Attaining High Performance in General-Purpose Computations on Current Graphics Processors

Francisco Igual-Peña, Rafael Mayo-Gual, and Enrique S. Quintana-Ortí

Depto. Ingeniería y Ciencia de los Computadores, Universidad Jaume I, 12.071–Castellón, Spain, {figual,mayo,quintana}@icc.uji.es

Abstract. The increase in performance of the last generations of graphics processors (GPUs) has made this class of hardware a coprocessing platform of remarkable success in certain types of operations. In this paper we evaluate the performance of linear algebra and image processing routines, both on classical and unified GPU architectures and traditional processors (CPUs). From this study, we gain insights on the properties that make an algorithm likely to deliver high performance on a GPU.

Key words: Graphics processors (GPUs), general purpose computing on GPU, linear algebra, image processing, high performance.

1 Introduction

During the last years, since the emergence of the first generation of programmable graphics processors (GPUs), many studies have evaluated the performance of these architectures on a large number of applications. Thus, linear algebra operations [10, 6], medical image processing [9, 12], or database querying [8] are just a few examples of different arenas in which GPU computation has been successfully applied.

Recently, the design of GPUs with unified architecture and the development of general-purpose languages which enable the use of the GPU as a general-purpose coprocessor has renewed and increased the interest in this class of processors. Unfortunately, the rapid evolution of both the hardware and software (programming languages) of GPUs has outdated most of the performance studies available to date.

In this paper, we design and implement a reduced collection of "benchmark" routines, composed of four linear algebra operations (matrix-matrix product, matrix-vector product, saxpy, and scaling of a vector) and an image processing kernel (convolution filter). These routines are employed to evaluate the impact of the improvements introduced in the new generation of GPUs (*Nvidia G80*), comparing the results with those obtained on a GPU from a previous generation (*Nvidia NV44*) and current general-purpose processors (*AMD Athlon XP 2400+* and *Intel Core 2 Duo*). The ultimate purpose of this evaluation is to characterize the properties that need to be present in an algorithm so that it can be correctly and efficiently adapted into the GPU execution model.

The rest of the paper is organized as follows. Section 2 describes the basic architecture and execution model of both the old and new generations of GPUs Section 3 characterizes the routines in the benchmark collection. Sections 4 and 5 evaluate the performance of the benchmark routines on the *Nvidia NV44* and the *Nvidia G80*, respectively, comparing the results with those obtained on a CPU, and identifying a set of properties that must be present in an algorithm to deliver high performance on that GPUs. Finally, Section 6 summarizes the conclusions that can be extracted from our analysis.

2 GPU Architecture and Execution Model

2.1 GPU graphics pipeline

The graphics pipeline consists of a set of sequential stages, each one with a specific functionality and operating on an specific type of data. The process transforms original graphical information (vertices) into data suitable for being shown on display (pixels). Figure 1 illustrates the usual stages (or phases) that form the graphics pipeline.

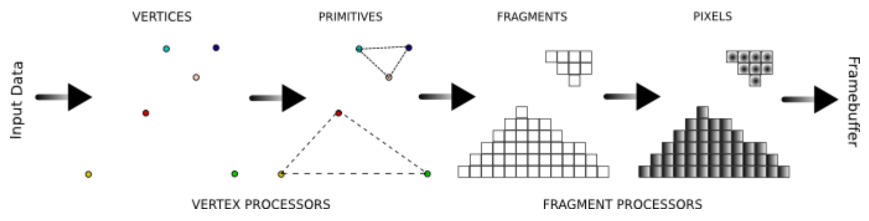

Fig. 1. Graphics pipeline process with its main stages.

Current GPUs implement this pipeline depending on the generation they belong to. Thus, classical GPUs have specific hardware units, known as *shaders* (or processors), for each one of the stages of the graphics pipeline. On the other hand, GPUs from the latest generation have a *unified shader* (or unified processor), with the ability to both execute any of the stages of the pipeline and work with any type of graphical data.

