Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors

Sergio Barrachina Maribel Castillo Francisco D. Igual Rafael Mayo
Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
{barrachi,castillo,figual,mayo,quintana}@icc.uji.es

Abstract

The increase in performance of the last generations of graphics processors (GPUs) has made this class of platform a coprocessing tool with remarkable success in certain types of operations. In this paper we evaluate the performance of the Level 3 operations in CUBLAS, the implementation of BLAS for NVIDIA® GPUs with unified architecture. From this study, we gain insights on the quality of the kernels in the library and we propose several alternative implementations that are competitive with those in CUBLAS. Experimental results on a GeForce 8800 Ultra compare the performance of CUBLAS and the new variants.

Keywords: Graphics processors, linear algebra, BLAS, high performance.

1. Introduction

Dense linear algebra operations lie at the heart of many scientific and engineering applications. The interest of the scientific community to solve larger or more complex numerical problems, where the computation time is often the limiting factor, naturally leads to the need of attaining high performance on whatever architectures are the state-of-the-art.

In this paper we evaluate the implementation of the *Basic Linear Algebra Subroutines* (BLAS) provided in CUBLAS 1.0 [9]. This is a library implemented on top of the NVIDIA® CUDA™ (compute unified device architecture) [10]. Our evaluation is focused on the kernels of the Level 3 BLAS, which are often used to perform large numbers of arithmetic operations, and are thus natural candidates for execution on graphics processors. The target architecture is the *GeForce 8800 Ultra*. Several previous studies have evaluated the performance of tuned implementations of the Level 3 BLAS constructed using graphics application programming interfaces (APIs) [8, 3, 7]. However, the recent development of CUBLAS and the fast evolution of graphics hardware renews the interest in evaluating the performance of these operations on new generation hardware.

The results of the evaluation demonstrate that not all kernels in CUBLAS are equally tuned. Therefore, as part of our work we also propose several variants of the kernels that improve the performance of the basic implementations in CUBLAS. In addition, we propose a hybrid parallel algorithm that splits the computation between the CPU and the GPU, achieving good performance results.

The rest of the paper is structured as follows: Section 2 introduces the unified architecture of current GPUs and the CUDA API. Section 3 explains the importance of the performance of BLAS implementations, and introduces CUBLAS. Section 4 evaluates the implementation of the Level 3 BLAS kernels in CUBLAS. In Section 5, we propose several improvements over CUBLAS and report the performance gains. Finally, Section 6 summarizes the conclusions and outlines future work.

2. GPUs with unified architecture

In 2006 a generation of GPUs with a completely different architectural design appeared which solved many of the restrictions related with general purpose computation that were present in previous generations of graphics processors. These new GPUs feature a *unified architecture*, with one processing unit or unified shader that is able to work with any type of graphical data, transforming the sequential pipeline of previous GPUs into a cyclic pipeline.

In particular, there are several characteristics in the new generation of GPUs which favour its use as a general-purpose coprocessor:

1. In general, the clock frequency of the unified shader is much higher than that of the fragment processors present in previous GPUs (though still much lower than the clock frequency of current CPUs).

2. The shader consists of a large collection of computation units (up to 128, depending on the GPU version), called Streaming Processors (SP), which operate in clusters of 16 processors in SIMD mode on the input data stream.
3. The memory hierarchy is much more sophisticated, and includes a L2 cache and small fast memories shared by all the SP in the same cluster.

Altogether with these unified architecture, the CUDA general-purpose API [10] has been developed to exploit the potential computational power that this hardware offers. In fact, CUDA has been proposed as a standard (although only compatible with NVIDIA hardware so far) to program the new generation of GPUs, without the requirement of learning more complex graphics-oriented languages.

3. BLAS and CUBLAS

The BLAS are a collection of kernels that provide standard building blocks for performing basic vector and matrix operations. Level 1 BLAS perform scalar, vector and vector-vector operations; Level 2 BLAS perform matrix-vector operations; and Level 3 BLAS perform matrix-matrix operations. Highly efficient implementations of BLAS exist for most current computer architectures and the specification of BLAS is widely adopted in the development of high quality linear algebra software, such as LAPACK and FLAME [1, 2].

The Level 3 BLAS are specially important as the performance of more complex routines that employ them directly depends on that of the underlying BLAS implementation. Level 3 BLAS is basically formed by five kernels: GEMM (matrix multiplication), SYMM (symmetric matrix multiplication), SYRK (symmetric rank-k update), TRSM (triangular system solve with multiple right-hand sides), and TRMM (triangular matrix multiplication). Among these, in our evaluation we select GEMM, SYRK, and TRSM. The two other kernels are quite similar to SYRK, and therefore we expect the result of our analysis to apply to SYMM and TRMM as well.

