A Comparative Study Of Calculation Performances Using Multi-Core CPU and GPU Parallel Computing Methods

D. Sentenac

VIR-0581A-10
Oct29,2010

Abstract

The standardization of CPU multi-core processing units in the last years has made parallel computing methods the main path to improve computing performances. The contemporary rapidly evolving graphical card GPU systems, offering cost-effective general purpose calculation possibilities, are a strong additional resource for parallel computing. In this note, we perform a desktop comparison study of computing performances between GPU and CPU multi-core processing units, using most recent computing methods.

I. Table of Content

II. Reference documents

[1] Parallel Computing: http://en.wikipedia.org/wiki/Parallel_computing
[2]CPUBenchmarking:http://www.intel.com/support/processors/xeon/sb/CS-020863.htm
[3] GPUBenchmarking: http://cuda-z.sourceforge.net/
[4] OpenMP: http://en.wikipedia.org/wiki/OpenMP
[5] CUDA: http://en.wikipedia.org/wiki/CUDA
[6] OpenCL : http://en.wikipedia.org/wiki/OpenCL
[7] CUDA Toolkit: http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html
[8] FFWT documentation: http://www.fftw.org/
[9] Matlab Help: http://www.mathworks.com/access/helpdesk/help/techdoc/
[10] GPUMat : http://gp-you.org/
[11] MAGMA: http://icl.cs.utk.edu/magma/overview/index.html
[12] BLAS: http://www.netlib.org/blas/
[13] CBLAS: www.netlib.org/clapack/cblas/
[14] ATLAS: http://www.netlib.org/atlas/
[15] GPGPU: http://gpgpu.org/

III. Acronyms and package meaning

CPU : Central Processing Unit
GPU : Graphical Processing Unit
GPGPU : General-purpose computing on graphics processing units
CUDA : Compute Unified Device Architecture
FFT : Fast Fourier Transform
Matlab : Technical computing software for engineers and scientists
MPI : Message Passing Interface

OpenMP : Open Multi-Processing
OpenCL : Open Computing Language
API : Application Programming Interface
GFLOP : Giga FLoating point Operations Per Second
Pthreads : POSIX threads library for C
POSIX : Portable Operating System Interface (for UNIX systems)
BLAS : Basic Linear Algebra Subprogram
ATLAS : Automatically Tuned Linear Algebra Software
CUBLAS : Implementation of BLAS on top of the NVIDIA CUDA
CTM : Close To Metal
MAGMA : Matrix Algebra For GPU AND Multicore Architectures

1. Introduction

Until the mid 2000, the continuous miniaturization of the semiconductor industry has allowed the computation power to linearly increase up to a few Ghz, corresponding to less than 100 nanometer size micro-processors. The last year's improvements in calculation performances are due principally to the parallelization of computing architectures in so called multi-core CPUs. Until recently, it was not easy to handle parallelization techniques which were restricted to super-computing clusters. Multi-core CPUs are now equipping the standard desktop computers. Recent compilators have integrated multi-core parallelization standard coding methods [1]. In parallel, graphics cards, originally designed for parallel computation in 3D graphics domain, have given rise to a growing interest to exploit their capabilities for general purpose calculations. Today, graphical and multi-core processor units are viewed as complementary resources to build up powerful modern super-computers. In this note we give an introduction on CPU multi-core and GPU from the point of view of coding techniques and existing libraries. In the first section we introduce what is becoming the standard code method for multi-core parallelization on CPUs and GPUs. In the second section we provide a performance comparative study using different available high level open source libraries. In particular we will show some practical code examples for matrix and FFT 1D calculations.

2. Parallel Computing Techniques

There are two main paths to parallelization: CPU and GPU. GPU are traditionally small multi-core processors. They have been originally designed to parallelize big matrix calculation for 3D graphical applications. To increase CPU computing power, the idea has been to expand the CPU architecture to a multi-core one. In this section, we introduce some recent promising coding techniques which are becoming a standard in multi-core parallel code development.

