Multithreading Benchmarks

Corky Maigre

May 20, 2016

Introduction

This project intended to compare the performance of a C/C++ console application using CPU & GPU programming. The console application consisted of doing a square matrix multiplication via several methods :

- CPU Programming
  - Sequential code
  - Multithreading code (one thread per cell of the result matrix)
  - Multithreading code (one thread per row of the result matrix)
  - Multithreading code (each cell of the result matrix is assigned to one of the n threads)
- GPU Programming

Each programming method was compared each other by doing a benchmark on the elapsed time for the computation only.

Information : - Homepage - GitHub - Website

Configuration

Hardware Specifications

Using an ASUS X93S Series laptop whose the charasteristics are listed below :

Charasteristics	Description
Processor	Intel Core i5 2430M 2.4 GHz ~ 2.9 GHz
Operating System	Windows 7 Home Premium
Chipset	Intel HM65 Express
Memory	DDR3 1333 MHz SDRAM, 4096 MB, (1 x 4096 MB)
Display	18.4" 16:9 Full HD (1920x1080) LED Backlight
Graphic	NVIDIA® GeForce® GT 540M with 1GB DDR3 VRAM
Storage	1 TB 7200 rpm
Optical Drive	DVD player
Card Reader	Card reader (SD/ SDHC/ MS/ MS Pro/ MMC)
Webcam	0.3 Mega Pixel Fixed web camera
Networking	Integrated 802.11 b/g/n, Bluetooth^{TM} V2.1+EDR,

Charasteristics	Description
	10/100/1000 Base T
Interface	1 x Microphone-in jack, 1 x Headphone-out jack,
	1 x VGA port / Mini D-sub 15 pins for external monitor,
	1 x USB 3.0 port, 3 x USB 2.0 ports, 1 x RJ45 LAN Jack
	for LAN insert, 1 x HDMI
Audio	Built-in Speakers And Microphone, SonicFocus,
	Altec Lansing® Speakers
Battery	6Cells : 5200 mAh 56 Whrs
Power Adapter	Output : 19 V DC, 6.3 A, 120 W Input : 100 -240 V AC,
	50/60 Hz universal
Dimensions	44.1 x 29.5 x 4.23 ~5.59 cm (WxDxH)
Weight	4.11 kg (with 6 cell battery)
Note	Master HDD: 3.5" SATA, Second HDD: 2.5" SATA

Table 1: Hardware specifications.

IDE Configuration

For this project, Visual Studio Community 2015 was used for CPU programming with the pthread library and Visual Studio 2013 was used for GPU programming with CUDA since CUDA is not supported in VC 2015.

> **CAUTION** Visual Studio Community 2015 does not support CUDA yet. You need to use an older version of VC such as Visual Studio 2012 & 2013.

First of all, you have to set that pthread library is used by typing 'pthreadVC2.lib' into the additional dependencies found at `Property` > `Links Editor` > `Additional Dependencies` as shown on the picture 1.

> **CAUTION** It is possible that you need to type `HAVE STRUCT TIMESPEC` into the preprocessor definition found at `Property` > C/C++ > `Preprocessor` > `Preprocessor Definition` as shown on the picture 2. Otherwise you will have this error message: Error C2011 'timespec' : redefinition of type 'struct'

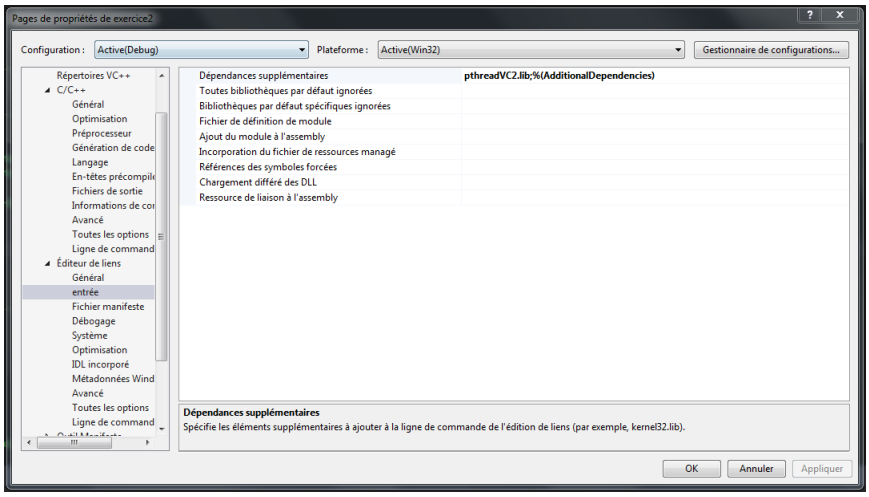

Figure 1: Using pthread.

