Generalized GEMM Kernels on GPGPUs: Experiments and Applications

Davide BARBIERI Valeria CARDELLINI Salvatore FILIPPONE [1]
Università di Roma "Tor Vergata", 00133 Roma, Italy

Abstract. General purpose computing on graphics processing units (GPGPU) is fast becoming a common feature of high performance computing centers. In this paper we discuss some implementation issues related to dense linear algebra computations on GPUs, such as the GEneral Matrix-Matrix product, as well as other kernels sharing the same computational pattern, such as the matrix form of the All-Pairs Shortest-Path problem. Our CUDA implementation has shown a significant improvement on the NVIDIA processing units over the vendor's software. We review the optimization techniques that can be employed to implement such operations, as well as outline further development work in connected application domains.

Keywords. Software and architectures, GPU programming, performance evaluation.

1. Introduction

Efficient matrix-matrix multiplication routines in the BLAS lie at the heart of linear algebra computations as implemented in modern computational libraries such as LAPACK [1,3]. The road to maximal performance is influenced by architectural characteristics of the graphics processing unit (GPU). Indeed, many techniques that were developed over the years for optimal exploitation of vector and RISC processing units have a counterpart in the GPU world, including stride one accesses, bank-conflict avoidance techniques, padding to optimal length, and copy-in/copy-out.

Adapting these techniques to suit GPU devices programming allowed us to obtain excellent results in both memory management and GPU exploitation; in particular,

- gains of orders of magnitude in GPU performance compared to CPU results on matrix operations;
- smooth behavior of the optimized code over many input configurations (up to 59% faster than the vendor's original version).

Indeed, the code developed at our institution is now included in the latest version of the CUBLAS software distribution [8]. Concentrating on GEneral Matrix-Matrix (GEMM) computations is not overly restrictive; it is well known that it is possible to implement efficiently the BLAS standard by reusing the GEMM routine with some additional software [6].

[1]Corresponding author e-mail: salvatore.filippone@uniroma2.it

We also consider the All-Pairs Shortest-Path (APSP) problem on general graphs. An implementation based on a matrix representation of the graph shares most features of the matrix-multiply code, once we make the algebraic substitution of using the sum in place of the product and the maximum in place of the sum. Therefore, we can reuse the implementation techniques adopted for the matrix-matrix multiplication routine, leading to a “brute-force” algorithm that can outperform the “smart” sequential algorithms: algorithms implementing better patterns of problem decomposition scale better than the optimal solution for legacy architectures, despite having worse sequential bounds. This is the case for two APSP algorithms, the original Floyd-Warshall one for general graphs, and a modified *GPU-friendly* version; our results show that there is a significant range of problems where the suboptimal algorithm actually gives the best time to solution on the GPU platform.

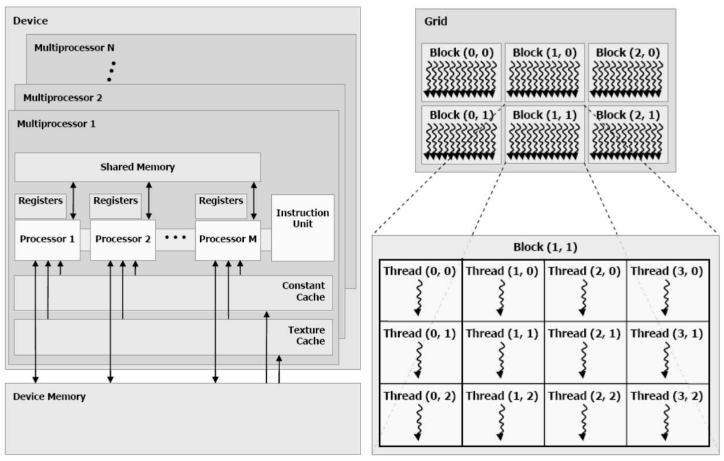
A significant number of research efforts has recently focused on GPGPU, among which [5,10] are more related to our work. In our GEMM kernel, we have applied some of the optimization strategies discussed by Volkov et al. in [10]; with respect to their work, which is included in the CUBLAS library from version 2.0, our kernels for single-precision and double-precision matrix multiplication (SGEMM and DGEMM, respectively) obtain a significant improvement for transposed matrices in input. Harish et al. [5] report results for the implementation of some APSP algorithms on GPUs. Differently from them, we tackle the APSP problem by developing an algorithm entirely based on our efficient matrix multiplication routine.