2.1 Multi-Core CPU

There are several ways to benefit from a multi-core processor CPU. The POSIX threads (Pthreads) is maybe the most known by c programmers. More recently, compilators starting from gcc 4.1 integrate a new option to use another shared-memory based API: OpenMP [4]. There are arguments to be inclined to favor the use of OpenMP instead of Pthreads:

1) Easier way to migrate serial to parallelized code
2) Overall simplification of the development process (debugging, tuning)

Using OpenMP doesn't require rewriting an application. Only the specific areas of code involving calculation loops need to be rearranged. OpenMP provides a set of pragmas, runtime routines, and environment variables that programmers can use to specify shared-memory parallelism. When OpenMP pragmas are used in a program, they direct an OpenMP aware compiler to generate an executable that will run in parallel using multiple threads. To compile the parallelized code, we just need to add the option '*–fopenmp'* for gcc, to make it understand the pragma *omp parallel*. Note that an older gcc compiler that does not understand this pragma will simply ignore it. Also note that OpenMP has extensions for C++ and FORTRAN.

2.2 GPU

Graphic cards are the other path to parallelize an application over multi-cores. Most of 3D graphics computations involve basic matrix and vector operations. Recently, GPU have been increasingly studied also for general purpose calculations. The advantage over multi-core CPU is the number of cores available which is about 2 orders of magnitude above (see Table 1). GPUs of the most powerful class simply interface with the motherboard by means of an expansion slot such as PCI express (PCIe) and can power up a machine with hundreds of additional cores. We observe two major bottlenecks of GPUs respect to CPU in the literature:

1) The data transfer latencies introduced by CPU and GPU communication through the PCIe bus, because the main program is always CPU based.
2) The limitation of GPU 32-bit processors having only single-precision data capabilities. However last generations of GPUs are able to handle double-precision floating-point data compliant with **IEEE 754** standard, but according to literature, the double-precision operations are less efficient compared to single-precision operations (see table 1).

Processor Type	Clock speed	# cores	GFLops 64	GFlops 32	Price
CPU Xeon W5590	3.33 GHz	4 (x2)	53.28	-	$1600
GPU Nvidia GTX 480	700MHz	480	168	1344.96	$500

Table 1: Main characteristics and performances comparison between last generation CPU and GPU processors taken from constructors' literature using Linpack benchmark for both CPU [2] and CUDA-Z for GPU [3].

One of the first development framework dedicated to 3D graphics applications for GPU is OpenGL. Nowadays, constructors like NVIDIA and ATI have released their own software development kits for general purpose and graphical calculations. They are called CUDA [5] and Stream, respectively (Stream was originally called CTM). A More general coding standard has emerged with the advent of the open source framework OpenCL [6] managed by the Khronos Group. It is now supported by both NVIDIA and ATI for GPU systems, but aims at becoming a standard for parallel programming of heterogeneous systems in general, including multi-core CPU and GPU systems collaboratively.
Code using GPU resources requires some learning. However the code sequence is quite easy to understand. The piece of code for the function to be executed on GPU is called a kernel. Prior to calling the kernel, memory must be copied from host (CPU) to device. There are methods to copy memory from host to device and vice versa. After calling the kernel, the result is copied back from device to host.

The typical code sequence is:

// 1. Allocate all inputs and outputs host and device memory

// 2. Copy input host memory to input device

// 3. Call the kernel function with inputs and outputs as arguments

// 6. Copy output device memory to output host

Fortunately, there are already some high level libraries which give the possibility to take advantage of GPU computing power without entering the details of the low level coding aspects. For instance, CUFFT [7] is the FFT specialized library for CUDA. It presents the same functionalities as the well known FFTW library [8], so it is very easy to handle for those who already know FFTW. Another important available library is CUBLAS for basic algebraic operations [7]. There are also projects to port these libraries to Matlab, like GPUmat [9,10]. Using GPUmat allows running interactively equivalent Matlab operations on GPU.

