Initial Experiences Porting a Bioinformatics Application to a Graphics Processor

Maria Charalambous[1], Pedro Trancoso[1], and Alexandros Stamatakis[2]

[1]Department of Computer Science, University of Cyprus,
75 Kallipoleos Ave., P.O.Box 20537, 1678 Nicosia, Cyprus
{cs00cm, pedro}@ucy.ac.cy
http://www.cs.ucy.ac.cy/~pedro
[2]Institute of Computer Science, Foundation for Research and Technology-Hellas,
P.O. Box 1385, Heraklion, Crete, GR-771 10 Greece
stamatak@ics.forth.gr
http://www.ics.forth.gr/~stamatak

Abstract. Bioinformatics applications are one of the most relevant and compute-demanding applications today. While normally these applications are executed on clusters or dedicated parallel systems, in this work we explore the use of an alternative architecture. We focus on exploiting the compute-intensive characteristics offered by the graphics processors (GPU) in order to accelerate a bioinformatics application. The GPU is a good match for these applications as it is an inexpensive, high-performance SIMD architecture.

In our initial experiments we evaluate the use of a regular graphics card to improve the performance of RAxML, a bioinformatics program for phylogenetic tree inference. In this paper we focus on porting to the GPU the most time-consuming loop, which accounts for nearly 50% of the total execution time. The preliminary results show that the loop code achieves a speedup of 3x while the whole application with a single loop optimization, achieves a speedup of 1.2x.

1 Introduction

The demands from the game application market have been driving the development of better and faster architectures. One such example is the development of the Graphics Cards and more specifically the Graphics Processing Units (GPU). The GPUs are the responsible entities for drawing the fast moving images that we observe on the computer screens. To achieve those real-time realistic animations, the GPUs must perform many floating-point operations per second. As such, and given that the work performed by the GPUs is dedicated to these applications, the GPUs are forced to offer many more computational resources than the general purpose processors (CPU). Given the characteristics of these applications, performance can easily be improved from the use of vector units, *i.e.* using the SIMD programming model. In some way these GPUs have similar characteristics with the classical supercomputers (*e.g.* Cray supercomputers).

P. Bozanis and E.N. Houstis (Eds.): PCI 2005, LNCS 3746, pp. 415–425, 2005.
© Springer-Verlag Berlin Heidelberg 2005

While the first GPU models were only capable of handling graphics operations, as they were hardwired, the latest models offer the capability of executing user code. This was originally done for users, and game writers in particular, to be able to write their own graphics operations. Nevertheless, the programmability has opened the power of the GPU for other non-graphics applications. This has lead to the rising interest in a new research field known as General Purpose Computation on Graphics Processing Units or GPGPU [1].

Although different manufacturers offer different models of GPUs, the interface exported for their programming is standard and supported by the graphics card drivers. Currently the two major standards for the GPU interface are OpenGL [2] and DirectX [3]. Because these interfaces were written for programming graphics operations, they are not trivial to be used by general-purpose applications. Consequently, there are some efforts into making environments and tools that make the GPU programming easier for general-purpose applications. One such environment is BrookGPU [4]. Brook makes some extensions to ANSI C in order to support the execution of general-purpose applications on the GPU, making it relatively easy to port an application.

Several general-purpose applications have been mapped to the GPU: dense matrix multiply [5], linear algebra operations [6], sparse matrix solvers for conjugate gradient and multigrid [7], and database operations [8] among others.

In this paper we present our initial experiments in porting a bioinformatics application, RAxML, to execute on the GPU. RAxML is a program for inference of evolutionary trees based on the Maximum Likelihood method. After profiling RAxML's execution we focused on the porting to the BrookGPU environment of a single loop which accounts for nearly 50% of the original execution time. With this program we study the applicability of BrookGPU to real-world applications as RAxML has approximately 10000 lines of source code. In our experiments we compared the execution on a mid-class GPU (NVIDIA FX 5700LE) with a high-end CPU (Pentium 4 3.2GHz). The results show that the loop code may be accelerated by a factor of 3 when executing on the particular GPU. Given that we only optimized a single loop and also that we did not make a considerable effort yet into eliminating all the overheads, the overall improvement observed for the complete application is of 20%. These results are very encouraging and show that the GPU is a cost-effective solution for this application.