The rest of this paper is organized as follows. Section 2 gives an overview of the main features of the NVIDIA GPU architecture we used and its programming environment. Section 3 presents the performance optimization techniques for the GEMM kernel and discusses the performance results of the matrix multiplication for both single and double precision computations. Section 4 presents the application of the same techniques to the APSP problem and evaluates their performance. Section 5 discusses our current investigation of other kinds of linear algebra operations. Finally, Section 6 concludes the paper with some final remarks.

2. The Architecture of NVIDIA GPU and its Programming Environment

The NVIDIA GPU architectural model is based on a scalable array of *multi-threaded streaming multi-processors* (SMs), each composed by eight *scalar processors* (SP), one instruction fetch unit, one on-chip shared memory plus additional special-function hardware; a scheme is depicted in Fig. 1(a). Each multiprocessor is capable of creating and executing concurrent threads in a completely autonomous way, with no scheduling overhead, thanks to the hardware support for thread synchronization; threads are created, scheduled, and executed in groups called *warps*.

A warp (a group of 32 threads) is the minimum execution unit, because each compute element has 8 processors sharing a single instruction fetch unit, running at 1/4 of the processor speed; it is also quite common to concentrate on a half-warp when considering accesses to shared memory (both on-chip, and on DRAM), because in this case read and write operations are executed in separate halves. The GPU is capable of hosting a maximum number of warps (and hence, threads) at any given time; the GPU occupancy is defined as the ratio between the actual number of threads and this theoretical maximum



(a) Hardware model (b) Execution hierarchy

Figure 1. NVIDIA GPU architectural model.

To the hardware hierarchy there corresponds a software hierarchy of threads, blocks and grids, as shown in Fig. 1(b). Threads in a given block may synchronize, share data, and be executed on a given multiprocessor with essentially zero overhead. The dimension of blocks and grids is specified by the programmer to match the problem size with the available execution hardware. The CUDA programming environment specifies a set of facilities to create, identify, and synchronize the various threads involved in the computation.

The device memory is divided into global, local, and on-chip shared memory areas. The efficient exploitation of the available global memory bandwidth depends critically on two basic code features: data structure alignment and coalesced accesses.

The alignment issue is quite clear, because of the additional overhead of unaligned accesses; its implementation is aided by the use of compilation directives. Coalesced accesses to global memory involve all threads in a half-warp, and enable completion of read accesses in a single memory transaction. Optimal access patterns for coalescing may vary among different GPU models, but so far they have done so in a backward compatible manner.

Local memory is private to each thread; it is not directly available to the programmer, and is only used by the compiler to handle register spill.

Shared memory is actually local to each multiprocessor; it is shared among threads and it is interfaced to the cores with a crossbar of 16 elements, so it is organized in 16 banks. Given this structure, threads in a half-warp can access the memory with no overhead provided that there are no back conflicts.

3. Performance Optimization Techniques for the GEMM Kernels

The GEMM multiply that computes $C = \alpha A \times B + \beta C$ is one of the best known computational kernels in common scientific applications, and lies at the heart of the BLAS and LAPACK software packages. It is usually among the first kernels to be implemented on a new architecture, both because of its intrinsic usefulness, and because its study can reveal tricks and techniques useful in other contexts.

We started from a naive implementation of the matrix multiply, and following the programming guide [7, p. 71] guidelines, we iteratively improved our code, comparing our timing results with those obtained with the CUBLAS 2.0 SGEMM.