3. Comparative Study GPU versus CPU

In this section we present some results from tests we performed on a multi-core CPU machine integrating a GPU system, using various techniques. We tried to verify what is been currently said in the literature, about advantages and disadvantages of using one or another methods. We perform some basic model operations, i.e, a matrix product and a 1D FFT, using a set of selected open source libraries available for C language. We also have tested Matlab performances on CPU multi-cores. This study does not include the collaborative one CPU + GPU, but it should be noted that there are concrete projects like MAGMA [11] which aims at unifying CPU multi-cores and GPU computational power for the BLAS calculation toolkit [12].

3.1 Experimental Set-Up

We use a Dell desktop computer T3500 host a last generation Xeon quadri-core CPU, on which we mounted a last generation NVIDIA GTX470 graphics card on the PCIe slot. Both CPU and GPU processors are of 40 nanometer size technology. Their characteristics are reported on Table 2. The Linux platform used is the Scientific Linux 5.4, kernel 2.6.18, 64-bit (x86_64), gcc version 4.1.2.

Processor	Clock speed	# cores	GFLops 64	GFlops 32	Price
CPU Xeon E5530	2.4 GHz	4 (x2)	38.4	-	$530
GPU Nvidia GTX 470	1.2GHz	448	136.1	1088.64	$350

Table 2: Main characteristics and performances comparison between last generation CPU and GPU processors taken from constructors' literature, using Linpack benchmark for CPU and CUDA-Z for GPU.

With the present set-up, we checked that the best context for calculations are *single-precision for GPU*, while they are *double-precision for CPU*.

3.2 Matrix Operation

In a first approach, we perform a matrix multiplication using the following calculation:

```
#pragma omp parallel for shared(array, ncols, nrows) private(i, j, k)
for (i = 0; i < nrows; i++) {
 for (j = 0; j < ncols; j++) {
  for (k = 0; k < nrows; k++) {
   array[i][j] = array[i][k] * array[k][j];
```

```
} } }
```

We make use of OpenMP pragma to automatically parallelize the code on CPU. In the loop, variables *array*, *ncols* and *nrows* are shared among the threads, while variables *i*, *j*, and *k* are private to each thread. Of course, performances can be greatly improved by using a better algorithm. For instance, using the matrix multiplication from CBLAS [13], the C version of BLAS, we obtain a much better result event without parallelization. We greatly improve the performances by using a parallelized version of CBLAS on CPU multi-cores. This is what we have done using ATLAS [14], an optimized library for linear algebra especially compiled on our experimental platform. ATLAS is a parallelized version of CBLAS library. The piece of code for ATLAS (or CBLAS) is a unique call to the function *cblas_dgemm,* used for matrix multiplication.

A second interesting method is provided by Matlab. We use the straightforward vectorized matrix multiplication *C = AxB* from Matlab 2009A, which is automatically parallelized for CPU multi-cores.

We compared these methods with CUBLAS [14], the BLAS version adapted for Nvidia GPU system. From CUBLAS, we used the functions *cublasSetMatrix*, *cublasGetMatrix* to set and get the matrixes before and after calculation. We performed the calculation using either *cublasSgemm* and *cublasDgemm*, the matrix multiplication functions in single and double precision mode, respectively.

3.3 FFT 1D

The 1D FFT calculations have been performed using three different methods. The first two methods perform the calculation on CPU and the third one on GPU. The first method uses FFTW [8], a famous FFT library for C. We use the multithreaded capabilities of FFTW in our test. This requires an initialization and routine:

```
fftw_init_threads();
fftw_plan_with_nthreads(nth);
```

Where *nth* is the number of threads to run.

The piece of code implemented for the FFT 1D calculation is the following:

```
fftw_plan plan;
plan = fftw_plan_dft_1d(nIn,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
fftw_execute(plan);
fftw_destroy_plan(plan);
```

Where *nIn* represent the vector size, *in,out* the input and output vectors. Since it is a multithreaded version of FFTW, the application must terminate with the cleanup routine:

```
fftw_cleanup_threads();
```