CUBLAS [9] is an implementation of BLAS developed by NVIDIA on top of the CUDA driver. CUBLAS provides functions for creating/destroying matrix and vector objects in GPU memory space, filling them with data, executing BLAS on the GPU, and transferring data back to main memory. Thus, CUBLAS offers basic BLAS functions as well as helper functions for writing data to and retrieving data from the GPU memory. As current GPUs only support single precision arithmetics, no double precision version of the kernels has been implemented in CUBLAS.

The following code illustrates how easy it is to use CUBLAS from a C/C++ program to scale a vector:

```
1  int main( void ){
2   ...
3   float* host_vector, * device_vector;
4
5   host_vector = (float*) malloc(M*sizeof(float));
6
7   ... // Initialize vector of M floats
8   cublasAlloc(M, sizeof(float),
9               (void**) &device_vector);
10
11  cublasSetVector(M, sizeof(float), host_vector,
12                  device_vector, 1);
13  cublasSscal(M, ALPHA, device_vector, 1);
14  cublasGetVector(M, sizeof(float), device_vector,
15                  host_vector, 1);
16
17  cublasFree(device_vector);
18  ...
19 }
```

Lines 8 and 21 initialize and terminate the CUBLAS environment much in the style of packages like MPI. Lines 10-11 and 19 allocate and free space for the vector in the GPU memory. Lines 13-14 and 16-17 move the data from the main memory to the GPU memory and retrieve the results. Finally, the call in line 15 scales the contents of the vector using the GPU hardware.

CUBLAS also provides wrappers to help writing Fortran programs that use the library.

4 Evaluation of the Level 3 CUBLAS

We first evaluate the performance of the Level 3 BLAS implementation of CUBLAS on a GPU with unified architecture. Detailed specifications of the hardware of the system can be found in Table 1.

	CPU	GPU
Processor	Intel Core 2 Duo	NVIDIA 8800 Ultra
Codename	Crusoe E6320	G80
Clock frequency	1.86 GHz	575 MHz
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 DDR2	768 MB GDDR3
Bus	PCI Express x16 (4 GB/s)	

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

The Linux implementations of CUDA and CUBLAS version 1.0 were used in the evaluation of the GPU together with the compiler `nvcc` release 1.0, version 0.2.1221. The implementation of BLAS in GotoBLAS [4] version 1.19 was used on the CPU and code in this platform was compiled using `gcc` version 4.1.2.

All the results on the GPU hereafter include the time required to transfer the data from the main memory to the