The basic optimization principles we followed can be easily stated:

- minimize low-throughput instruction use;
- maximize memory bandwidth usage, for each memory subsystem;
- overlap arithmetic operations with memory transactions.

If we analyze the examples provided in the CUDA programming guide, we see that even though the theoretical maximum for the number of threads on a given (multi)processor is constant, the actual limitations come from the amount of shared memory and from the number of registers employed per thread block. Specifically, the number of registers is very difficult to control, being determined by the compiler optimizations. However, it is possible to force the maximum number of registers of a kernel, passing some special parameters to the compiler. Even in this case, it is still necessary to check the compiler output for register occupancy, employing some tools provided in the CUDA environment or from third parties, such as the binary code disassembler *decuda* [9].
Subsequent tests showed that ensuring maximum GPU occupancy is not the most important factor in the GEMM kernel. Indeed, we adopted the large tiling technique presented in [10], consisting of:

- computing a tile of matrix C of size 64×16 per thread block, where each thread computes a column within the tile (see Fig. 2);
- using shared memory just for tiles from matrix B, whereas tiles from matrix A are loaded from global memory.

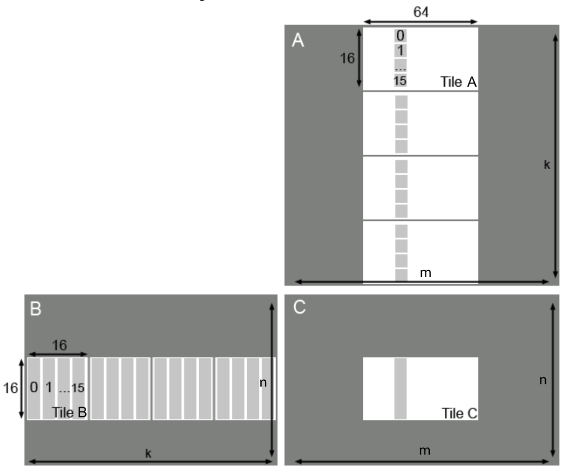


Figure 2. Optimized matrix product tiling
(A, B, and C matrices are stored in Fortran memory layout).

This implied a drop in occupancy down to 33%; nevertheless, it shows better performance due to better memory transactions hiding, more efficient calculation of the element offsets, and less use of shared memory (i.e., a lower number of total thread block synchronizations).

The offset computation reduction is achieved by properly sequencing the accesses to tile B by columns (rather than by rows), and by doing explicitly the pointer arithmetic needed to update the base addresses of the individual columns. However, care must be taken to avoid bank conflicts; in our case, it is sufficient to adopt the time-honored trick of increasing by one the leading dimension of the buffer in shared memory that will hold the (transposed) tile.

Further performance improvements were gained through the *prefetching* of elements from tile A, within the multiplication loop; this was enough to reach CUBLAS 2.0 SGEMM performance level.

Variations of the above scheme apply for the other combinations of transpose/no-transpose of the input matrices A and B that have to be supported to conform to the BLAS standard. In particular, we succeeded in exploiting the same optimizations in a version of the SGEMM routine for transposed matrices in input; our version computes the AB product by first calculating BA and then transposing the result on-the-fly without using additional shared memory. The performance results in Fig. 3 show an improvement of up to 59% with respect to the 2.0 version of the CUBLAS library on the NVIDIA GeForce 9800GTX+ in the innermost kernel, and up to 50% when considering host-device-host memory transactions; overall this achieves approximately 40% of the available peak performance. Note that NVIDIA has included some of our code in its 2.1 version of the CUBLAS library, and therefore the performance gap has now disappeared.

