_{496,0}$	...	$t_{511,0}$
17	$t_{0,1}$	$t_{1,1}$	...	$t_{15,1}$	$t_{32,1}$	...	$t_{47,1}$
...	...	...	...	...	...	...	...

Figure 4: Planned running scheme for 512 threads with floating point number multiplication.

Figure 5 represents the hypothesis made about the pipeline when 256 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 group core is provided in integer multiplication but the second. $t_{j,k}$ is the k-eme instruction of the thread number 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	$t_{32,0}$	$t_{33,0}$	...	$t_{47,0}$	-	...	-
...	...	...	...	...	...	...	...
16	$t_{240,0}$	$t_{241,0}$	...	$t_{255,0}$	-	...	-
17	$t_{0,1}$	$t_{1,1}$	...	$t_{15,1}$	-	...	-
...	...	...	...	...	...	...	...

Figure 5: Planned running scheme for 256 threads with integer number multiplication.

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

5 Understanding pipeline length and performance deterioration

5.1 The experiment

With the results of 4.1 in mind, it's clear that the pipelines depth must be 16. The objective 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.

5.1.1 Expectations

As seen in 4.1 the cost of simple-precision floating-point operations goes up at 512 threads. This would imply that every pipeline of the SM is perfectly filled with 512 threads. Dividing 512 threads by the 32 cores gives a pipeline length of 16. In addition, the cost increase is suspected to be 1/8 of the base cost as the scheduling is expected to be something like described in the following schematic:

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	$t_{64,0}$	$t_{65,0}$	...	$t_{79,0}$	$t_{96,0}$	...	$t_{111,0}$
...	...	...	...	...	...	...	...
16	$t_{464,0}$	$t_{465,0}$	...	$t_{479,0}$	$t_{496,0}$	...	$t_{511,0}$
17	$t_{512,0}$	-	...	-	-	...	-
18	-	-	...	-	-	...	-
19	$t_{0,1}$	$t_{1,1}$	...	$t_{15,1}$	$t_{32,1}$	...	$t_{47,1}$
...	...	...	...	...	...	...	...

Figure 6: Planned running scheme for 513 threads with floating point number multiplication.

The 1/8 can be derived from the following formula

$$\frac{t_{513\text{threads}} - t_{512\text{threads}}}{t_{512\text{threads}}} .$$

The latency of an instruction is equal to the pipeline length, plus 1/16 of it for every additional warp after the 16^{th}. So the latency l of an operation is $L + \max(0, \lceil \frac{N}{32} - 16 \rceil)$ 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

$$\frac{(16 \cdot l_{512} + l_{513} + l_{513}) - (16 \cdot l_{512} + l_{512})}{16 \cdot l_{512} + l_{512}} .$$

Which, with numerical values and simplifications is

$$\frac{(16 \cdot 17 + 2 \cdot 17) - (16^2 + 16)}{16^2 + 16} = \frac{1}{8} .$$

Thus, the outcome expected is that once the loop cost is removed from the times obtained in 4.1 the remaining time is 16 millions which is 16 cycles per operation, fitting the 16-stages pipeline hypothesis. The expected difference in running time after adjustments is 2 millions cycles, 1/8 of 16 millions.