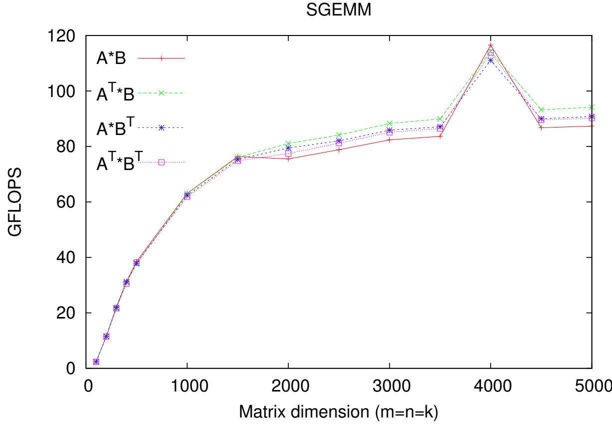

Figure 1. Performance evaluation for the implementation of SGEMM in CUBLAS

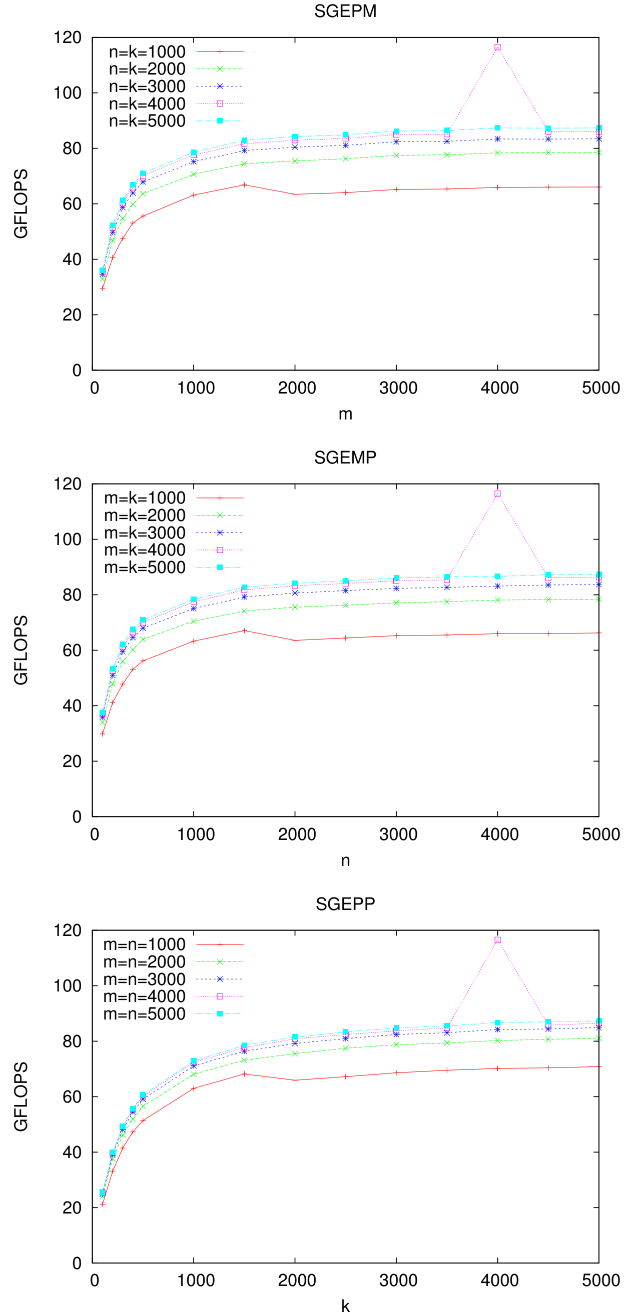

Figure 2. Performance evaluation for the "implementations" of SGEPM (top), SGEMP (middle), and SGEPP (bottom) in CUBLAS, used to compute the matrix multiplication $C := C + AB$; two dimensions fixed

GPU memory and retrieve the results back. The kernels all operate on single-precision real data and results are reported in terms of GFLOPS (10^9 floating-point arithmetic operations per second). A single core of the Intel processor was employed in the experiments.

4.1 Evaluation of SGEMM

The SGEMM kernel in BLAS can be used to compute any of the following four matrix multiplications

$$C \quad := \quad \beta \cdot C + \alpha \cdot op(A) \cdot op(B),$$

where $op(X) = X$ or X^T, and α, β are both scalars. C is $m \times n$, $op(A)$ is $m \times k$, and $op(B)$ is $k \times n$.

Our first experiment evaluates the performance of the implementation of the SGEMM kernel in CUBLAS for square matrices A, B, and C (i.e., $m = n = k$), and all transpose combinations of the operands. The results in Figure 1 shows that the layout in memory of the matrices has little effect in the performance of the kernel. It is also interesting to note the much higher performance of the kernel when $m = 4000$. Further experiments revelated that all Level 3 CUBLAS kernels share this behaviour on the GeForce 8800 Ultra when the dimensions of the matrices are a multiple of 32. These particular values are likely to allow the kernel to present an optimal memory access pattern by correctly aligning data in memory, as suggested in [9]. "Optimal" dimensions will probably vary with the specific GPU model. This insight leads to our proposal to improve the performance of the SGEMM kernel, as well as other kernels such as SSYRK or STRSM, described in subsection 5.1.

Following the characterization of the matrix multiplication in [5], we next analyze the performance of this operation when one of the matrix dimensions (m, n, or k) is

small with respect to the other two. This gives us three different kernels: SGEPM (m is small), SGEMP (n is small), and SGEPP (k is small). Figure 2 shows that the difference in performance among the three kernels is not significant. (Strictly speaking, in the experiment two of the dimensions are fixed and the other one varies, outgrowing the other two; this does not correspond exactly to the definition of the previous kernels). We note again the performance boost in all three kernels when operating with matrices of dimensions that are a multiple of 32.

4.2 Evaluation of SSYRK

The SSYRK kernel computes

$$C := \beta \cdot C + \alpha \cdot A \cdot A^T, \text{ or}$$
$$C := \beta \cdot C + \alpha \cdot A^T \cdot A,$$

where C is an $m \times m$ symmetric matrix, A is $m \times k$ ($k \times m$ if transposed in the operation), and $\alpha,\ \beta$ are scalars. Given the symmetry of the result, only the lower or upper triangular part of C is computed. The performance of this kernel is specially relevant due to its impact on other procedures, such as Cholesky factorization.