This paper is organized as follows: Section 2 presents the GPU architecture, its characteristics and limitations, along with its programming environment. Section 3 describes the bioinformatics application and its porting to the GPU. Section 4 shows the experimental setup and Section 5 discusses the experiments and results obtained. Finally the conclusions are presented in Section 6.

2 Graphics Processing Unit

2.1 Architecture

The GPU is an interesting architecture as it offers a large degree of parallelism at a relatively low cost. Its operations are similar to the well known *vector*

processing model. This model is also known from Flynn's taxonomy [9] as *Single Instruction, Multiple Data* or *SIMD*. As such, it is natural that the GPU will be able to perform well on many of the applications that in the past were executed on vector supercomputers.

Another characteristic of the simple parallel architecture of the GPU is that it allows for its performance to grow at a rate faster than the well known Moore's law. In fact the GPU has been increasing at a rate of 2.5 to 3.0x a year as opposed to 1.4x for the CPU.

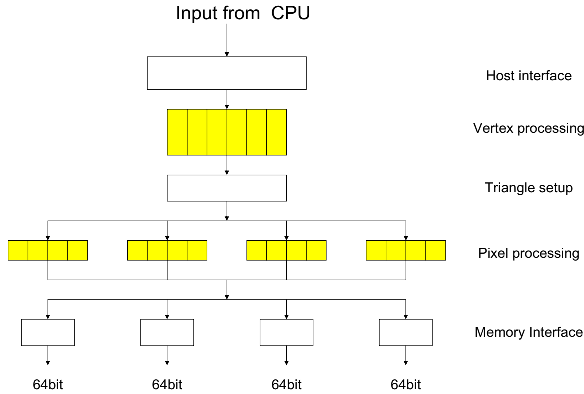


Fig. 1. GPU architecture

The general architecture of the GPU is depicted in Figure 1. Notice that the GPU includes two different types of processing units: vertex processors and pixel (or fragment) processors. This terminology comes from the graphics operations that each one is responsible for. For example, the vertex processor performs mathematical operations that transform a vertex into a screen position. This result is then pipelined to the pixel or fragment processor, which performs the texturing operations.

2.2 Programming Environment

As mentioned before, programming the GPU in an high-level language is a recent development. The first high-level language programming environments were developed for graphics applications in mind. Such examples are Cg from NVIDIA [10] and OpenGL Shading Language [11]. Although helpful, they make the job of mapping a general purpose application a considerable task. Therefore, some research teams are working on developing high-level language programming environments for general purpose programming. One such environment is BrookGPU [4] from Stanford University.

Brook is an extension to the standard ANSI C and is designed in order to facilitate the porting of general purpose applications to the GPU. The main

differences from the standard C language are the introduction of the concept of *stream* variables, and *kernel* and *reduction* functions. The programming model offered by BrookGPU for the functions to be executed on the GPU is a *streaming* model. In this model a function processes streams, *i.e.* sequences of data, but operates on a single element at a time.

Using Brook, a function that is executed on the GPU, can be of two types: *kernel* and *reduction*. The former is a general function that accepts multiple input and output parameters which may be of type stream or not. The latter is a function that takes only one stream parameter and returns a single stream output or scalar value. This is used to execute the known reduction operations such as a sum of all the values in a stream.

Finally, BrookGPU has the advantage that it is necessary to write the code once and its runtime will take care of selecting the correct implementation. For example, the same code can execute either on the CPU, or the GPU, using the OpenGL or the DirectX interface. In addition, it also provides an indirection layer such that the user does not need to be aware if the card has a NVIDIA or an ATI chip, for example.

3 GPU-RAxML

3.1 RAxML

RAxML (Randomized Axelerated Maximum Likelihood) [12,13] is a program for inference of evolutionary (phylogenetic) trees from DNA sequence data based on the Maximum Likelihood (ML) Method [14].

Phylogenetic trees are used to represent the evolutionary history of a set of n organisms. An alignment with the DNA or protein sequences of those n organisms can be used as input for the computation of phylogenetic trees. In a phylogeny the organisms of the input data set are located at the tips (leaves) of the tree whereas the inner nodes represent extinct common ancestors. The branches of the tree represent the time which was required for the mutation of one species into another new one.