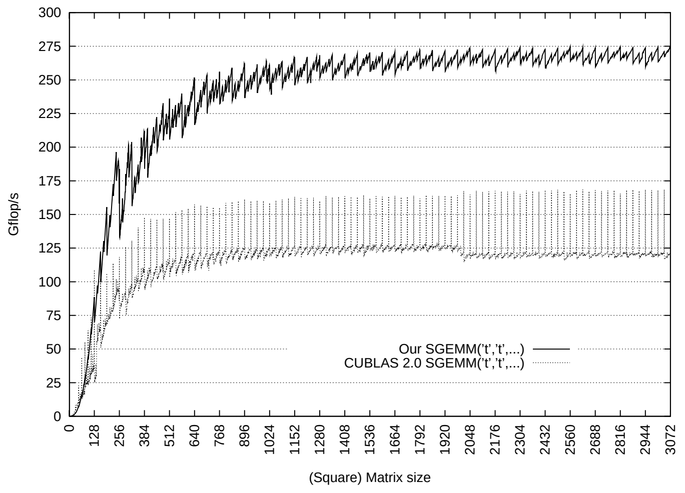


Figure 3. Performance of transpose-transpose SGEMM on NVIDIA GeForce 9800GTX+ (kernel only).

The last technique we employed to achieve maximum performance stems from the need to handle matrix sizes that are not necessarily multiples of tile size, causing the need of additional control and the misalignment of matrix rows (and so uncoalesced accesses); The obvious solution is to pad with zeroes the various involved matrices during the copy-in/copy-out between host and device memory; strictly speaking, this technique is external to the GEMM kernel, but affects the user anyway.

Newer NVIDIA GPUs offer native support for double precision computations. As shown in Fig. 4, for DGEMM we obtained nearly 100 % of peak performance (versus 42% of SGEMM), proving that the optimization techniques are essentially the same as in the single precision version, modulo the obvious adjustments to the memory allocations and address computations. The main difference is that DGEMM does not need to use the prefetching technique, because the new hardware seems to improve the double precision pipeline throughput when operands are in shared memory (i.e., it does not need to move all operands first from shared memory to registers).

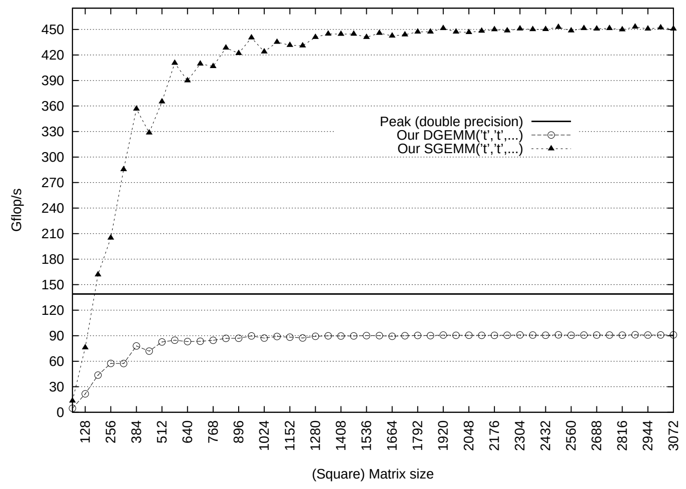


Figure 4. Performance of transpose-transpose DGEMM on NVIDIA GeForce GTX285 (kernel only).

4. Performance Results for the APSP Problem

An interesting application of the same performance optimization techniques we have analyze so far is in the context of the All-Pairs Shortest-Path (APSP) problem. Given a weighted graph $G = (V, E, W)$, with non negative weights, the aim is to find out the distance matrix between all possible pairs of vertices. In the serial context, the Floyd-Warshall algorithm is a well-known solution characterized by time-complexity $O(V^3)$ and space complexity $O(V^2)$. If the graph is quite sparse, the algorithm turns out to be much more expensive than, e.g., the Dijkstra algorithm. Nevertheless, Floyd-Warshall has a redeeming feature: its formal structure is that of a generalized matrix “multiplication” algorithm, as shown in Fig. 5, where Ms is the matrix of distance estimates and Ma is the adjacency matrix. Thus, the Floyd-Warshall algorithm is amenable to the same optimization techniques we have discussed above.

```
$Ms \leftarrow Ma$
**for** $k = 1, \ldots$ **do**
    **for** $D(i,j) \in Ms$ **do**
        $D(i,j) \leftarrow \min(D(i,j), D(i,k) + D(k,j))$
    **end for**
**end for**
```

Figure 5. Floyd-Warshall algorithm for APSP.