Figure 3 reports the results for the CUBLAS implementation of SSYRK when used to update the lower or upper triangular part of C adding $A \cdot A^T$ or $A^T \cdot A$ to it, and with $m = k$. As was the case for SGEMM, differences between the different variants are negligible. Note the much lower performance of the SSYRK implementation when compared to that of SGEMM (at most, 40 GFLOPS for SSYRK compared with 120 GFLOPS for SGEMM). In subsection 5.3, we improve this performance by building SSYRK on top of SGEMM.

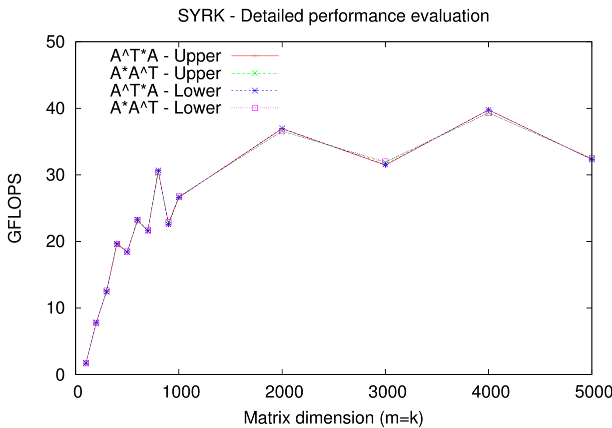

Figure 3. Performance evaluation for the implementation of SSYRK in CUBLAS

Figure 4 illustrates the performance of SSYRK for various dimensions of m and varying values for k. Again, there is a significant increase in performance for matrix dimensions that are a multiple of 32 (m=4000).

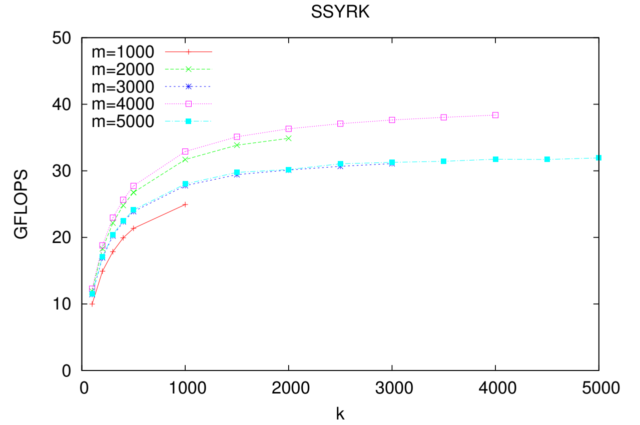

Figure 4. Performance evaluation for the implementation of SSYRK in CUBLAS, used to compute the upper triangular part of the results of the symmetric rank-k update, $C := C + AA^T$; m fixed

4.3 Evaluation of STRSM

The last kernel that is included in this study, STRSM, can be used to solve the triangular linear systems:

$$op(A) \cdot X = \alpha \cdot B, \text{ or}$$
$$X \cdot op(A) = \alpha \cdot B,$$

where $op(A) = A$ or $op(A) = A^T$. The unknown matrix X and B are $m \times n$, and the (upper/lower) triangular matrix $op(A)$ is $m \times m$ or $n \times n$ depending, respectively, on whether it appears to the left or the right of the unkwown matrix. On completion, the solution X overwrites B.

Figure 5 shows the performance of the CUBLAS implementation of the kernel when solving the equations with the two possible combinations of memory layouts of matrix A (transpose/no transpose), shapes (upper/lower triangular) and $m = n$. The results are very similar for the majority of the combinations, with the exception of the case in which A is a lower triangular matrix and it is not transposed. In this case, performance is significantly lower.

5 Tuning of CUBLAS

This section introduces some improvements to the kernels in CUBLAS. We can distinguish three types of optimizations: application of padding, implementation of Level

