Informe Técnico ICC 2009-04-1

Level-3 BLAS on a GPU: Picking the Low Hanging Fruit

Gregorio Quintana-Ortí, Robert van de Geijn

30 de Abril de 2009

Departamento de Ingeniería y Ciencia de Computadores

Correo electrónico: `gquintan@icc.uji.es, rvdg@cs.utexas.edu`

Universidad Jaime I
Campus de Riu Sec, s/n
12.071 - Castellón
España

Level-3 BLAS on a GPU: Picking the Low Hanging Fruit

Gregorio Quintana-Ortí[1] Robert van de Geijn[2]

Abstract:

The arrival of hardware accelerators has created a new gold rush to be the first to deliver their promise of high performance for numerical applications. Since they are relatively hard to program, with limited language and compiler support, it is generally accepted that one needs to roll up one's sleeves and tough it out, not unlike the early days of distributed memory parallel computing (or any other period after the introduction of a drastically different architecture). In this paper we remind the community that while this is a noble endeavor, there is a lot of low hanging fruit that can be harvested easily. Picking this low hanging fruit benefits the scientific computing community immediately and prototypes the approach that the further optimizations may wish to follow. We demonstrate this by focusing on a widely used set of operations, the level-3 BLAS, targeting the NVIDIA family of GPUs.

Keywords:

Numerical Linear Algebra, Hardware Accelerators, BLAS-3.

[1]Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
`gquintan@icc.uji.es`

[2]Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
`rvdg@cs.utexas.edu`

BLAS-3 sobre una GPU: Recogiendo la fruta fácil

Gregorio Quintana-Ortí[3] Robert van de Geijn[4]

Resumen:

La llegada de los aceleradores *hardware* ha creado una nueva fiebre del oro en ser los primeros en conseguir las prometidas elevadas prestaciones en aplicaciones numéricas. Ya que son relativamente difíciles de programar, con un soporte de lenguajes y compiladores limitado, se acepta que uno tiene que arremangarse la camisa y apretar los dientes, de forma no muy distinta a los primeros días de la programación de máquinas con memoria distribuida (o a cualquier otro periodo tras la introducción de una arquitectura drásticamente diferente). En este trabajo recordamos a la comunidad que mientras ésa es una actitud noble, hay un montón de fruta que puede ser recogida mucho más fácilmente. Recoger esta fruta beneficia a la comunidad científica inmediatamente y sirve para prototipar las aproximaciones que las subsiguientes optimizaciones deberían seguir. En este artículo demostramos lo anterior aplicándolo a un amplio conjunto de operaciones, el BLAS de nivel 3, orientado la la familia de GPUs de NVIDIA.

Palabras clave:

Algebra Lineal Numérica, Aceleradores *Hardware*, BLAS-3.

[3]Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
gquintan@icc.uji.es

[4]Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
rvdg@cs.utexas.edu

Level-3 BLAS on a GPU: Picking the Low Hanging Fruit

Gregorio Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
gquintan@icc.uji.es

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
rvdg@cs.utexas.edu

Technical Report DICC 2009-04-01

FLAME Working Note #37

April 30, 2009

Abstract

The arrival of hardware accelerators has created a new gold rush to be the first to deliver their promise of high performance for numerical applications. Since they are relatively hard to program, with limited language and compiler support, it is generally accepted that one needs to roll up one's sleeves and tough it out, not unlike the early days of distributed memory parallel computing (or any other period after the introduction of a drastically different architecture). In this paper we remind the community that while this is a noble endeavor, there is a lot of low hanging fruit that can be harvested easily. Picking this low hanging fruit benefits the scientific computing community immediately and prototypes the approach that the further optimizations may wish to follow. We demonstrate this by focusing on a widely used set of operations, the level-3 BLAS, targeting the NVIDIA family of GPUs.

Insanity: doing the same thing over and over again and expecting different results.