2.2 Classical architecture

Until 2006 GPUs were based on a design where each pipeline stage was executed on a specific hardware unit or processor inside the pipeline. Thus, e.g., vertices are processed by *vertex processors* while pixels (also called fragments) are transformed by *fragment processors.* In practice, general-purpose algorithms implemented on these classical architectures exploit fragment processors only, due to their larger number and broader functionality. Fragment processors operate in SIMD mode, taking a fragment as input, and processing its attributes; they can also process vectorial data types, working simultaneously on the four components of a fragment (R, G, B, and A). This class of hardware is able to read from random memory locations (commonly known as a *gather* operation in graphics algorithms), but can only modify one memory position per processed fragment, the one associated with the position of the fragment. This lack of support for *scatter* is one of the main restrictions of the classical GPU.

In the latter generations of this "classical architecture", programming capabilities were added to vertex and fragment processors. Altogether, the previous characteristics enable the use of fragment processors as a hardware platform to process non-graphical data. Unfortunately, the graphical-oriented design of this class of hardware, its SIMD execution model, the lack of a sophisticated memory hierarchy and the use of graphical-oriented APIs are problems for an efficient implementation of general-purpose applications on the GPU.

2.3 Unified architecture

In 2006 a new generation of GPUs was introduced, with a completely different architectural design. These new platforms feature a *unified architecture*, with one processing unit or unified shader that is able to work with any kind of graphical data, transforming the sequential pipeline in Figure 1 into a cyclic one, in which the behavior of the unified shader varies depending on the stage of the pipeline that it is being executed at each moment.

There are several characteristics in the new generation of GPUs which specifically favor their use as a general-purpose coprocessor: in general, the clock frequency of the unified shader is much higher than that of a fragment processor (even though it is still much lower than the clock frequency of current CPUs); the shader consists of a large collection of computation units (up to 128, depending on the GPU version), called Streaming Processors (SPs), which operate in clusters of 16 processors in SIMD mode on the input data stream; and the architecture includes a sophisticated memory hierarchy, which comprises a L2 cache and small fast memories shared by all the SP in the same cluster.

These hardware advances are complemented with the CUDA [3] general-purpose programming library, which eases the programming effort on these platforms. In fact, CUDA has been proposed as a standard (although only compatible with Nvidia hardware) to program the new generation of GPUs, without the requirement of learning more complex graphics-oriented languages.

3 Benchmark Collection

In order to identify the algorithmic properties that yield correct and efficient codes for the GPU execution model, we have studied three major computational aspects of algorithms:

Data parallelism. The replication of functional units inside the GPU (fragment processors in the non-unified architectures, SPs in the unified architectures) makes this class of architectures specially appropriate for applications which exhibit a high degree of data parallelism.

Input data reutilization. The simple memory hierarchy in non-unified GPUs makes it difficult to exploit the locality of reference; in these architectures, high memory latency and limited bus bandwidth imply a penalty cost much higher than in a CPU; for this reason, input data reutilization is one of the biggest issues when trying to attain high performance on graphics processors.

Computational intensity per stream element. Due to the previous restriction, to achieve high performance the expensive cost of memory references should be masked with a high number of operations per memory access.

Our benchmark collection is composed of four *Basic Linear Algebra Subprograms* or BLAS [5]: the matrix-matrix product (`SGEMM`), the matrix-vector product (`SGEMV`), the "saxpy" (`SAXPY`), and the scaling of a vector (`SSCAL`); and a convolution filter, common in image processing. From the computational viewpoint, the routines in the benchmark present the following properties:

SGEMM The matrix multiplication routine,

$$C = \alpha \cdot A \cdot B + \beta \cdot C$$

where A is $m \times k$, B is $k \times n$, and C is $m \times n$, being α and β scalars, features some properties that make it a good candidate to achieve good results when mapped into graphics hardware. It exhibits a regular memory access pattern, a high degree of data parallelism, and a very high computational load. On the other side, it is interesting to study the importance of the high input data reutilization in this type of algorithm.

For our study, we have chosen square matrices to evaluate the performance of the routine, and a non-transposed memory layout. The scalars α and β were set to 1. For a detailed study of the `SGEMM` performance on a GPU, refer to [1].