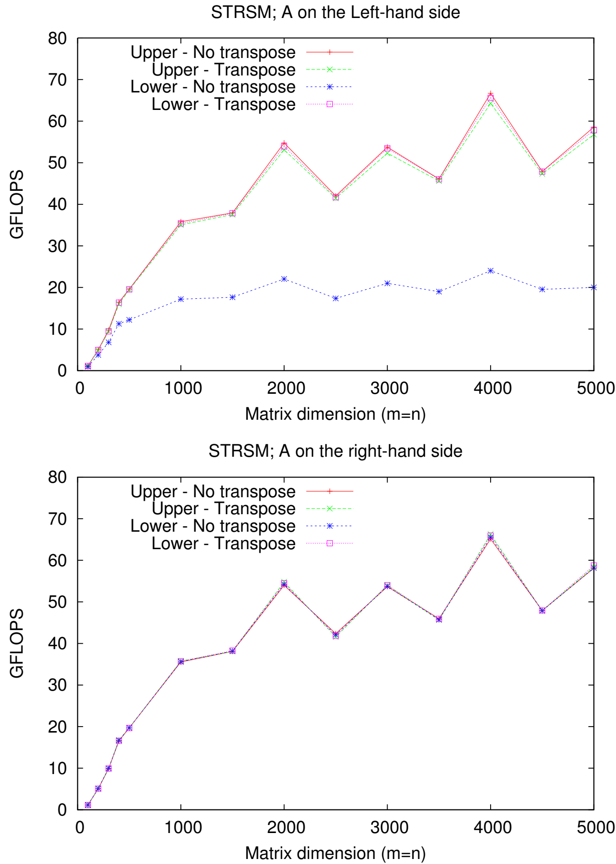

Figure 5. Performance evaluation for the implementation of STRSM in CUBLAS with A to the left-hand (top) or right-hand (bottom) side of the unknown matrix

3 BLAS kernels on top of SGEMM, and hybrid approaches that split computations between the CPU and GPU.

5.1 Padding for SGEMM and SSYRK

One of the observations from the initial evaluation of the kernels in CUBLAS was the superior performance when these operate on matrices that are a multiple of 32. According to this, the first improvement introduced is *padding*. Thus, our proposal for SGEMM and SSYRK is to pad with zeros the input matrices, transforming their dimensions into the next multiple of 32. With this transformation, we introduce a very small overhead in the computation of the kernel, negligible for large matrices, as the dimensions are most increased in 31 columns/rows.

The implementation creates and sets to zeros a padded matrix in GPU memory for each operand matrix, and then transfers the data from main memory to the correct position

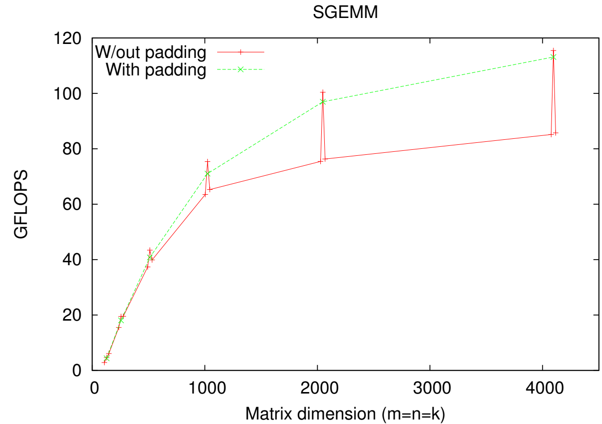

Figure 6. Performance evaluation for the implementation of SGEMM in CUBLAS with and without padding

in GPU memory.

In Figure 6 we compare the performances of the original CUBLAS SGEMM kernel and the modified kernel with padding applied on matrices. Results are reported for matrices of dimension $m = n = k = 2^i$ and $2^i - 1$, $i = 7, 8, \ldots, 12$. As a result of the application of padding, the performance attained by the kernel with padding is uniform for all matrix sizes, hiding the irregular performance of original CUBLAS implementation. There is some overhead associated with the cost of the padding process and the non-contiguous store of the data in GPU memory during the transference of the matrices; however, its impact over the whole process is small, and the improvement when operating with non-multiple of 32 dimensions is important.

5.2 Partitioning for larger matrices

Transfer times between GPU memory and main memory is one of the bottlenecks of current GPUs. Therefore, overlapping computation and transfers could imply better performance. We have implemented a *blocked version* of SGEMM that allows to overlap the computation of the partial multiplication $A_p \cdot B_p$ (with A_p and B_p being, respectively, a block of columns of A and a block of rows of B) and the transference of the next pair of blocks A_{p+1} and B_{p+1}.

Unfortunately, the current version of CUDA is unable to overlap computation and communication. The benefits of this approach, however, will be exploited with future versions of CUDA and Nvidia hardware, that will allow simultaneous transfers and computation on the GPU.

An orthogonal (independent) benefit of this approach is that the amount of GPU memory needed to compute the matrix multiplication is more reduced ($mn + mb + bn$ num-

bers, with b the column/row size of blocks A_p/B_p, compared with $mn + mk + kn$). This enables the computation with larger matrices that do not fit in GPU memory.