The inference of phylogenies with computational methods has many important applications in medical and biological research, such as e.g. drug discovery and conservation biology [15]. Due to the rapid growth of sequence data over the last years it has become feasible to compute large trees which comprise more than 1.000 organisms. The computation of the tree-of-life containing representatives of all living beings on earth is considered to be one of the *grand challenges* in Bioinformatics.

The fundamental algorithmic problem computational phylogeny faces is the immense amount of alternative tree topologies which grows exponentially with the number of organisms n, e.g. for $n = 50$ organisms there exist $2.84 * 10^{76}$ alternative trees (number of atoms in the universe $\approx 10^{80}$). Thus, for most biologically meaningful optimality criteria the problem is NP-hard. Moreover, there is a speed/quality *trade-off* among the various evolutionary models which have been devised for tree reconstruction. This means that a phylogenetic analysis with an

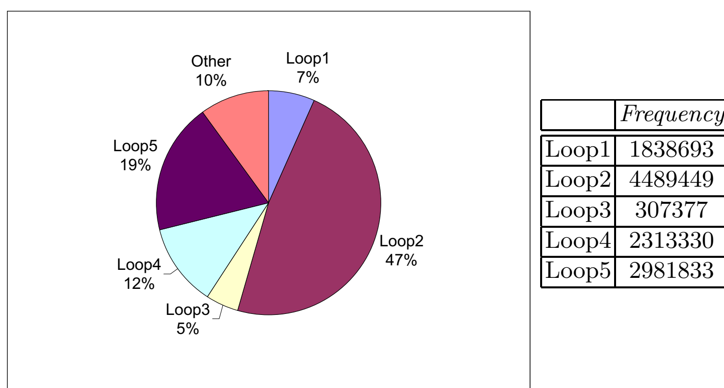
elaborate model such as ML requires significantly more time but yields trees with superior accuracy than Neighbor Joining [16] or Maximum Parsimony [17,18]. However, due to the higher accuracy it is desirable to infer complex large trees with ML.

The current version of RAxML incorporates novel fast hill climbing and simulated annealing heuristics and is, to the best of our knowledge, the currently fastest and at the same time most accurate program for phylogenetic inference with ML on real world sequence data. Moreover, it has significantly lower memory requirements than comparable implementations [19]. Finally, like every ML-based program, RAxML exhibits a source of fine-grained loop-level parallelism in the likelihood functions which consume over 90% of the overall computation time (see Section 3.2).

3.2 RAxML Profiling

Before porting the application to BrookGPU we profiled its execution time in order to identify the most time-consuming portions of the code. We concentrated our analysis on the loops that had been parallelized for the OpenMP version of RAxML [20]. For each loop we added instructions to measure the time consumed by the corresponding loop execution and also to measure its frequency. The execution time was measured with accuracy using the processor's hardware performance counters [21]. The results obtained for the *test150* input data set (see Section 4) are presented in Figure 2.

These results show that the most time-consuming piece of code is *loop2*, which is visited 4489449 times and accounts for 47% of the total execution time. Equally important is the fact that the five loops that were identified in the profiling phase account altogether for 90% of the total execution time. Also, an analysis of the code shows that the code of the loops is vectorizable without



	Frequency
Loop1	1838693
Loop2	4489449
Loop3	307377
Loop4	2313330
Loop5	2981833

Fig. 2. Execution time and loop frequency profile of RAxML for test150

major changes. Consequently, the profiling results indicate that this application has a good potential for speedup when ported to the GPU.

3.3 Porting RAxML to the GPU

Based on the results presented in the last Section, in our initial experiments we focus on porting *loop2* to execute on the GPU. To achieve this goal we use the BrookGPU system as presented in Section 2.2. This porting is a good case-study for studying the applicability of BrookGPU as a platform for real-world applications as RAxML is composed of approximately 10000 lines of source code.

In BrookGPU there are two types of source files: regular C++ (or C) source files (*.c* files) and Brook source files (*.br* files). The latter are the ones containing the code that is to be executed on the GPU. The code to execute on the GPU has to be included in special functions named *kernel*. These *kernel* function perform the same set of operations on different data elements belonging to the same *stream* of data.