Benchmarks

All benchmarks presented here are resulted from a specific use of threading such as the number of threads doing the computation.

Benchmark 1

This first benchmark is created by executing the C++ sequential code with CPU. The code uses dynamic arrays and a 'Matrix' structure. The elapsed time is determined by an average of three tests.

Table 2: Execution times for benchmark 1.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.000000 s
25 x 25	625	0.000149 s
50 x 50	2,500	0.001165 s
100 x 100	10,000	0.008968 s
200 x 200	40,000	0.071950 s
500 x 500	250,000	0.934330 s
1,000 x 1,000	1,000,000	15.01570 s
1,500 x 1,500	2,250,000	55.67037 s

Table 2: Execution times for benchmark 1.

Matrix Dimension	Number of cells	Average Elapsed Time
2,000 x 2,000	4,000,000	142.5680 s

Benchmark 2

The second benchmark is created by executing the parallel code with CPU in C++ using pthread. The code uses dynamic arrays and a dynamic number of threads. Each cell in the result matrix is performed by one thread. The elapsed time is determined by an average of three tests.

Table 3: Execution times for benchmark 2.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.001207 s
25 x 25	625	0.558716 s
50 x 50	2,500	1.841263 s
100 x 100	10,000	8.092927 s
200 x 200	40,000	33.41967 s
500 x 500	250,000	213.4013 s
1,000 x 1,000	1,000,000	936.1580 s
1,500 x 1,500	2,250,000	0.000000 s
2,000 x 2,000	4,000,000	0.000000 s

Before launching the console application for a square matrix of 1,000 x 1,000, the memory is low (but Visual Studio takes a big part of memory).

At the end of the console application for a square matrix of 1,000 x 1,000, the memory is full because all pointers take a lot of space and there are 1,000,000 of threads in memory. When the application finished, all the memory is released.

> **NOTE** The execution times for 1500x1500 and 2000x2000 matrice dimension has not been calculated since it takes more than 15 minutes.

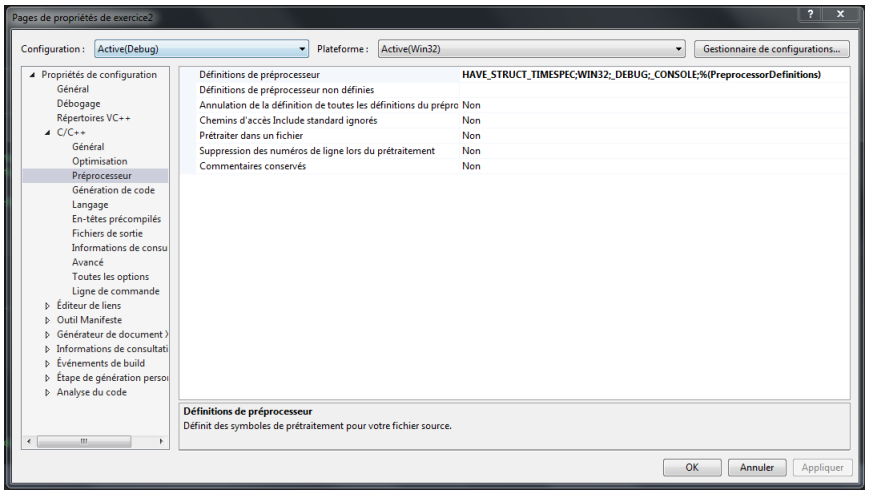

Figure 2: Timespec error.

Benchmark 3

The third benchmark is created by executing the parallel code with CPU in C++ using pthread. The code uses dynamic arrays and a dynamic number of threads. Each row in the result matrix is performed by one thread. The elapsed time is determined by an average of three tests.

Table 4: Execution times for benchmark 3.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.000451 s
25 x 25	625	0.012860 s
50 x 50	2,500	0.025046 s
100 x 100	10,000	0.057352 s
200 x 200	40,000	0.112013 s
500 x 500	250,000	1.108248 s
1,000 x 1,000	1,000,000	8.496977 s
1,500 x 1,500	2,250,000	26.31603 s
2,000 x 2,000	4,000,000	44.62573 s

Benchmark 4

The fourth benchmark is created by executing the parallel code with CPU in C++ using pthread. The code uses dynamic arrays and a dynamic number of threads. Cells in the result matrix are performed by one of the threads according to a shifting algorithm.