5.3 SSYRK built on top of SGEMM

The evaluation of the SSYRK kernel in CUBLAS in subsection 4.2 shows a poor performance compared with that of the SGEMM implementation. Following the idea from [6], it is possible to transform part of the computations performed by SSYRK into SGEMM calls, as we describe next. Consider, e.g., the partitioning of the matrices in Figure 7, where C_{11} is $b \times b$ and A_1 consists of b rows. Assuming that the first block of columns of C has already been computed, in the column-oriented version of the algorithm, we proceed by computing the following operations in the current iteration:

$$C_{11} := \beta \cdot C_{11} + \alpha \cdot A_1 \cdot A_1^T \quad \text{SSYRK}$$
$$C_{21} := \beta \cdot C_{21} + \alpha \cdot A_2 \cdot A_1^T \quad \text{SGEMM}$$

or, in the row-oriented version, considering updated the first block of rows of C:

$$C_{11} := \beta \cdot C_{11} + \alpha \cdot A_1 \cdot A_1^T \quad \text{SSYRK}$$
$$C_{10} := \beta \cdot C_{10} + \alpha \cdot A_1 \cdot A_0^T \quad \text{SGEMM}$$

After these operations, the computation proceeds by updating the next block of columns (or rows) of C. By computing C by blocks of b columns (or rows), where at each step the diagonal $b \times b$ block is computed using the SSYRK kernel and the off-diagonal block is computed using the SGEMM kernel, it is possible to exploit the higher performance of the CUBLAS kernel for the matrix multiplication, and speed up the computation of the SSYRK operation.

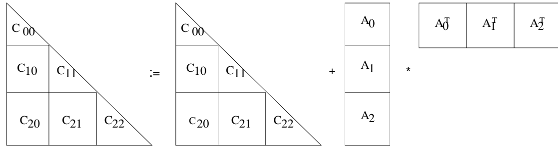

Figure 7. Decomposition of the SSYRK operation to build it on top of SGEMM

Figure 8 shows a comparison between the SSYRK implementation in CUBLAS and our *column-oriented blocked implementation* built on top of SGEMM, with and without padding. The row-oriented implementation presents similar behavior, so it is not shown in the figure. The performance of the blocked implementation is still limited by the CUBLAS SSYRK implementation that is employed to compute the diagonal blocks, but results are closer to those of SGEMM.

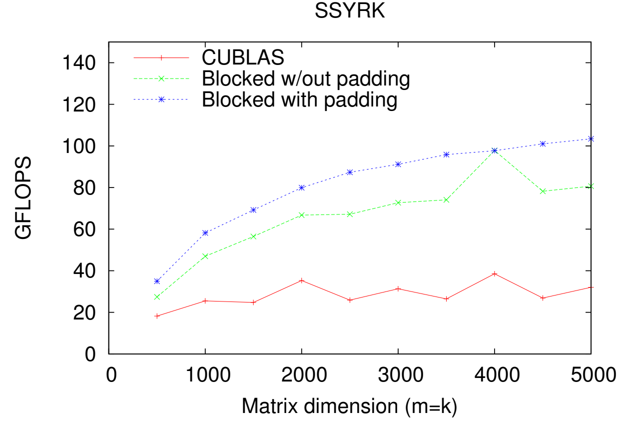

Figure 8. Performance evaluation for the implementation of SSYRK built on top of SGEMM

5.4 STRSM built on top of SGEMM

A blocked algorithm can also be derived for the solution of $A \cdot X = \alpha \cdot B$, with A lower triangular, as follows. Consider the partitioning of the operation in Figure 9, where A_{11} is $b \times b$ and both X_1 and B_1 consist of b rows, and assume that X_0 has already been computed and the corresponding updates of B_1 and B_2 have been performed. Then, during the current iteration, in the column-oriented version of forward-substitution we proceed by computing:

$$A_{11} \cdot X_1 = \alpha \cdot B_1 \quad \text{STRSM}$$
$$B_2 := B_2 - A_{21} \cdot X_1 \quad \text{SGEMM}$$

while, in the row-oriented version (assuming that X_0 has been computed) we need to compute:

$$B_1 := B_1 - A_{10} \cdot X_0 \quad \text{SGEMM}$$
$$A_{11} \cdot X_1 = \alpha \cdot B_1 \quad \text{STRSM}$$

The computation proceeds with the next $b \times b$ diagonal block in A and block of b rows in X/B.