The second method uses the *fft* Matlab function. It is coded in the most efficient manner, making fully use of the vectorization technique. In most recent versions, Matlab automatically take advantage of the multi-core CPU by instantiating multiple threads.
The third method uses CUFFT for Nvidia GPU calculations. The piece of code is very similar to the FFTW one:

```
cufftHandle plan;
cudaMemcpy(din,hin,sizeof(cufftComplex)*nIn,cudaMemcpyHostToDevice);
cufftResult planer;
cufftPlan1d(&plan, nIn,CUFFT_C2C,1);
cufftExecC2C(plan,din,dout, CUFFT_FORWARD);
cudaMemcpy(hout,dout,sizeof(cufftComplex)*nIn, cudaMemcpyDeviceToHost);
cufftDestroy(plan);
```

Where *nIn* represent the vector size, *hin,hout* the host input and output vectors, and *din,dout*, the device input and output vectors. Note that before and after executing the plan, a memory copy from host to device and vice versa is done.

3.4 Results

3.4.1 Matrix results

The results of computation time for a matrix multiplication are shown in Fig. 1. The biggest size computed is a 6000 x 6000 matrix size. Beyond this size, some computation errors appear in the GPU system either in double or single precision, certainly due to memory limitations (The GTX 470 card host 1280 MB GDDR5 onboard memory). The first observation is that the Matlab method seems the less efficient one, especially at small sizes. We observe two main regimes between ATLAS and CUBLAS methods. Below about 800 x 800 matrix size, the ATLAS computation appears to be the best method. Beyond this value, the CUBLAS methods appears to be the best one whatever precision used, by more then a factor 2 for double, and nearly a factor 10 for float.
To understand better how is distributed the calculation time for GPU, we report in Fig. 2 the latency contribution from the PCIe bus transfer. A general observation is that latencies from device GPU to host CPU are always bigger than from host CPU to device GPU. Moreover, double precision bus latencies are always bigger than single precision ones. This may be directly correlated with the size of data transferred. We see that the bigger the matrix size, the more the bus latencies contribute to the total time of computation. It goes from at least 50% for very small sizes to up to 90 % for the biggest size. For low matrix sizes (< 800 x 800), the bus transfer latencies are responsible for the worse performances of the GPU. Clearly, one may pay a lot of attention to the data transfers between CPU and GPU not to loose the GPU gain of computation. This requires considering the data size and the serialization of operations to be done on the GPU. However, despite this bottleneck, GPU computation appears as the best cost-effective method, even in double precision.