SGEMV The matrix-vector multiplication routine

$$y = \alpha \cdot A \cdot x + \beta \cdot y$$

where A is a $m \times n$ matrix, x and y are vectors of length n and α and β are scalars, exhibits a smaller input data reutilization than `SGEMM`. Thus, while each input element for the `SGEMM` routine is used $O(n)$ times to compute the result, `SGEMV` reutilizes $O(n)$ times the data of the input vector, but only $O(1)$ times the data of the input matrix. This behavior makes the matrix-vector product routine a more streaming-oriented code, and so it is theoretically possible to achieve better results on a GPU. The simplest form of de `SGEMV` routine will be evaluated, with the matrix A not transposed in memory, and $\alpha = \beta = 1$.

SAXPY and SSCAL The BLAS-1 routines `SAXPY` and `SSCAL`

$$y = \alpha \cdot x + y \qquad\qquad x = \alpha \cdot x$$

where x and y are vectors, and α is a scalar, are specially interesting for graphics processors, as they do not reutilize input data at all. These algorithms fit perfectly to the GPU architecture explained in Section 2. In fact, they can be seen as fully stream-oriented algorithms, where the input is a stream of data (for the `SSCAL` routine) or two streams (for the `SAXPY`), operations are performed over each of the elements of the input stream, without any kind of data reutilization, and finally an output data stream is returned.

The main difference between these two operations, from the performance viewpoint, is the amount of computational load per stream element. Thus, `SAXPY` performs twice as many operations as `SSCAL` per element. This difference offers some information on the importance of the computational load in the performance of the processor.

2D Convolution Image processing algorithms traditionally exhibit a high performance when executed on graphics processors. More specifically, the convolution filters exhibit some properties which favor GPU hardware. First, the high degree of data parallelism will take advantage of fragment processors (or SP) replication of modern GPUs. Second, input data reutilization is very low (proportional to the size of the applied filter, usually small). Third, the computational load per calculated element is high, and based on multiply-and-add (MAD) operations, for which the GPU is specially appropriate.

For our evaluation purposes, we have implemented a bidimensional convolution filter with a square mask of different sizes, comparing optimized versions on CPU, using tuned BLAS libraries, and on GPU, using optimized Cg and CUDA implementations.

4 Previous Generation GPU-CPU Comparison

4.1 Experimental setup

In this first experiment, we have chosen two experimental platforms of the same generation, an *AMD AthlonXP 2400+* CPU and a *Nvidia NV44* GPU processor (both from year 2004), so that we can do a fair comparison between general-purpose and graphics processors. Details on these architectures are given in Table 1. The GNU `gcc` 4.1.2 compiler is employed in the evaluation.

4.2 Implementation details

The highly tuned implementation of linear algebra kernels in GotoBLAS 1.15 [7] was used to evaluate the performance of the CPU. The convolution filter implementation was built on top of GotoBLAS, using exclusively fully optimized BLAS operations.

	CPU	GPU
Processor	*AMD AthlonXP 2400+*	*Nvidia GeForce 6200*
Codename	Thoroughbred A	NV44A
Clock frequency	2 GHz	350 MHz
Memory speed	2×133 MHz	2×250 MHz
Peak performance	8 GFLOPS	11.2 GFLOPS
Bus width	64 bits	64 bits
Max. bandwidth	2.1 GB/s	4 GB/s
Memory	512 MB DDR	128 MB DDR
Bus Type	AGP 8x (2 GB/s transfer rate)	
Year	2004	2004

Table 1. Description of the hardware used in our first experimental study.

On the other hand, the GPU was programmed using OpenGL and the Cg language (version 1.5). The routines were adapted to the architecture of the *Nvidia NV44* in order to optimize performance, as is briefly described next.

For routine SGEMM, we start from a simple implementation, applying successive refinements in pursue of high performance. First, we adapt the original algorithm using the vectorial capabilities of the fragment processors, as proposed in [4]. This type of optimization usually yields a four-fold increase in performance, and is frequently applied to all types of GPU codes. In addition, we try to exploit the simple cache hierarchy of the *Nvidia NV44* by implementing a multipass algorithm, following the ideas in [11]. The goal here is analogous to blocking techniques for CPUs; however, this technique often delivers poorer results on GPUs as the multiple memory writes after each rendering pass penalize the global performance. In general, an SIMD architecture attains higher performance when the instructions are executed only once on the data stream.