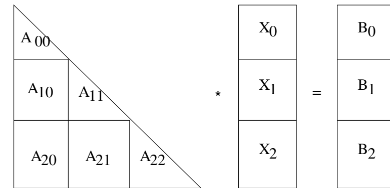

Figure 9. Decomposition of the STRSM operation to build it on top of SGEMM

We have implemented both versions of the STRSM kernel built on top of SGEMM. Figure 10 shows the performance

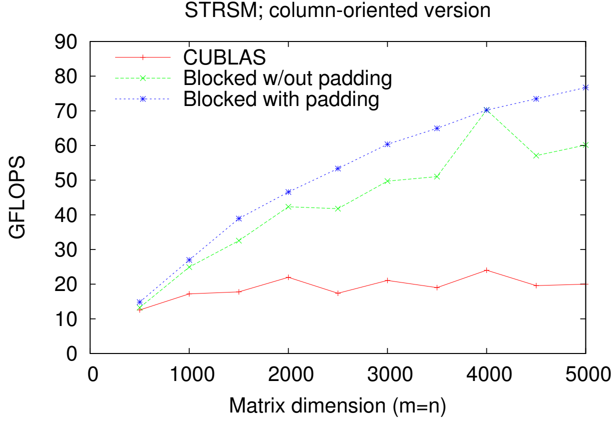

Figure 10. Performance evaluation of the column-oriented implementation of STRSM built on top of SGEMM

observed for our *column-oriented blocked implementation*. The row-oriented implementation presents a similar behavior. The figure also includes the results attained by applying padding to the SGEMM suboperations. Again there is a remarkable increase in performance by utilizing the SGEMM to compute the STRSM operation.

5.5 Hybrid implementation for SGEMM

It is possible to design an implementation in which CPU and GPU collaborate to obtain a common result. To assess the benefits of this, we have implemented a *hybrid implementation* to compute the matrix multiplication $C := \alpha A \cdot B$, but the technique is easily ported to other variants of the matrix multiplication or other kernels like, e.g., SSYRK or TRSM.

Consider the partitioning in Figure 11. There, matrix B is splitted into two column blocks, B_1 $(M \times N')$ and B_2 $(M \times N'')$. Block B_1, together with matrix A, is transferred to GPU memory and the GPU performs the operation $C_1 := \alpha \cdot A \cdot B_1$, while CPU performs the operation $C_2 := \alpha \cdot A \cdot B_2$. When GPU finishes, submatrix C_1 is transferred back to main memory, and the operation is complete.

Our implementation executes in parallel these two operations, improving the performance of the overall SGEMM kernel. Values for N' and N'' must be selected carefully in order to balance the load in a proper way between the GPU and CPU.

For our hardware configuration, experimental results determine that N' should be 7 times larger than N''. This factor has a direct impact on the performance that is possible to achieve. As illustrated in Figure 12, the results improve

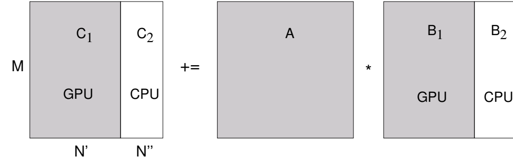

Figure 11. Decomposition of matrix multiplication for a hybrid CPU-GPU approach

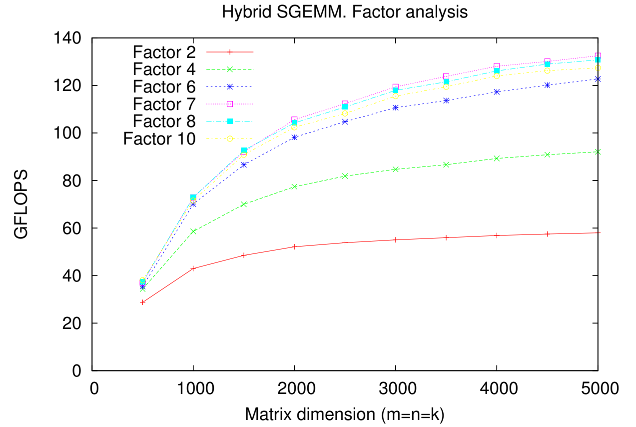

Figure 12. Impact of the partition sizes over the performance of the hybrid SGEMM

when the factor is increased, until it reaches the factor 7. For smaller factors, the CPU takes too long to perform its calculation, and the load is not well balanced. For larger factors, the performance obtained is also suboptimal, as the CPU stays idle while the GPU is still working. However, the decrease in performance in this case is not as important, since the CPU is less powerful, and its impact is less important over the whole process.