Number of threads: 2

The elapsed time is determined by an average of three tests.

Table 5: Execution times for benchmark 4 with 2 threads.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.002106 s
25 x 25	625	0.001467 s
50 x 50	2,500	0.003618 s
100 x 100	10,000	0.023646 s
200 x 200	40,000	0.051373 s
500 x 500	250,000	1.031829 s
1,000 x 1,000	1,000,000	10.90120 s
1,500 x 1,500	2,250,000	38.62227 s
2,000 x 2,000	4,000,000	101.9047 s

Number of threads: 3

The elapsed time is determined by an average of three tests.

Table 6: Execution times for benchmark 4 with 3 threads.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.011413 s
25 x 25	625	0.003701 s
50 x 50	2,500	0.0042903 s
100 x 100	10,000	0.009855 s
200 x 200	40,000	0.051281 s

Table 6: Execution times for benchmark 4 with 3 threads.

Matrix Dimension	Number of cells	Average Elapsed Time
500 x 500	250,000	0.733569 s
1,000 x 1,000	1,000,000	7.827880 s
1,500 x 1,500	2,250,000	30.25893 s
2,000 x 2,000	4,000,000	87.41533 s

Number of threads: 4

The elapsed time is determined by an average of three tests.

Table 7: Execution times for benchmark 4 with 4 threads.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.123637 s
25 x 25	625	0.002267 s
50 x 50	2,500	0.003300 s
100 x 100	10,000	0.027753 s
200 x 200	40,000	0.058286 s
500 x 500	250,000	0.668145 s
1,000 x 1,000	1,000,000	7.767977 s
1,500 x 1,500	2,250,000	25.13680 s
2,000 x 2,000	4,000,000	75.09370 s

Number of threads: 8

Table 8: Execution times for benchmark 4 with 8 threads.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.015070 s
25 x 25	625	0.031699 s
50 x 50	2,500	0.006153 s

Table 8: Execution times for benchmark 4 with 8 threads.

Matrix Dimension	Number of cells	Average Elapsed Time
100 x 100	10,000	0.013858 s
200 x 200	40,000	0.048016 s
500 x 500	250,000	0.651807 s
1,000 x 1,000	1,000,000	8.183037 s
1,500 x 1,500	2,250,000	30.89937 s
2,000 x 2,000	4,000,000	90.26363 s

NOTE Using the 'Cell' structure.

Benchmark 5

The fifth benchmark is created by executing the parallel code with GPU in CUDA C++. The code uses dynamic arrays and a dynamic number of threads. Each cell in the result matrix is performed by one thread. The elapsed time is determined by an average of three tests.

Table 9: Execution times for benchmark 5.

Matrix Dimension	Number of cells	Average Elapsed Time
2 x 2	4	0.000098 s
25 x 25	625	0.000110 s
50 x 50	2,500	0.000174 s
100 x 100	10,000	0.000537 s
200 x 200	40,000	0.004506 s
500 x 500	250,000	0.066445 s
1,000 x 1,000	1,000,000	0.537397 s
1,500 x 1,500	2,250,000	1.851377 s
2,000 x 2,000	4,000,000	3.164927 s

Comparison

Consider the dimensions below for all benchmarks :