We have also implemented optimized versions of routines SGEMV, SAXPY, and SSCAL which exploit the vectorial capabilities of the GPU by applying analogous optimizations to those described above for routine SGEMM.

Convolution filters allow us to introduce simple but powerful optimizations starting from a basic implementation. Our proposal to achieve high performance when executing this operation on a GPU is to divide the original $N \times N$ image into four $N/2 \times N/2$ quadrants. For simplicity, we assume here that N is a multiple of 2; the overlap applied to the boundaries is not illustrated. We then map the (i, j) elements of the four quadrants onto the four channels (R, G, B, and A) of an $N/2 \times N/2$ data structure. Since a GPU can process four-channel tuples as a scalar element, we can get up to four times higher performance with this type of optimization. Figure 2 illustrates the process. Although this strategy is quite simple, it illustrates the type of optimizations that can be easily applied when implementing general-purpose algorithms on a GPU.

4.3 Experimental results

By analyzing the experimental results, the goal to determine which algorithmic properties (computational aspects in Section 3) favor the execution of an algorithm on a GPU with a classical architecture.

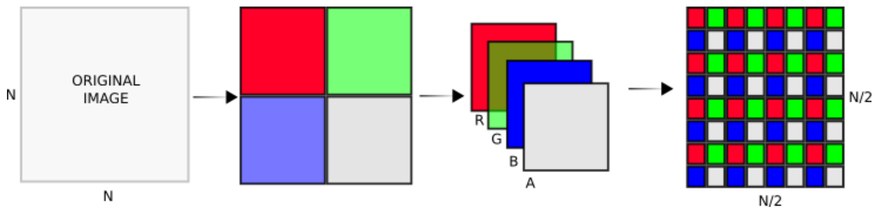

Fig. 2. Optimization applied to the computation of a convolution filters on a GPU with a classical architecture.

Data reutilization: `SGEMM` vs. `SGEMV`

Figure 3 shows the results for routines `SGEMM` and `SGEMV` on the CPU and GPU. On the latter architecture, we report two different MFLOPs rates, labeled as "`GPU`"/"`GPU w. TX`", obtained respectively by measuring only the execution time on the GPU or timing also the period required to transfer data and results between RAM and video memory. The high input data reutilization of the matrix-matrix product (see left-hand side plot) explains why the routine in Goto BLAS, which exploits the sophisticated cache hierarchy of the AMD CPU, outperforms the GPU implementation by a factor up to 4. The right-hand side plot illustrates how, when the data reutilization is lower as, e.g., in the matrix-vector product, the difference in performance between the CPU and GPU routines decreases, though still favors the CPU (between two and three times higher MFLOPs rate on this architecture).

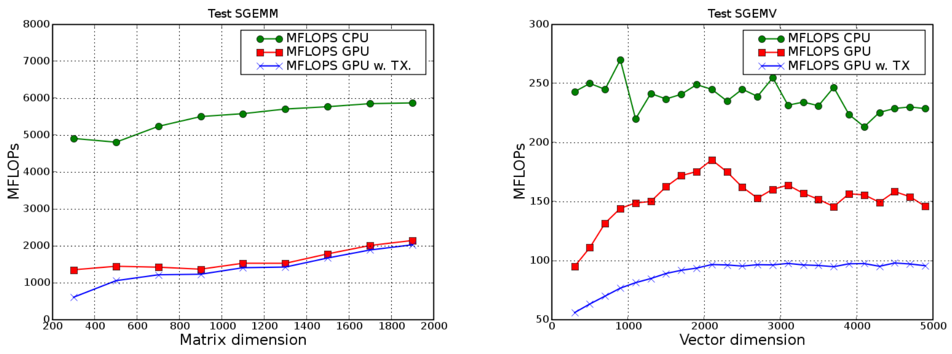

Fig. 3. Performance of routines `SGEMM` (left-hand side) and `SGEMV` (right-hand side) on the *AMD AthlonXP 2400+* CPU and the *Nvidia NV44* GPU.

The figure also reports that the impact of the data transference, however, is less important for routine `SGEMM`, which carries out a higher computational load per element that is transferred through the bus. From the previous results, it is possible to conclude that the amount of data reutilization is an important factor in order to achieve high performance on a GPU.