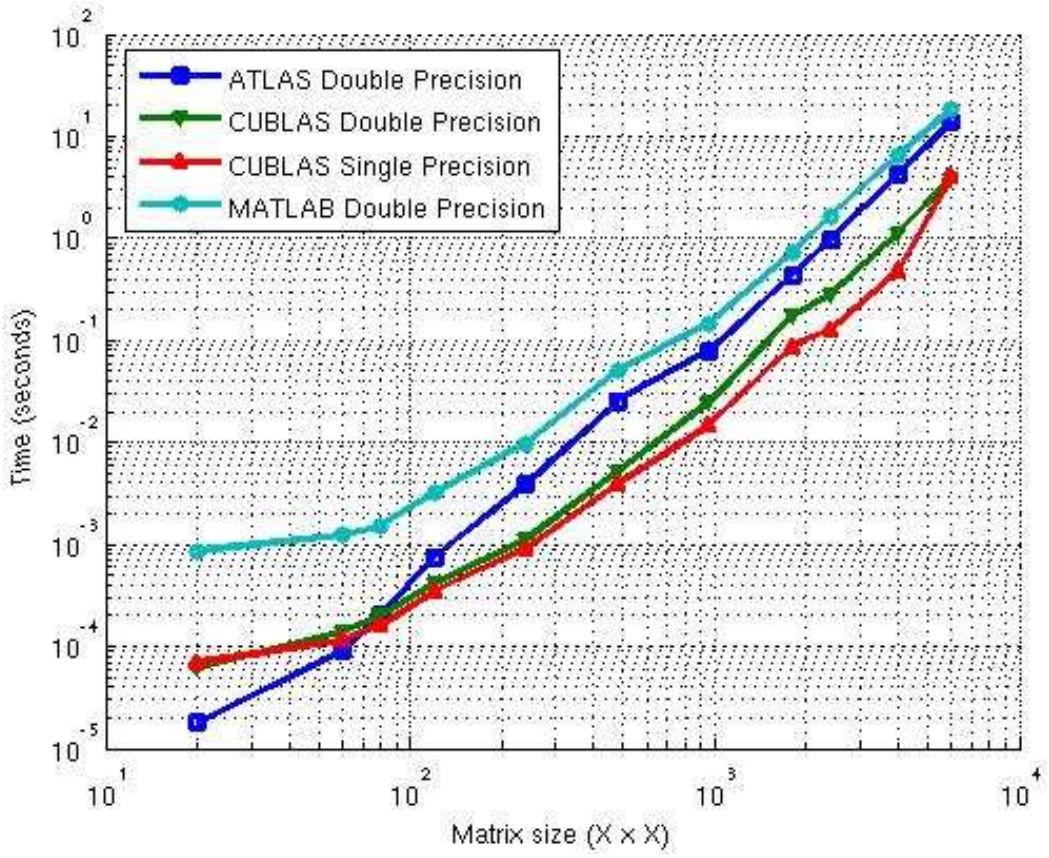

Fig. 1: Matrix multiplication parallel computation performances from Matlab (2009A) and ATLAS on CPU multi-core, compared to CUBLAS single and double precision on GPU.

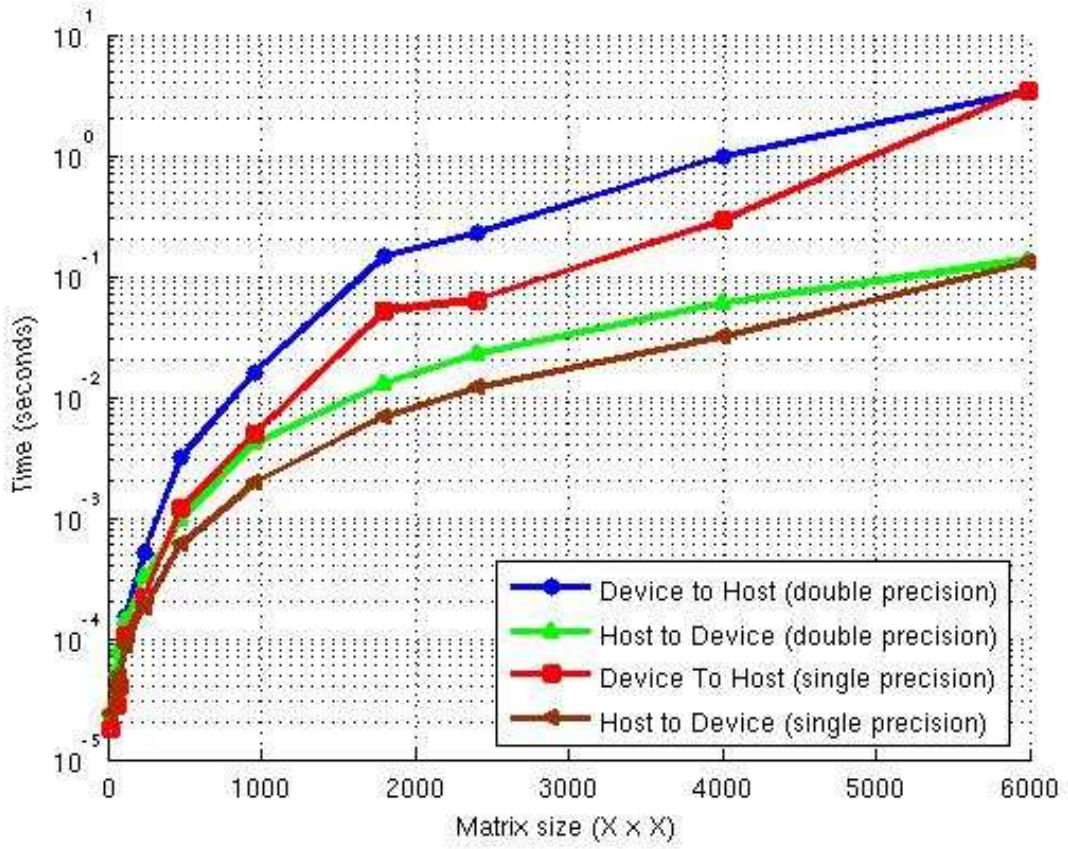

Fig. 2: PCIe bus data transfer latencies in both directions using CUBLAS on GPU using single and double precision.