– Albert Einstein (1879-1955)

1 Introduction

Every time a new architecture arrives, there is a mad dash for high performance. Since compilers, languages, and tools are still rudimentary, this means that some experts roll up their sleeves and achieve high performance the old-fashioned way: they earn it. The problem is that often there are only a few with the right expertise and interest, and therefore this yields only a few routines that are highly optimized. Furthermore, it is acceptable for code that achieves high performance to be messy. When others then come into the picture, they use such implementations as their inspiration, meaning that programmability does not enter the picture until much later in the game. In this paper, we show how insights from the FLAME project, in particular the importance of having a family of algorithms at one's disposal, allow considerable performance gains to be attained with minimal effort. We do so by focusing on the familiar and important matrix-matrix operations that are part of the Basic Linear Algebra Subprograms (BLAS) [4] and targeting the NVIDIA family of GPUs.

The arrival of NVIDIA's GPUs and IBM's Cell Broadband Engine and the recognition that they can be used for computation outside of the field of graphics has created the latest gold rush for performance. In scientific computing this has meant that considerable effort has been expended on implementing the most important kernel: matrix-matrix multiplication (GEMM). Very admirable performance has been achieved [11].

Yet, even operations that are very similar to GEMM, e.g., the other level-3 BLAS, did not achieve decent performance in the CUBLAS library for the NVIDIA GPUs when we started this study. Worse, the effort required to achieve the high performance for GEMM is daunting enough that experts like ourselves have stayed on the sideline, focusing our efforts on using the GEMM implementation for high-level operations like Cholesky factorization by using the accelarators only to compute subproblems that were matrix-matrix multiplications [8, 7, 6]. We all hoped that soon other functionality will be ported to the GPUs, but that some other poor soul would do it for us.

In this paper we once again show that as new functionality and optimizations appear, there are, for those of us who have an aversion to hard work, opportunities to quickly and easily help improve performance in the short run while simultaneously prototyping how performance can eventually be improved by those who are willing to code at a lower level.

This paper is organized as follows: In Section 2 we briefly review the three commonly encountered matrix-matrix multiplication algorithms and use this to remind the reader of the FLAME notation for presenting algorithms. In Section 3 we discuss the corresponding algorithms for various level-3 BLAS operations, where these algorithms have been modified to take advantage of special structure in the matrices. The benefits of picking the right algorithmic variant is illustrated in Section 4. Concluding remarks are found in the final section.

2 Matrix-Matrix Product

At the top level, there are three variants of matrix-matrix product, which we have come to refer to as matrix-panel product (GEMM_MP) based, panel-matrix product (GEMM_PM) based, and (outer) panel-panel product (GEMM_PP) based (also known as rank-k updating). We will discuss these briefly in this section, so that we can refer to them later as we discuss algorithms for the other matrix-matrix operations.

In Figure 1, we illustrate the GEMM_MP based algorithm. At the beginning of the iteration, C_0 and B_0 have already be updated and used, respectively. In the current iteration the next panel of matrix C is updated: $C_1 := C_1 + AB_1$. Then, the advancement for the next iteration shifts C_1 and B_1 to the next blocks of data making blocks C_0 and B_0 larger since they contain more processed data. This visual description of the algorithm motivates the algorithm, in FLAME notation, given in Figure 2. In that figure, we also give the GEMM_PM and GEMM_PP based algorithms. Although all three perform the same number of floating point operations, the final performance that is achieved can be very different depending on the matrix shapes and cache subsystems.

3 Accelerating the CUBLAS

The level-3 BLAS operations are variations of the matrix-matrix product. We will study three: symmetric rank-k update (SYRK), triangular matrix-matrix multiplications (TRMM). and symmetric