Computation load per stream element: BLAS-1 routines

Therefore, one could expect that BLAS-1 operations (`SAXPY` and `SSCAL`) will deliver a high MFLOPs rate on this class of hardware. Surprisingly, as shown in Figure 4, we get a poor performance for our implementations of `SAXPY` and `SSCAL`, much lower than those of the corresponding CPU implementations.

This behavior can be explained as follows: the scarce amount of computational load per memory access in BLAS-1 operations limits their performance. This is partially due to the lower efficiency of the memory system of the *Nvidia NV44* GPU, with a poor use of cache memories. The elaborated cache memory of the CPU, and its efficient use by the optimized routines in Goto BLAS, are the reasons for such a notable difference in efficiency. Furthermore, results on the GPU are slightly better for `SAXPY` when compared with the corresponding implementation on CPU than for `SSCAL`, as the computational load per stream element calculated in the former operation is twice as high as that of `SSCAL`.

In conclusion, high computational load per stream element is one of the basic conditions for an algorithm to deliver high performance when executed on GPU.

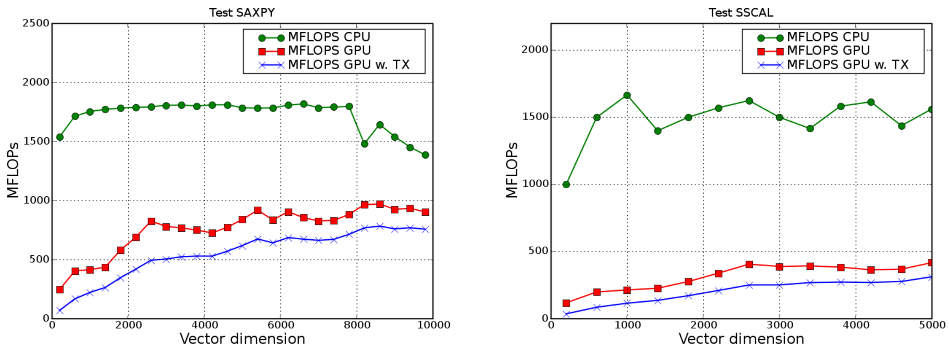

Fig. 4. Performance of routines `SAXPY` (left-hand side) and `SSCAL` (right-hand side) on the *AMD AthlonXP 2400+* CPU and the *Nvidia NV44* GPU.

Bidimensional convolution filters

Convolution filters combine in the same operation a set of very favorable properties for GPUs: high data parallelism, low input data reutilization, and high computational load per stream element. Figure 5 shows the results of the implementations of the convolution filter on the CPU and GPU. The optimized implementation on GPU (labeled as "GPU4") employs the four channels of each element of the input stream in order to store data (as explained at the end of Section 4), attaining a speed-up factor close to 4x with respect to a basic GPU implementation (labeled as "GPU"). The comparison between this implementation and the optimized CPU version shows a comparable performance between the optimized CPU implementation and the optimized GPU one.

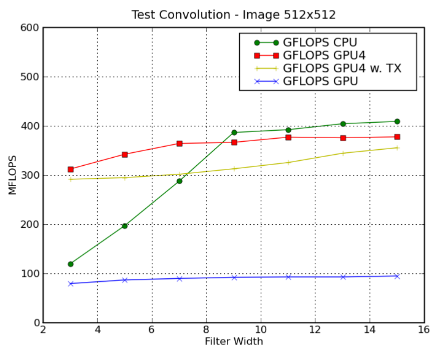

Fig. 5. Performance of the implementations of the convolution filter on the *AMD AthlonXP 2400+* CPU and the *Nvidia NV44* GPU.

Convolution filters are the type of algorithms that better fit into the execution model of GPUs with classical architecture. These operations exhibit all the properties that make good use of GPUs computational power and, at the same time, hide those aspects in which CPUs are better than graphics processors (basically at memory access).