5.1.2 Results and analysis

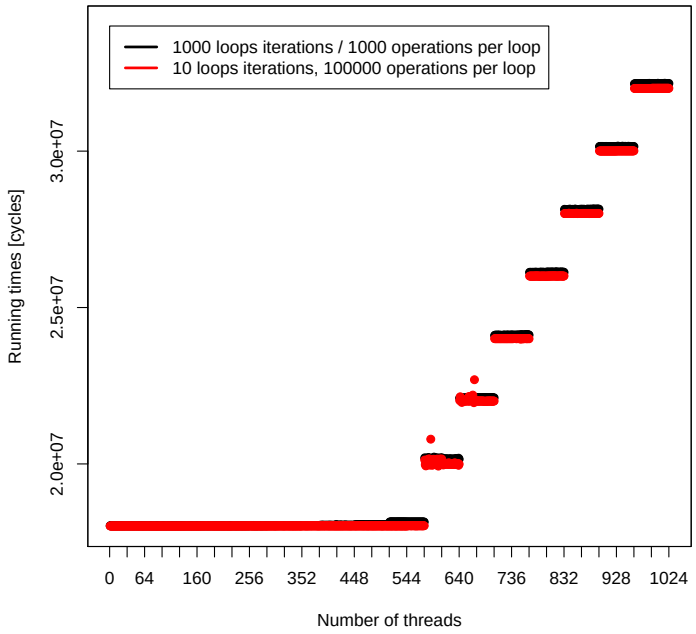

Figure 7: Running a million operations broken down in 10 and 1000 loop iterations.

Figure 7 suggests that only a insignificant amount of time is spent by the loop iterations' instructions. The next possible explanation would be data transfer overhead caused by compiler shenanigans.

6 Mixing single-precision floating-points and integer multiplication

6.1 The experiment

Information has been found in 4.1 that was implying that the number of integer multiplication able to run parallely on an SM was only half the number of single-precision floating-point multiplication. Leading to the conclusion that only one of the two 16 cores group of an SM was equipped with integer. The following experiments issue integer multiplication in parallel of floating-point operations to confirm this hypothesis.

6.2 Benchmark running times, 1 single-precision floating-points for 1 integer multiplication

If indeed only 1 out of 2 cores group can run integer multiplication then adding the same amount of floating-point multiplications as there were integer multiplications should not increase the total time spent executing the benchmark program as the single-precision floating-point multiplication can be ran on the other core group (the one that does not possess integer multiplication).

One million multiplication of each kind has been ran on 1 to 1024 threads to see if the results were comparable to the graph were there was only integer multiplication.

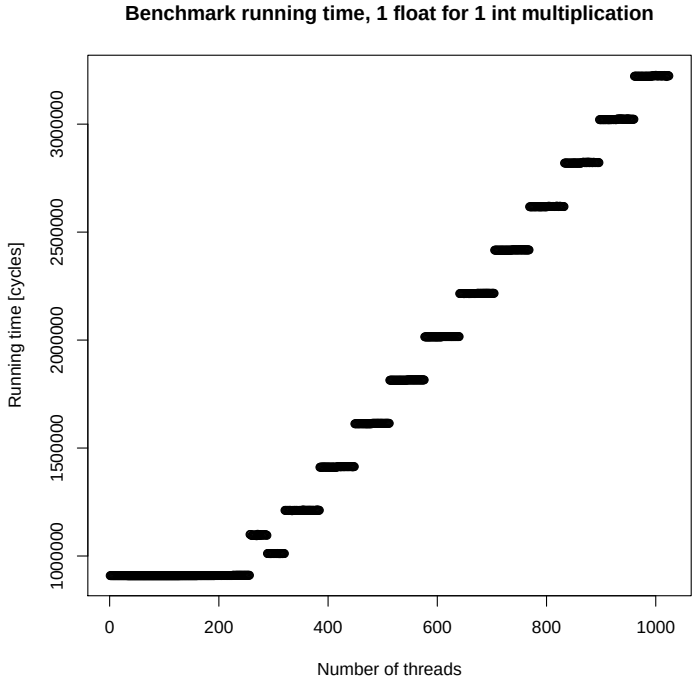

Figure 8: Integer/Floating point multiplication ratio: 1.