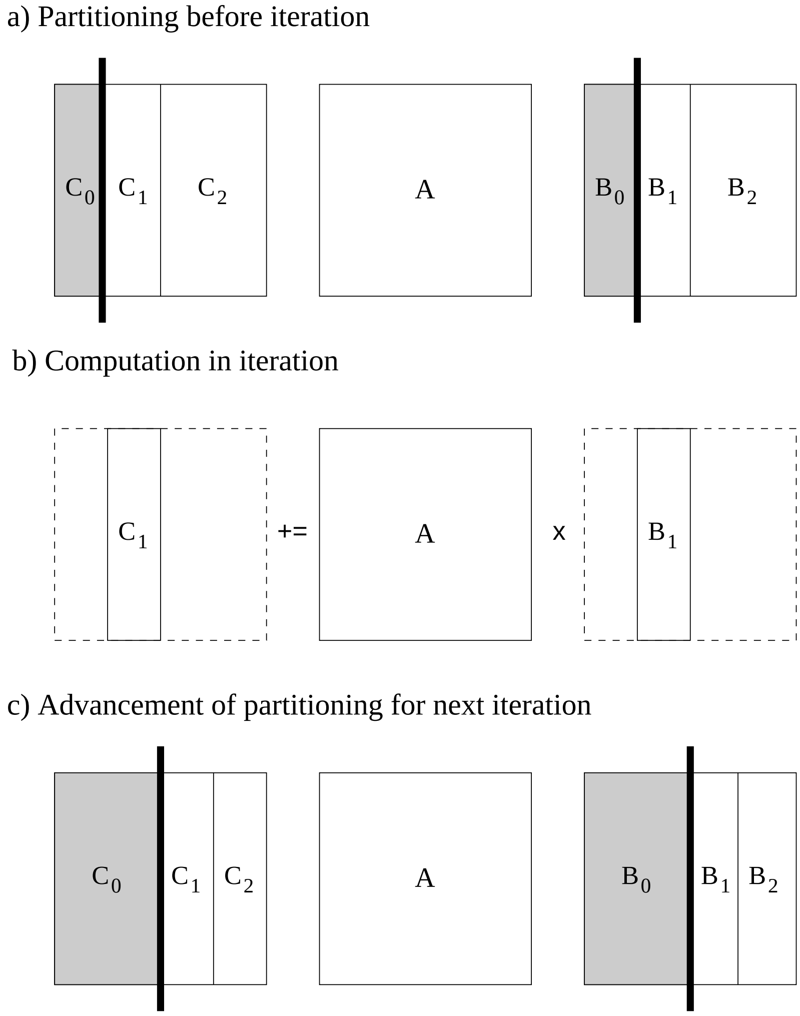

Figure 1: Visual algorithm for matrix-panel variant of matrix-matrix product. Dark background means block is already processed.

matrix-matrix multiplication (SYMM),

It is well-known that for each operation there are algorithms that cast most computation in terms of matrix-matrix multiplication, as was pioneered in [5]. Moreover, as part of the FLAME project we have long advocated that it is important to have multiple algorithmic variants at our disposal so that the best algorithm can be chosen for each situation [3]. The FLAME methodology advocates systematic derivation of these variants [2, 9]. In Section 4 we will show that this is again the case for GPUs. We view our ability to rapidly develop different algorithms as a way of performing *software acceleration*, the natural (and much needed) counterpart to *hardware acceleration*. It yields a cheap (in terms of effort) boost to performance.

The algorithms presented in this section correspond naturally to the matrix-matrix multiplication algorithms given in Section 2, except that they take advantage of the special structure of one of the matrices. Thus, the ..._PP algorithm corresponds to the GEMM_PP algorithm, etc.

Algorithm: $\text{GEMM\_MP}(A, B, C)$

Partition $B \rightarrow \left(B_L \,\|\, B_R \right)$, $C \rightarrow \left(C_L \,\|\, C_R \right)$
  where B_L has 0 columns, C_L has 0 columns

while $n(B_L) < n(B)$ **do**
  Determine block size b
  Repartition
  $\left(B_L \,\|\, B_R \right) \rightarrow \left(B_0 \,\|\, B_1 \,|\, B_2 \right)$,
  $\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, C_1 \,|\, C_2 \right)$
    where B_1 has b columns, C_1 has b columns