Impact of data transfers

From the empirical results, it is possible to conclude that the data transfer stage previous to any operation executed on GPU is a penalty to the final performance, although the overhead introduced is not critical. For this generation of GPUs, the AGP port is not a significative bottleneck for the overall computation process. In fact, the impact of data transfer is minimal for those routines in which the computational load per transferred element is high, e.g. matrix-matrix multiplication or convolution.

5 New Generation GPU-CPU Comparison

5.1 Comparison goals

Although the study of the non-unified generation of GPUs has identified some of the characteristics desirable in algorithms that target GPUs with classical architecture, it is also interesting to carry over this study to new generation GPUs with unified architecture. The goal of this study is to verify if our previous insights also hold for these new architectures, and to evaluate how the hardware and software improvements (at computational power, memory hierarchies and interconnection buses level) affect the performance of the implemented routines.

5.2 Experimental setup

In this second set of experiments, we again chose two experimental platforms from the same generation, an *Intel Core 2 Duo* CPU and a *Nvidia GeForce 8800 Ultra* (with a *Nvidia G80* processor) GPU (year 2007); see Table 2 for details. The GNU `gcc` 4.1.2 compiler is employed in the evaluation. The multithreading capabilities of Goto BLAS were enabled so that the two cores in the Intel CPU cooperate in solving the linear algebra operations.

The implementations of the linear algebra routines in the the CUBLAS library ([2]) were used in the evaluation. This is a library developed by Nvidia, implemented on top of CUDA, and optimized for unified graphics architectures as the *Nvidia G80*. The experimental evaluation showed that the implementations in CUBLAS outperformed our implementations using Cg.

	CPU	GPU
Processor	*Intel Core 2 Duo*	*Nvidia GeForce 8800 Ultra*
Codename	Crusoe E6320	G80
Clock frequency	1.86 GHz	575 MHz
Peak performance	14.9 GFLOPS	520 GFLOPS
Memory speed	2×333 MHz	2×900 MHz
Bus width	64 bits	384 bits
Max. bandwidth	5.3 GB/s	86.4 GB/s
Memory	1024 MB DDR	768 MB DDR
Bus	PCI Express x16 (4 GB/s transfer rate)	
Year	2007	2007

Table 2. Description of the hardware used in our second experimental study.

For the convolution filter, we implemented a tuned version using CUDA, with intensive use of the fast shared memory per group of SP in order to optimize performance. We also applied other optimization guidelines proposed in [3], and common in the CUDA programming paradigm. On the CPU side, an optimized, BLAS-based implementation of the bidimensional convolution filter was used. This type of implementation is fully optimized with respect to the memory and SSE unit, so the comparison is considered to be fair.

5.3 Experimental results

Input data reutilization: `SGEMM` vs. `SGEMV`

Figure 6 (left-hand side) shows the performance of routine `SGEMM` on both platforms. Although this is not the most appropriate algorithm for the GPU (indeed, it only delivers about 20% of the peak power of the GPU), the performance on that platform is roughly 10 times higher than that obtained on the CPU.

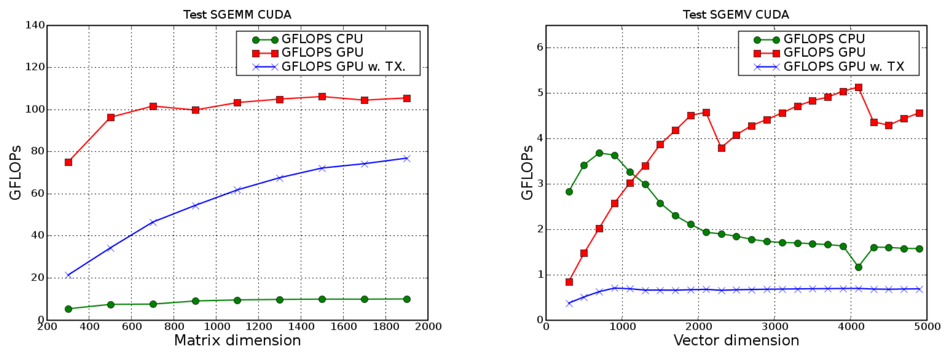

Fig. 6. Performance of routines `SGEMM` (left-hand side) and `SGEMV` (right-hand side) on the *Intel Core 2 Duo* CPU and the *Nvidia G80* GPU.