3.4.2 FFT 1D results

The results of computation time for a FFT 1D are shown in Fig. 3. The biggest number of point computed is 8 millions. This limit is imposed by the CUFFT library. Note that for a FFT 2D and FFT 3D, the limit imposed is [2, 16384] points in any dimension. We observed data transfer and computation errors beyond this limit either in double or single precision. As illustrated in Fig. 3, We first note that in FFT 1D computations using multi-threaded FFTW, the optimal number of threads does not increase linearly as a function of the number of points in the FFT 1D. Using multi-threading appears interesting above 1 millions points, and the number of threads may be carefully adjusted each time, as the number of thread used may penalize the efficiency of FFTW. The results of computation times for all techniques used are reported in Fig. 4. Again we note that Matlab is globally the less efficient method, while CUFFT, the CUDA method for GPU, is the most powerful either in single and double precision. However, we note that the difference between FFTW and double precision CUFFT is reduced to only a factor 2 for 8 millions points used, while in

single precision, CUFFT is always about 10 times better than FFTW whatever the number of points used. Looking at bus latencies in Fig. 5, we see that contrary to matrix operations its contribution stays rather constant and does not exceed 50 % of the total computation time.

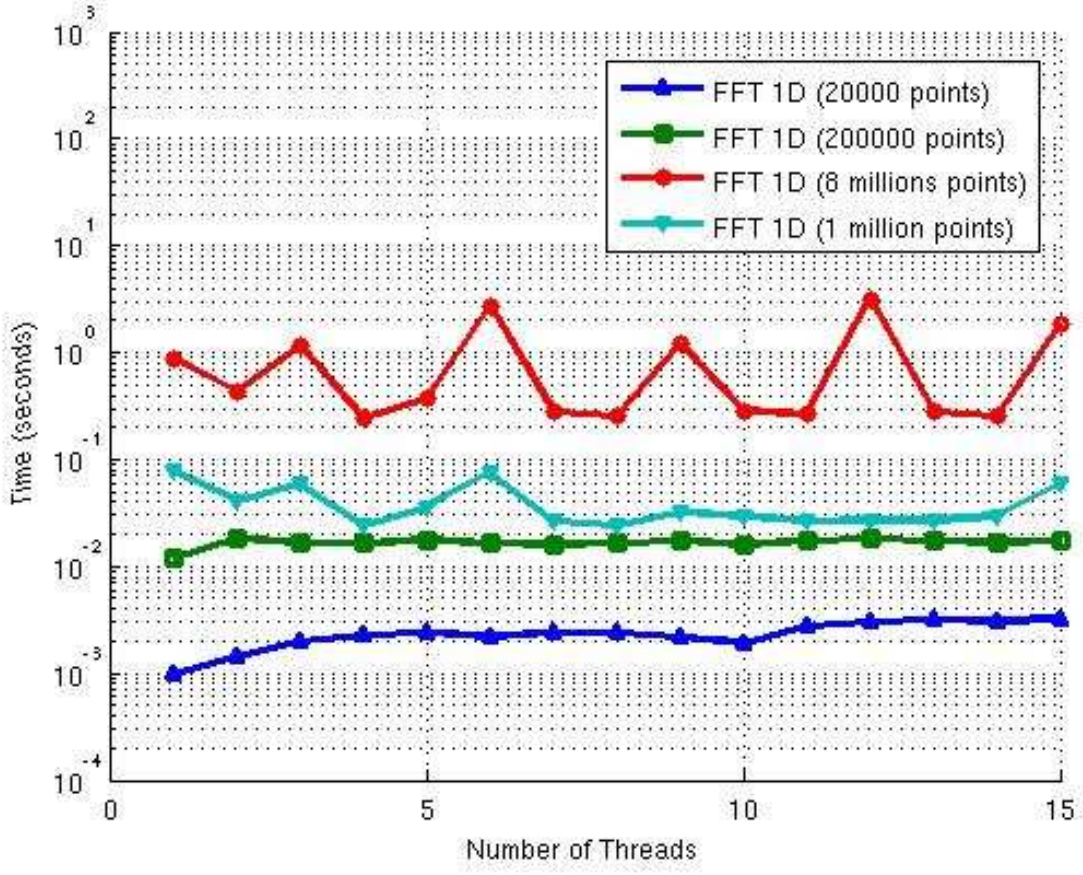

Fig. 3: FFT 1D computation performances of multi-threaded FFTW (on CPU) as a function of the number of threads used.

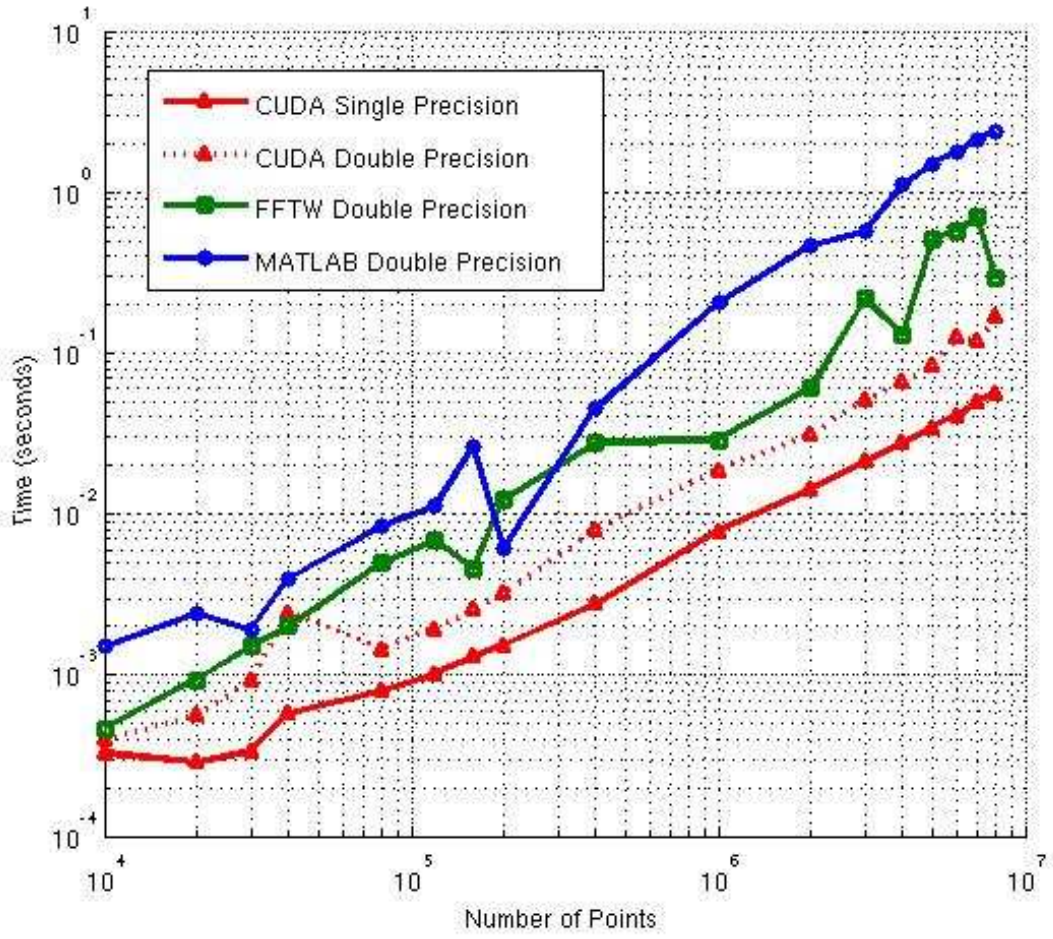

Fig. 4: FFT 1D performance comparison between CPU and GPU. For FFTW, the number of threads is adjusted manually to get the best result (between 1 and 10 threads). For GPU we use the CUFFT library from CUDA framework.