  $C_1 := C_1 + AB_1$

  Continue with
  $\left(B_L \,\|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,\|\, B_2 \right)$,
  $\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, C_1 \,\|\, C_2 \right)$

endwhile

Algorithm: $\text{GEMM\_PM}(A, B, C)$

Partition $A \rightarrow \left(\frac{A_T}{A_B} \right)$, $C \rightarrow \left(\frac{C_T}{C_B} \right)$
  where A_T has 0 rows, C_T has 0 rows

while $m(A_T) < m(A)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array} \right), \quad \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ C_2 \end{array} \right)$$

  where A_1 has b rows, C_1 has b rows

  $C_1 := C_1 + A_1 B$

  Continue with

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left(\begin{array}{c} A_0 \\ A_1 \\ \hline A_2 \end{array} \right), \quad \left(\begin{array}{c} C_T \\ \hline C_B \end{array} \right) \leftarrow \left(\begin{array}{c} C_0 \\ C_1 \\ \hline C_2 \end{array} \right)$$

endwhile

Algorithm: $\text{GEMM\_PP}(A, B, C)$

Partition $A \rightarrow \left(A_L \,\|\, A_R \right)$,
$B \rightarrow \left(\frac{B_T}{B_B} \right)$
  where A_L has 0 columns, B_T has 0 rows

while $n(A_L) < n(A)$ **do**
  Determine block size b
  Repartition

$$\left(A_L \,\|\, A_R \right) \rightarrow \left(A_0 \,\|\, A_1 \,|\, A_2 \right), \quad \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ B_2 \end{array} \right)$$

  where A_1 has b columns, B_1 has b rows

  $C := C + A_1 B_1$

  Continue with

$$\left(A_L \,\|\, A_R \right) \leftarrow \left(A_0 \,|\, A_1 \,\|\, A_2 \right), \quad \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \leftarrow \left(\begin{array}{c} B_0 \\ B_1 \\ \hline B_2 \end{array} \right)$$

endwhile

Figure 2: Algorithms for computing matrix-matrix product: Top-left, matrix-panel variant; top-right, panel-matrix variant; bottom-left, panel-panel variant.

Algorithm: $\text{SYRK\_MP\_PM}(C, A)$

Partition
$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$
where C_{TL} is 0×0, A_T has 0 rows
while $m(C_{TL}) < m(C)$ **do**
Determine block size b
Repartition
$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$
where C_{11} is $b \times b$, A_1 has b rows

Syrk_mp	Syrk_pm
$C_{11} := C_{11} - A_1 A_1^T$	$C_{10} := C_{10} - A_1 A_0^T$
$C_{21} := C_{21} - A_2 A_1^T$	$C_{11} := C_{11} - A_1 A_1^T$

Continue with
...
endwhile

Algorithm: $\text{SYRK\_PP}(C, A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_L & A_R \end{array}\right)$
where A_L has 0 columns
while $n(A_L) < n(A)$ **do**
Determine block size b
Repartition
$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right)$
where A_1 has b columns

$C := C - A_1 A_1^T$

Continue with
$\left(\begin{array}{c|c} A_L & A_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array}\right)$
endwhile

Figure 3: Algorithms for computing SYRK

Algorithm: $\text{TRMM\_PP\_PM}(A, B)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$
where A_{TL} is 0×0, B_T has 0 rows
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$
where A_{11} is $b \times b$, B_1 has b rows

Trmm_pp	Trmm_pm
$B_0 := B_0 + A_{01} B_1$	$B_1 := A_{11} B_1$
$B_1 := A_{11} B_1$	$B_1 := B_1 + A_{12} B_2$

Continue with
...
endwhile

Algorithm: $\text{TRMM\_MP}(A, B)$

Partition $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$
where B_L has 0 columns
while $n(B_L) < n(B)$ **do**
Determine block size b
Repartition
$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$
where B_1 has b columns

$B_1 := A B_1$

Continue with
$\left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$
endwhile

Figure 4: Algorithms for computing TRMM.