The impact of the data transfer bottleneck in the performance of a routine is higher when its computational load decreases. For example, Figure 6 (right-hand side) illustrates the performance of routine `SGEMV`. The decrease in the GFLOPS rate is higher in this case when the transmission time is included. This difference is so important for this routine that, in case the transfer time is considered in the evaluation, the performance is lower on the GPU than on the CPU. When transfer times are not considered, the implementation on the CPU outperforms the CUBLAS implementation for large stream dimensions.

Note the different behavior of the `SGEMV` routine executed on CPU and on GPU. While on a general purpose processor the peak performance is achieved for relatively small amounts of data, obtaining worse results for bigger vectors, the maximum performance on GPU is always attained for large vectors. In fact, the optimized implementations of the BLAS, such as GotoBLAS, exploit very well the sophisticated cache systems of the most modern processors, and thus benefits the computation with small matrices. On the other hand, GPUs do not present such advanced cache memories; this fact, and the stream-oriented architecture of this class of processors, benefit the computation over big amounts or streams of data, attaining poor results for small vectors.

Comparing routines `SGEMM` and `SGEMV`, the introduction of a sophisticated memory hierarchy in the *Nvidia G80* diminishes the impact of the data reutilization. The results for routine `SGEMM` are better when we compare them with CPU implementation than the results we obtain for routine `SGEMV`. The introduction of cache memories is one of the main differences between both generations of GPU and, from the previous results, we can conclude it has an important influence in the performance of general-purpose algorithms on GPUs with unified architectures.

Computational load per stream element: BLAS-1 routines

The amount of computational load per stream element is also critical in this class of architectures. Figure 7 reports the results for routines `SAXPY` and `SSCAL`. Compared with the results attained for the classical architectures in Figure 4, although being better in absolute terms, the behaviors are similar: despite being stream-oriented algorithms, without any type of input data reutilization, the results are not comparable with those obtained by the tuned implementations in GotoBLAS. As occurred in previous experiments, results are better for a more computationally intensive algorithm such as `SAXPY`, attaining better results than `SSCAL`. The transfer time is more relevant in this case, as the computational load of the algorithms is quite low compared with that on more computationally intense routines, such as `SGEMM`.

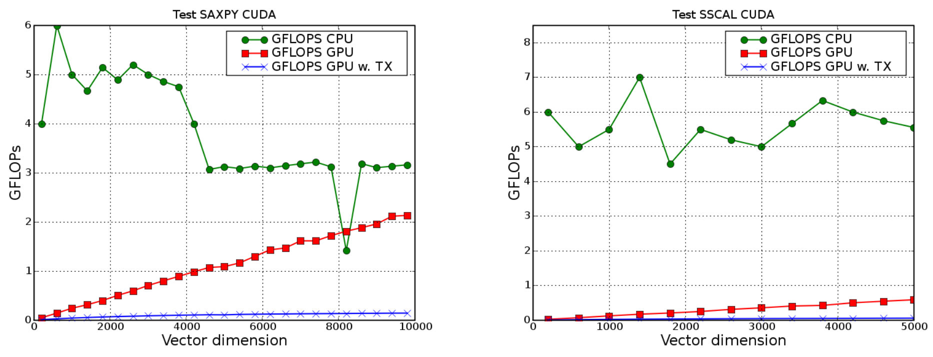

Fig. 7. Performance of routines `SAXPY` (left-hand side) and `SSCAL` (right-hand side) on the *Intel Core 2 Duo* CPU and the *Nvidia G80* GPU.

Bidimensional convolution filter

Figure 8 shows the results obtained for the application of a convolution filter on a 512×512 image and variable filter size. This application again presents the most favorable properties for its execution on current GPU architectures, attaining results up to 20 times better than those achieved for the same routines on a CPU. This is, in fact, the highest speedup achieved in our study.

Impact of data transfers

Data transfers were not a critical stage for the past generation GPUs. However, from the empirical results extracted for the most modern generation of graphics processors, we have proved that communication through the PCIExpress bus is now a factor to be considered.