6.3 Benchmark running times with mixed single-precision floating-point and integer multiplications

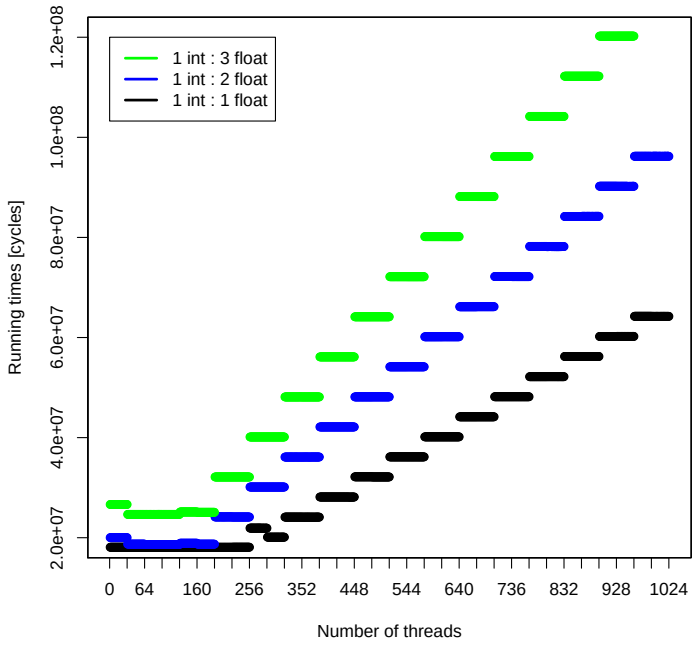

Figure 9: Running times of benchmarks with a mix of single-precision floating-points and integers multiplications.

6.4 Results

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

7 Results

7.1 Pipeline structure

As seen during the experiments, the CUDA core's pipeline appears to be a 16 steps pipeline. The fact that integer multiplication and simple-precision floating-point multiplication both take the same amount of time (on a machine that's supposed to be a floating-point calculation optimized device) until the pipelines are filled suggests a simple, no-dependency-check, scheduler that fires up new instructions every 16 cycles.

It also appears rather clearly that, while 32 cores per SM are advertised by Nvidia, only 16 are equipped with integer ALUs; allowing only 16 integer operations to be scheduled every 18 cycles.

7.2 Prospects

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

- If any instruction is to be added (e.g.: Montgomery's multiplication, larger integer multiplication) these can take up to 18 cycles without having 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.

8 Additionnal graphics and tables

[H]

8.1 Integer multiplication: 1024 threads starting times

8.2 Graphics intersteps data

The following tables describe the steps between running times in the graphics presented previously. Analysing 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.

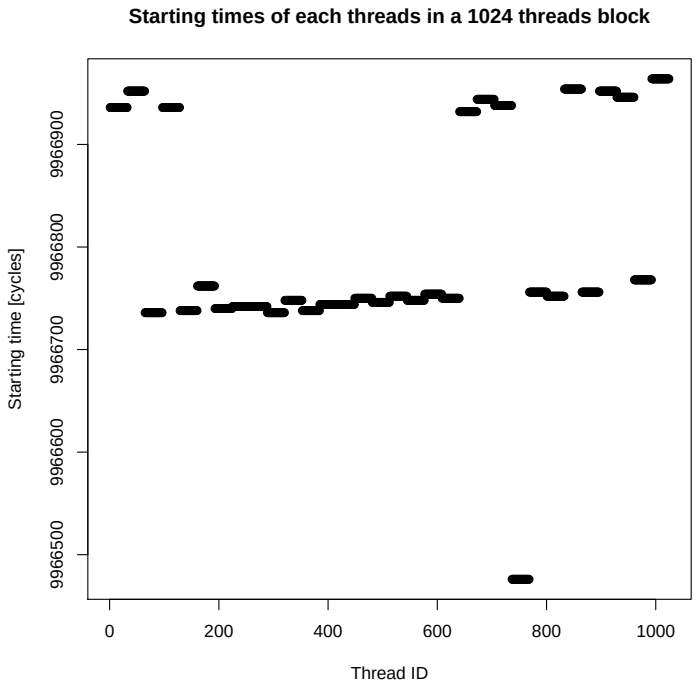

Figure 10: Order in which thread batches are started.

#	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 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 2: Intersteps between single-precision floating-point multiplications benchmarking.