- 200 x 200
- 500 x 500
- 1,000 x 1,000
- 1,500 x 1,500
- 2,000 x 2,000

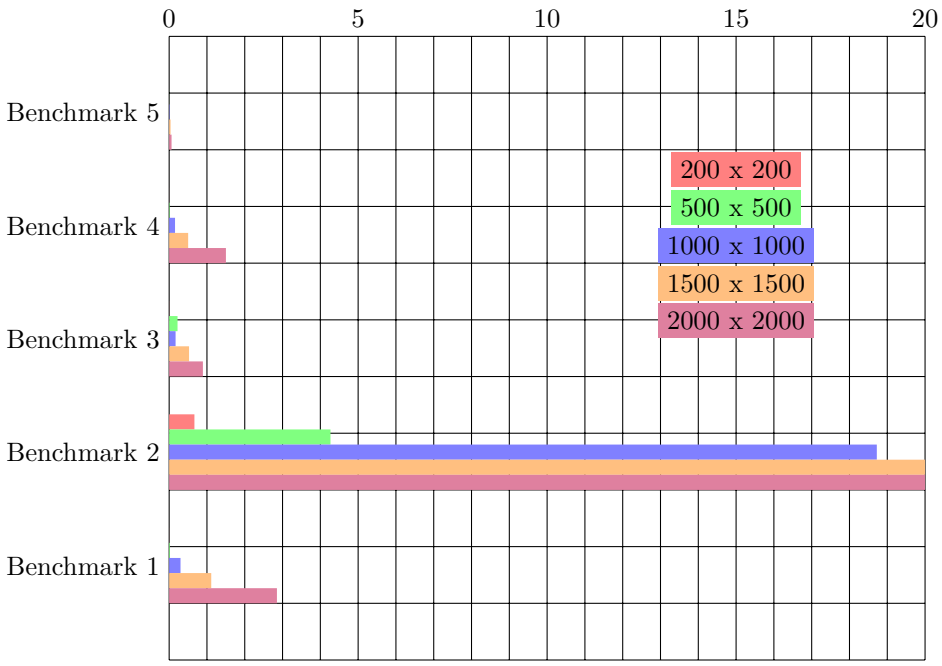

Figure 3: Execution time in milliseconds (scale: $\frac{1}{50}$)

Conclusion

In conclusion, threads have to be well assigned for a better performance. For instance, the execution of the code of benchmark 2 takes a long time because each cell of the result matrice is computed by an unique thread. Thus, there are a lot of threads ($n \times n$) and the operating system must open and close them via

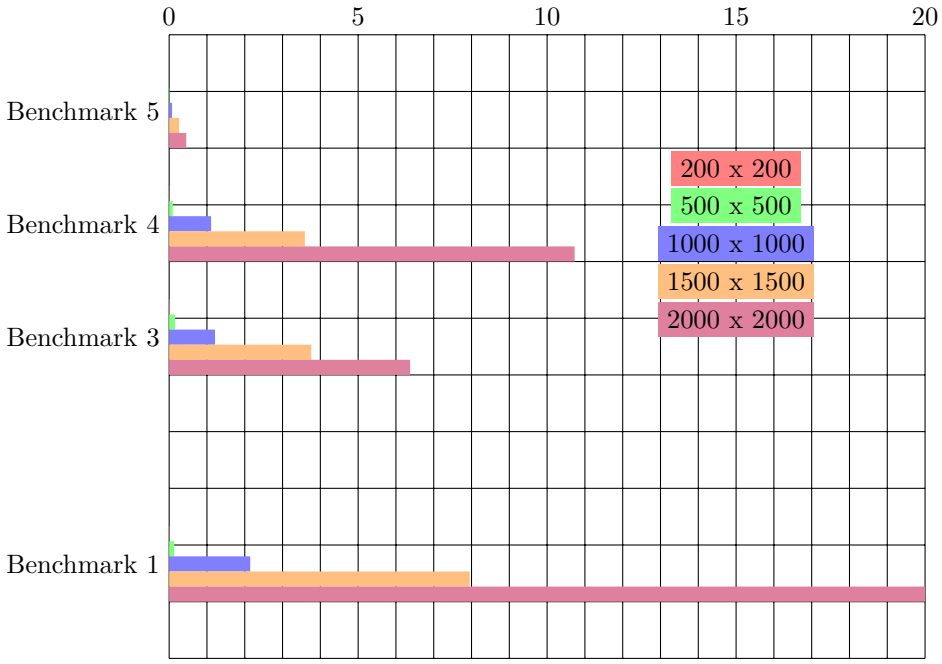

Figure 4: Execution time in milliseconds (scale: $\frac{1}{7}$)

a preemptive method since the number of threads is limited by the machine power. In this case, the execution of the sequential code (*i.e.*, no threads) of benchmark 1 is faster than the one of the code of benchmark 2.

Besides, when the matrices dimension are too small, the use of threads has not a big impact on the performance since only a little gain of performance has been reported.

However, when threads are well assigned, we can massively gain in performance. For example, the benchmark 3 is better than the two previous benchmarks and there are n threads since only one thread per row is created.

We can go further with benchmark 4 by assigning a static number of threads for the matrices computation. This allows to set the number of threads of the machine. For example, we tested 2, 3, 4, and 8 threads on benchmark 4, and the result is that using four threads gives the best performance. It is logic because the test machines has only four threads. The threads are alternately assigned to the result matrice cells via an custom algorithm.

Finally, GPU programming largely beats CPU programming on performance with very low execution times compared to the ones of CPU codes.

Bugs

- Error with the memory deleting:
  - detected the 01 May 2016
  - solved the 01 May 2016
- Error on benchmark 4 with matrix n x n where n $>$ 2
  - detected the 30 April 2016
  - solved the 19 May 2016 by using pthread_join() function.