Algorithm: SYMM_PP_PM(C, A, B)

Partition $C \rightarrow \left(\frac{C_T}{C_B}\right)$, $B \rightarrow \left(\frac{B_T}{B_B}\right)$,
$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where C_T, B_T have 0 rows, A_{TL} is 0×0
while $m(C_T) < m(C)$ **do**
Determine block size b
Repartition
$\left(\frac{C_T}{C_B}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$, $\left(\frac{B_T}{B_B}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$,
$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
where C_1, B_1 have b rows, A_{11} is $b \times b$

Symm_pp	Symm_pm
$C_0 := C_0 + A_{10}^T B_1$	$C_1 := C_1 + A_{10} B_0$
$C_1 := C_1 + A_{11} B_1$	$C_1 := C_1 + A_{11} B_1$
$C_2 := C_2 + A_{21} B_1$	$C_1 := C_1 + A_{21}^T B_2$

Continue with
...
endwhile

Algorithm: SYMM_MP(C, A, B)

Partition $C \rightarrow \left(C_L \,|\, C_R \right)$, $B \rightarrow \left(B_L \,|\, B_R \right)$
where C_L has 0 columns, B_L has 0 columns
while $n(C_L) < n(C)$ **do**
Determine block size b
Repartition
$\left(C_L \,|\, C_R \right) \rightarrow \left(C_0 \,|\, C_1 \,|\, C_2 \right)$,
$\left(B_L \,|\, B_R \right) \rightarrow \left(B_0 \,|\, B_1 \,|\, B_2 \right)$
where C_1 has b columns, B_1 has b columns

$C_1 := C_1 + AB_1$

Continue with
$\left(C_L \,|\, C_R \right) \leftarrow \left(C_0 \,|\, C_1 \,|\, C_2 \right)$,
$\left(B_L \,|\, B_R \right) \leftarrow \left(B_0 \,|\, B_1 \,|\, B_2 \right)$
endwhile

Figure 5: Algorithms for computing SYMM.

SYRK Operation We will focus on a representative case of this operation: $C := C - AA^T$, where C is symmetric and only the lower triangular part of this matrix is stored and computed. In Figure 3 we give three algorithmic variants for this operation.

TRMM Operation For this operation we focus on $C := AB + C$, where A is upper triangular. In Figure 4 we give three algorithmic variants.

SYMM Operation For this operation we focus on $C := AB+C$, where A is symmetric and only the lower triangular part of this matrix is stored. In Figure 5 we give three algorithmic variants.

Other BLAS-3 Operations The same technique can be applied to the other cases of the BLAS-3 operations above presented. Similarly, the same technique can be applied to other BLAS-3 operations (TRSM and SYR2K). We expect achieving similar results since the issues are the same.