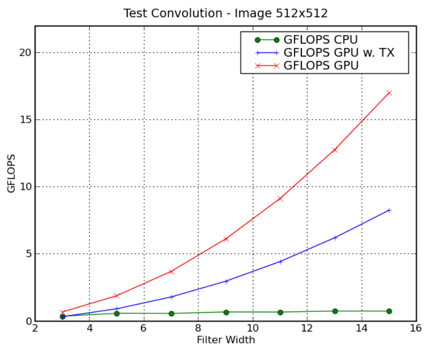

Fig. 8. Performance of the implementations of the convolution filter on the *Intel Core 2 Duo* CPU and the *Nvidia G80* GPU.

The impact of the transfer time is larger for the unified architecture compared with non-unified architecture, with less powerful interconnection buses. In fact, the peak performance of the *Nvidia G80* is about 20 times higher than that of the *Nvidia NV44*, but the speed of the interconnection bus in the unified platform is only twice as fast as the one in the non-unified platform. This is a major bottleneck in current graphics platforms, and determines that GPU algorithms must be redesigned to reduce the communications so that data in video memory is reused as much as possible before sending them back to RAM.

The latest GPU models from NVIDIA support the overlapping between memory transfer and computation on GPU, making it possible to hide the data transfer bottleneck for some operations. Unfortunately, the tested GPU did not support this feature.

6 Conclusions

We have presented a study of the properties which favor efficient execution of general-purpose algorithms on a graphics processor, considering both classical and unified architectures.

GPUs from previous generations, with classical architecture, are suitable for certain types of general-purpose algorithms with three basic characteristics: low input data reutilization, high data level parallelism, and high computational load per stream element. Despite their high computational power, the graphics-oriented nature of this class of hardware carries a set of limitations at the architecture level which ultimately limit the performance of certain types of algorithms like, e.g., routines from BLAS. On the other hand, GPUs of this nature obtain remarkable results for general-purpose algorithms which exhibit the three properties specified above, outperforming in this case the CPU.

The improvements introduced in the new generation of GPU (unified architecture, higher processing units replication, more sophisticated memory hierarchies, etc.) have increased the efficiency of this hardware to execute also for general-purpose algorithms. In fact, current GPUs deliver higher performance than that of timely CPUs in many applications.

Therefore, the last generation of GPUs appears as a high performance and low cost co-processing platform for a larger variety of applications. The emergence of general-purpose languages that facilitate their programming makes them even more interesting hardware from general-purpose computations. Nevertheless, GPUs still present some limitations in general-purpose computing such as numerical precision, data transfer stages, memory hierarchies not as sophisticated as CPU ones, etc. All this makes necessary to evaluate carefully the suitability of GPU as an accelerator for calculations.

Acknowledgments

This work has been supported by the CICYT project TIN2005-09037-C02-02 and FEDER. Francisco Igual-Peña is supported as well by a research fellowship from the *Universidad Jaume I of Castellón* (PREDOC/2006/02).

References

1. S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Workshop on Multithreaded Architectures and Applications, MTAAP 2008.*
2. NVIDIA Corp. *NVIDIA CUBLAS Library.* 2007.
3. NVIDIA Corp. *NVIDIA CUDA Compute Unified Device Architecture. Programming Guide.* 2007.
4. K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. *Graphics Hardware*, 2004.
5. Basic Linear Algebra Subprograms Technical (BLAST) Forum. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard.* 2001.
6. N. Galoppo, N. Govindaraju, M. Henson, and D. Monocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *ACM/IEEE SC/05 Conference*, 2005.
7. K. Goto and R. Van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software.*
8. N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 215–226, June 2004.
9. J.Y. Hong and M.D. Wang. High speed processing of biomedical images using programmable GPU. In *Image Processing, 2004. ICIP '04. 2004 International Conference on*, volume 4, pages 2455 – 2458 Vol. 4, 24-27 Oct. 2004.
10. E.S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 43 – 43, Nov. 2001.
11. A. Moravánszky. Dense matrix algebra on the GPU. 2003.
12. A. Ruiz, O. Sertel, M. Ujaldon, U. Catalyurek, J. Saltz, and M. Gurcan. Pathological image analysis using the GPU: Stroma classification for neuroblastoma. *Proceedings IEEE Intl. Conference on BioInformation and BioMedicine*, 2007.