The first step in porting the application is to identify the code to execute on the card. To be effective this has to be a loop in the original source code. Once this loop is identified, a pre-condition for it to be ported to the GPU is that there are no dependencies in the data accesses in the loop's operations across different iterations. This means that the data accesses in a certain iteration are different from the ones in any other iteration, therefore independent, and consequently iterations may safely be executed in parallel, without resulting in any conflicts. In practice this is the definition of a loop being vectorizable. For our example an excerpt of the code is presented in Figure 3.

```
...
for (i = 0; i < tr->cdta->endsite; i++) {
      fxqr = tr->rdta->freqa * lqa[i] + tr->rdta->freqg * lqg[i];
      fxqy = tr->rdta->freqc * lqc[i] + tr->rdta->freqt * lqt[i];
      fxqn = fxqr + fxqy;
      ...
      lpa[i] = sumaq * (zzr * (lra[i] - tempi) + tempj);
      ...
}
...
```

Fig. 3. Original loop2 code

After the verification that the loop code is vectorizable, we need to identify the data accessed within an iteration. Simple variables will not be changed but arrays will have to be transformed into streams. In addition, we need to determine if the data is accessed in a read-only, write-only, or read-write fashion.

The next step is to extract the loop code and create a kernel function with it. The function parameters are the variables as identified in the previous step. In addition, the keyword *out* needs to be used for all returning parameters. Notice

that this function, as it contains the code to be executed on the GPU, needs to be placed in a separate file, a *Brook* file (*.br*), in order for the corresponding code to be correctly generated by the system. For our example mentioned above, the Brook kernel function is the one presented in Figure 4.

```
kernel void
second_loop(float4 lq<>, out float lp_x<>, ...
          float i_freqa, float i_freqg, float i_freqc, float i_freqt,...)
{
    float fxqr, fxqy, fxqn;
    ...
    fxqr = i_freqa * lq.x + i_freqg * lq.y;
    fxqy = i_freqc * lq.z + i_freqt * lq.w;
    fxqn = fxqr + fxqy;
    ...
    lp_x = sumaq * (zzr * (lr.x - tempi) + tempj);
    ...
}
```

Fig. 4. Brook kernel function for loop2 code.

In the original code, the loop code is replaced with a function call to the new kernel function. In addition, it is necessary to pack the arrays into streams before the kernel call and unpack the output streams into regular arrays after the kernel call. The packing function is called *streamRead* and the unpacking one *streamWrite*. For our example the code looks like the one presented in Figure 5.

```
...
streamRead(lq, lq_array);
second_loop(lq,lp_x,...,i_freqa,i_freqg,i_freqc,i_freqt,...);
streamWrite(lp_x, lpa);
...
```

Fig. 5. Application with call to kernel function

It is necessary to note that this procedure may not be straightforward in all cases as the use of the graphics card imposes some limitations. Examples of such limitations are the size of the stream used, the type and number of parameters that may be passed to the kernel function, and the operations performed within the kernel function. A detailed analysis of some of these limitations and proposed solutions is presented in [22].

4 Experimental Setup

For the experiments presented in this paper we used one graphics card and one computer setup. The graphics card used was a NVIDIA GeForce FX 5700

LE [23]. This card has a NV36 graphics processor clocked at 250MHz, 128MB DDR video memory clocked at 200MHz and the data transfers with the PC are done through the AGP interface. The NV36 processor includes 3 vertex and 4 pixel pipelines [24].

As for the computer system we used a *high-end* Intel Pentium 4 3.2GHz based system with 1GB RAM.

The application used is RAxML as described in Section 3.1. The input data set used in these experiments is composed of an alignment of 150 sequences or organisms where each organism is represented by a DNA sequence of a length of 1269 nucleotides. This input set is named *test150*.

The environment used was BrookGPU version 0.3 [4] as described in Section 2.2. For the experiments, Brook was compiled using the Intel C++ compiler with the release compile flag, *i.e.* with full code optimizations.

For the experiments we compare the execution time of running the code on the regular CPU of the system and on the GPU of the graphics card. We measure these two situations using the same code as the BrookGPU runtime allows the user to decide where the code should be executed depending on the value of an environment variable (*BRT_RUNTIME*). If we set the variable *BRT_RUNTIME* to *cpu* the code will execute on the system's CPU while if we set the variable to *nv30gl* it will execute on the card's GPU.