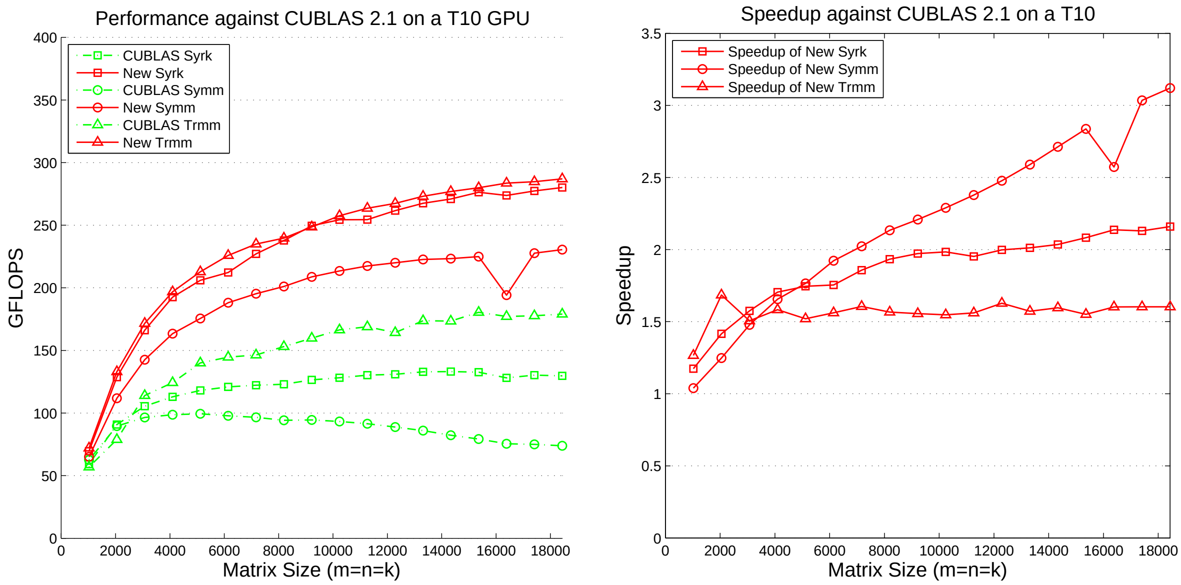

Figure 6: Performance (left) and speedup (right) of the new implementations and equivalent CUBLAS (release 2.1) routines.

4 Experimental Results

The target platform used in the experiments was a NVIDIA T10 GPU (a single GPU of a four GPU NVIDIA Tesla S1070) with 4 GBytes of RAM. The system is connected to a workstation with one Intel Xeon QuadCore E5405 processors (4 cores) at 2.83 GHz with 8 GBytes of DDR2 RAM. CUBLAS Release 2.1 and single precision real floating-point arithmetic were employed in the experiments. Performance is measured in terms of GFLOPS (billions of floating-point operations—flops—per second). The time to transfer data from the host to the memory of the GPU has been included in the performance results.

Figure 6 (right) reports the performance for the three operations discussed in the previous section. Only results for best variants are shown. Figure 6 (left) summarizes the speedups obtained by the new operations against the corresponding routines in CUBLAS version 2.1. The results in both figures show the benefits of our approach. We believe them to be representative of other cases of the presented level-3 BLAS (those where matrices may have been transposed and/or stored in the other triangular part of the array) and the other level-3 BLAS.

The improvement of performances could have been even larger if we had used storage-by-blocks, a well-known modification used in more recent software. We did not employ it to keep full compatibility with NVIDIA CUBLAS.

5 Conclusion

We have demonstrated that with relatively little effort considerable performance gains can be attained when new architectures arrive. The key is to pay attention to the fact that there are many different algorithmic variants for the same operation and to program them in a productive manner. The programs we wrote for this paper required a few hours of time and could have been developed by a relative novice.

Undoubtedly, in response to this paper, there will be a flurry of activity to further improve the performance of the CUBLAS by coding at a much lower level and throwing programmability out the door. In this case, we have indirectly made a contribution to the scientific computing community because faster libraries will then become available sooner. But we are confident that this just means that we will be able to write yet another paper on how to improve the performance of high level routines, with functionality similar to that of LAPACK [1]. And before you know it, a new shift in computer architecture will come along and the mad dash will start all over again. Thus the quote from Einstein.

We are working on an tool, FLAMES2S [10], that can automatically translate algorithms represented in code with the FLAME/C API, used to implement our `libflame` library [12], to low-level code that uses loops and indexing. This tool could easily generate the code that was created manually for the experiments in this paper. With that, we will make further progress towards overcoming the programmability problem for this class of operations and codes.

Acknowledgements

The researcher at the Universidad Jaime I was supported by projects CICYT TIN2008-06570-C04-01, and P1 1B2007-19 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

We thank the other members of the FLAME team for their support.

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du 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] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[3] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 35(1).

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[5] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[6] Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving "large" dense matrix problems on multi-core processors and gpus. In *10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing - PDSEC'09. Roma (Italia)*, 2009. to appear.

[7] Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, and Robert van de Geijn. Using graphics processors to accelerate the solution of out-of-core linear systems. In

8th IEEE International Symposium on Parallel and Distributed Computing, Lisbon (Portugal), 2009. to appear.

[8] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*, 2009.

[9] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com/contents/contents/1911788/`, 2008.

[10] Richard M. Veras, Jonathan S. Monette, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Transforming linear algebra libraries: From abstraction to high performance. *ACM Trans. Math. Soft.* submitted.

[11] Vasily Volkov and James Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[12] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.