The use of GPU as a coprocessor with a hybrid approach leads to a significant improvement in performance. Figure 13 compares the performance of the hybrid implementation (including padding), the original CUBLAS SGEMM kernel, and the CUBLAS SGEMM kernel with padding. The figure summarizes the results which can be achieved with relatively simple transformations over default CUBLAS implementations.

5.5.1 Variants of the hybrid algorithm

We have implemented two variants of the hybrid algorithm. The first option creates a thread before the transfer of matrices A and B_1 begins. Thus, CPU computation overlaps not only with GPU computation, but also with transfers between GPU memory and main memory. The second option

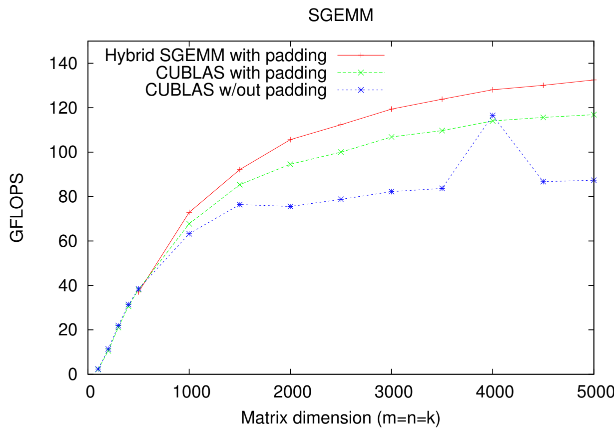

Figure 13. Performance evaluation for the implementation of SGEMM in CUBLAS, the implementation of SGEMM with padding, and the hybrid implementation of SGEMM (with padding)

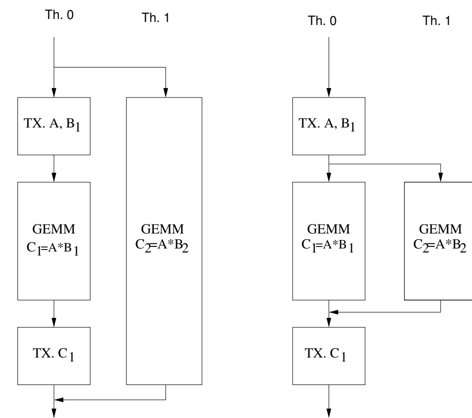

Figure 14. Two thread schemes implemented for hybrid SGEMM

is slightly different: in the first place, there is an initial transfer of matrices A and B_1; after finishing the transfer, a new thread is created, and GPU and CPU perform their part of the matrix multiplication. Only when both processors have finished, the transfer of C_1 back to main memory can take place. Figure 14 illustrates both algorithms.

However, the current version of CUDA does not allow to overlap transfers and computation in both CPU or GPU. This limitation yields the second variant the most suitable one in order to attain the best performance. The above results for the hybrid algorithm are thus based on this variant.

6. Conclusions

Graphics processors are becoming a cheap and efficient alternative to solve general-purpose compute-intensive applications. The appeal of these platforms is considerably increased by the development of high-level APIs for their use. In this paper we have evaluate one of these APIs, CUBLAS, which faciliates the computation of dense linear algebra operations on NVIDIA GPUs. Our study reveals that not all kernels in the Level 3 CUBLAS are equally optimized, and that a much higher performance can be extracted from the hardware by using a simple technique such as padding or building the kernels around the GEMM operations. Our experiments also demonstrate that hybrid algorithms, which split the computation between the CPU and the GPU, can increase the performance of an "pure" GPU implementation. Two major drawbacks still remain for GPUs: the lack of support for double-precision arithmetic and the subtle differences between GPU and IEEE arithmetics.

Acknowledgments

This research was sponsored by the CICYT project TIN2005-09037-C02-02 and FEDER, and project No. P1B2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI. Francisco D. Igual is supported as well by a research fellowship from the UJI (PREDOC/2006/02).

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, Mar. 2005.

[3] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. *Graphics Hardware*, 2004.

[4] K. Goto. Goto BLAS implementation, 2005.

[5] K. Goto and R. A. Van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 2006.

[6] B. Kågström, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, Sept. 1998.

[7] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 43 – 43, Nov. 2001.

[8] A. Moravánszky. Dense matrix algebra on the GPU. 2003.

[9] NVIDIA. *CUBLAS Library*. 2007.

[10] NVIDIA. *Nvidia CUDA Compute Unified Device Architecture. Programming Guide*. 2007.