5 Experimental Results

As previously described, in this preliminary study we focused on porting one loop, *loop2*, from the original code to execute on the GPU. The speedup obtained from executing the modified application on the GPU comparing to the execution on the CPU is shown in Figure 6.

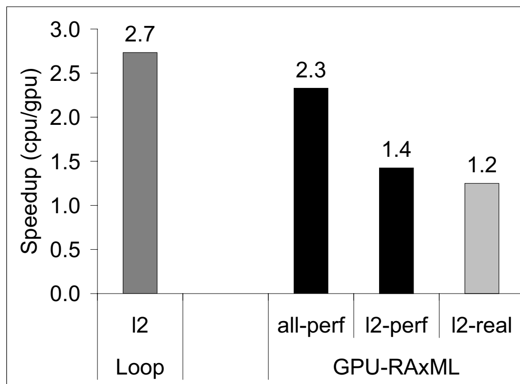


Fig. 6. Speedup of GPU-RAxML when executing on GPU compared to CPU

In this Figure we present four bars separated into two groups: *Loop* on the left, which represents the results for the speedup of the *loop2* loop code; and *GPU-RAxML* on the right, which represents the results for the speedup of the complete RAxML application compared to the execution on the CPU. In the latter group we have three bars: two dark ones representing predicted speedup and a lighter one representing measured speedup. From left to right, the first bar, *all-perf*, represents the predicted speedup for the complete application, assuming that all loops have been ported to the GPU and that each loop achieves the same speedup as the one observed for *loop2*. The second bar, *loop2-perf*, represents the predicted speedup for the complete application, assuming that only *loop2* has been ported to the GPU. Both predicted values have been determined using Amdahl's law, the profiling information, as well as the speedup measured for *loop2*. Finally, the third bar, *loop2-real*, represents the speedup measured for the whole application in the situation where only *loop2* has been ported to the GPU. Notice that both the *loop2* and predicted speedup values do not take into account any extra overheads of sending and receiving data to and from the GPU, hence the suffix *perf* for perfect speedup, while the *loop2-real* includes all overheads.

From the results in the Figure 6 it is possible to observe that the use of the GPU is very effective. For the code of the loop we optimize, the GPU achieves a speedup of nearly 3x. It is important to notice that the GPU used is not a high-end model while the CPU used is a high-end model. As a consequence, this is not a very fair comparison for the GPU. A more realistic setup would have either a high-end GPU or a lower-end CPU. This would result in a much larger advantage for the GPU.

Regarding the overall application speedup, notice that the observed real speedup is only 1.2, *i.e.* the GPU performs only 20% better than the CPU. Nevertheless, this is 86% of the perfect speedup of 1.4. The 14% of the "lost" speedup is due to data transfer and other overheads related to the preparation of the data to be sent and received from the GPU. At this point we did not concentrate in eliminating these overheads as we are still studying the potential for the use of the GPU for this application. In the near future we will study some changes to the original program in order to reduce these overheads and bring the *real* speedup closer to the *pref* one. In addition, we are currently porting the rest of the loops to the GPU and therefore we expect an increase in the observed speedup that could reach the predicted value of 2.3 shown in the *all-perf* bar.

Notice that the speedup observed is even more impressive when we consider the cost and power consumption of the processors used. If we consider the price information in PriceWatch [25], the NVIDIA 5700LE card costs approximately US$75 while the Pentium 4 US$200, *i.e.* 2.7 times more expensive. Furthermore, while the NVIDIA chip consumes approximately 24W [26], the Pentium 4 3.2GHz consumes approximately 5.5x more power, as it consumes more than 130W. This leads to the conclusion that the use of the GPU is a very cost- and power-effective solution for this type of application.

6 Conclusions

In this paper we presented our initial experiments in evaluating the potential of porting a bioinformatics application to execute on the Graphics Processor Unit. For the preliminary results presented in this paper we focused on porting to the BrookGPU environment a single loop from the RAxML application, which typically accounts for nearly 50% of the original execution time.

In our experiments, we compared the execution on a mid-class GPU (NVIDIA FX 5700LE) with a high-end CPU (Pentium 4 3.2GHz). In the experiments performed, the loop code achieved a speedup of 3x when executing on the GPU compared to the high-end CPU. Although this has resulted in only an improvement of 20% in the overall application, this value will increase as we port the rest of the loops to the GPU and perform some code modifications to reduce certain overheads.