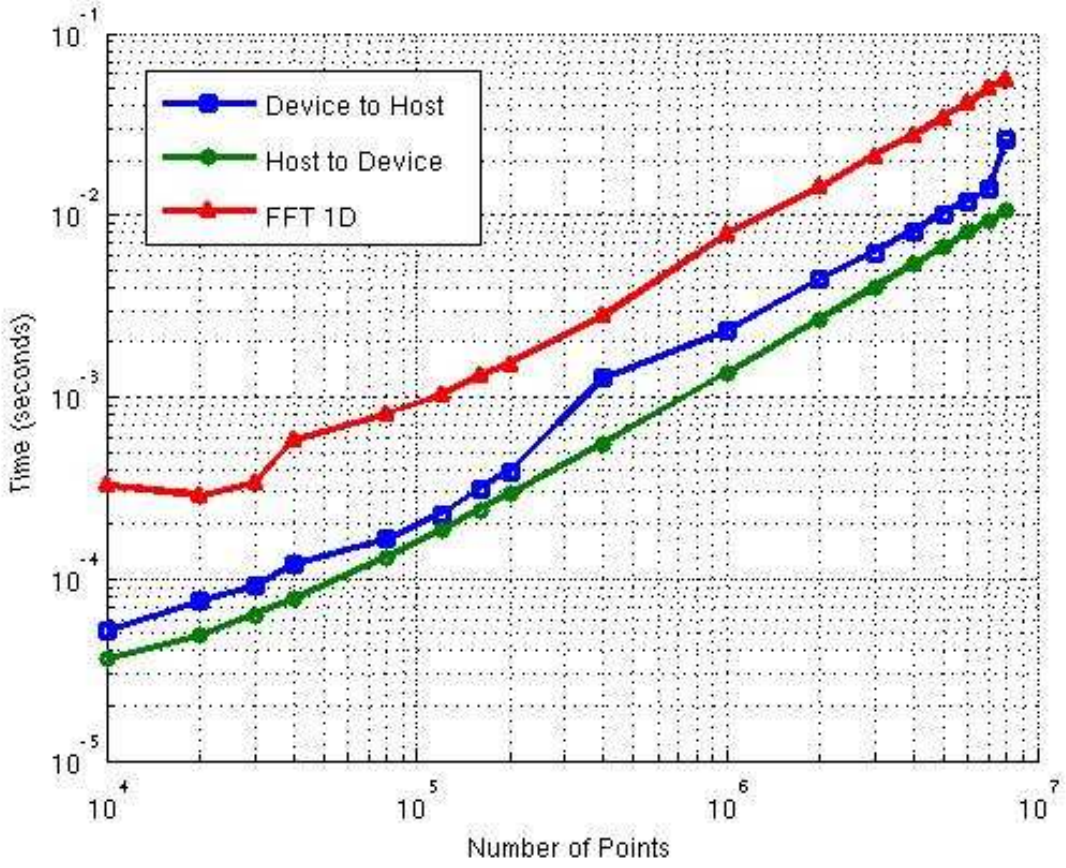

Fig. 5: PCIe bus data transfer latencies contribution to the FFT 1D calculation using CUFFT single-precision from CUDA.

4 Conclusion

We have presented a comparison between calculation performances on multi-core CPU and graphics cards GPU, using advanced computing methods. Our conclusion is that GPU systems are very promising calculation systems and may even become competitive in the super-computing area. We note that they are now breaking the TeraFlop barrier (in single precision), and we expect it to increase in the coming years. This leads us to think that super-computers scaling from desktop to cluster shall integrate more and more GPU systems collaboratively with CPUs in the future. Projects like OpenCL and MAGMA are interesting frameworks composing with such heterogeneous systems.

___oOo__