Overall, the loop speedup results are very encouraging and lead us to conclude that the GPU is a solution that is able to achieve high-speedup with lower cost and less power consumption compared to high-end systems.

Acknowledgments

We would like to thank Michael Ott from Technical University of Munich for his help with the RAxML code and the anonymous reviewers for their valuable comments. Finally, we thank the German Academic Exchange Service (DAAD) for funding the postdoctoral research of A. Stamatakis.

References

1. GPGPU: General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/(2005)
2. Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification (Version 2.0) (2004)
3. Peeper, C.: DirectX High Level Shading Language. Microsoft Meltdown UK Presentation, Microsoft Corporation (2002)
4. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transactions on Graphics **23** (2004) 777–786
5. Larsen, E., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), ACM Press (2001) 55–55
6. Kruger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics **22** (2003) 908–916
7. Bolz, J., Farmer, I., Grinspun, E., Schrooder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Transactions on Graphics **22** (2003) 917–924
8. Govindaraju, N., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast Computation of Database Operations using Graphics Processors. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM Press (2004) 215–226

9. Flynn, M.: Very high-speed computing systems. Proceedings of the IEEE **54** (1966) 1901–1909
10. Mark, W., Glanville, R., Akeley, K., Kilgard, M.: Cg: a system for programming graphics hardware in a C-like language. ACM Transactions on Graphics **22** (2003) 896–907
11. Kessenich, J., Baldwin, D., Rost, R.: The OpenGL Shading Language (2004)
12. Stamatakis, A., Ludwig, T., Meier, H.: RAxML-III: A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees. Bioinformatics **21** (2005) 456–463
13. Stamatakis, A.: An Efficient Program for phylogenetic Inference Using Simulated Annealing. In: Proceedings of IPDPS2005, Denver, Colorado, USA (2005)
14. Felsenstein, J.: Evolutionary trees from DNA sequences: A maximum likelihood approach. Journal of Molecular Evolution **17** (1981) 368–376
15. Bader, D., Moret, B.M., Vawter, L.: Industrial Applications of High-Performance Computing for Phylogeny Reconstruction. In: Proceedings of SPIE ITCom: Commercial Applications for High-Performance Computing, Denver, Colorado, USA (2001) 159–168
16. Gascuel, O.: BIONJ: An improved version of the NJ algorithm based on a simple model of sequence data. Molecular Biology and Evolution **14** (1997) 685–695
17. Guindon, S., Gascuel, O.: A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood. Systematic Biology **52** (2003) 696–704
18. Williams, T., Moret, B.M.: An Investigation of Phylogenetic Likelihood Methods. In: Proceedings of 3rd IEEE Symposium on Bioinformatics and Bioengineering (BIBE'03), Bethesda, Maryland, USA (2003)
19. Stamatakis, A., Ludwig, T., Meier, H.: New Fast and Accurate Heuristics for Inference of Large Phylogenetic Trees. In: Proceedings of IPDPS2004, Santa Fe, New Mexico, USA (2004)
20. Stamatakis, A., Ott, M., Ludwig, T.: RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. In: Proceedings of 8th International Conference on Parallel Computing Technologies (PaCT2005), Krasnojarsk, Russia (2005) Preprint available on-line at `www.ics.forth.gr/~stamatak`.
21. Intel: IA-32 Intel Architecture: Software Developers Manual. Volume 3 of System Programming Guide. Intel (2003)
22. Trancoso, P., Charalambous, M.: Exploring Graphics Processor Performance for General Purpose Applications. In: Proceedings of the Euromicro Symposium on Digital System Design, Architectures, Methods and Tools (DSD 2005). (2005)
23. NVIDIA: NVIDIA GeForce FX: Performance. http://www.nvidia.com/page/fx_5700.html (2005)
24. TechPowerUp: GPU Database. http://www.techpowerup.com/gpudb/ (2005)
25. PriceWatch: Price Comparison Search Engine. http://www.pricewatch.com (2005)
26. Tscheblockov, T.: Power Consumption of Contemporary Graphics Accelerators. http://www.xbitlabs.com/articles/video/display/ati-vs-nv-power.html (2004)