The only significant difference between the Floyd-Warshall algorithm and the Matrix-Multiply based one is that Floyd-Warshall is sequentially bounded by the outer loop, that prevents the algorithm to be execute entirely on a kernel launch, due to the grid synchronization. For this reason, it is harder to gain the same GEMM code efficiency. Therefore, we have developed an algorithm entirely based on the matrix multiply code, that determines, after N iterations, the 2^N-shortest paths for all pairs of nodes. Every

Table 1. APSP kernel timing on random graphs for NVIDIA GeForce GTX285 (time values are in msec).

	Floyd-Warshall			*Matrix-Multiply based*		
nodes	#edges/#nodes ratio			#edges/#nodes ratio		
	1/10	1/1	10/1	1/10	1/1	10/1
64	1.28	1.3	1.4	**0.49**	**0.72**	**0.74**
128	2.49	2.53	2.81	**0.75**	**1.13**	**1.27**
256	5.5	5.6	8	**1.45**	**2.12**	**3**
512	14	14	40	**5.73**	**8.45**	**15**
1024	41	**42.6**	275	**30**	43	**102**
2048	**136**	**157**	1525	191	262	**816**
4096	**508**	**659**	8514	1480	1896	**6877**
8192	**2067**	**3012**	**51949**	13901	19661	58736

iteration has the same time complexity of the Floyd-Warshall algorithm, so it totally has $O(V^3 \cdot \log(\min(E, V-1)))$ time complexity, where $\min(E, V-1)$ is the maximum length of all shortest paths in a graph. Table 1 compares the performance results of the Floyd-Warshall algorithm vs. the matrix-multiply based one, which represents the "GPU-friendly but theoretical worse" approach. The graphs with a number of nodes varying from 64 up to 8192 (and three different values for the ratio between the number of edges and the number of nodes) have been generated using the GTgraph-random tool [2]. In Table 1 the values reported in bold represent the better performing solution for each graph size. The results show that for graphs of large sizes and not too dense the Floyd-Warshall algorithm outperforms the matrix-multiply based one, while the latter turns out to be the better solution when the graph density increases.

5. New Developments

The usage of the GPU programming model is spreading throughout the scientific computing world because of the very appealing price/performance ratio. However, the successful applicability of these computing devices is far from universal; in particular, applying the GPU programming techniques to sparse linear algebra computations is not an obvious proposition.

The bane of sparse matrix computations is the need to perform indexed addressing; this is well known in the scientific computing community, and is also recognized by (at least some in) the hardware design community, as recently highlighted in [4]. Unfortunately, the GPU architecture does not offer specialized support for indexed addressing, and thus the performance level that can be achieved falls quite short of the theoretical peak supported by the hardware. In preliminary results, we have found that the performance ratio between single precision and double precision kernels is just near to 2x (although peak ratio is more than 10x). This proves as our first sparse matrix-vector multiply kernel implementation still holds a significant memory bottleneck, due to the hardness to provide coalesced accesses to such algorithms, and so to the impossibility of hiding memory transactions with ALU operations.

6. Conclusions

In this paper we have shared our programming experience on the NVIDIA CUDA environment and devices, showing as it is possible to take advantage of this cheap architecture for high performance computing. Besides an evident benefit from using GPU for embarrassingly parallel algorithms, there are some perplexities on using CUDA for algorithms that have several sequential constraints or, more in general, for algorithms that do not provide a large ratio of arithmetic operations per memory accesses, like sparse numerical computing. Despite the GPU inefficiency to suit these algorithms compared to the peak performance, it is still convenient compared to other general purpose solutions. Nevertheless, GPU manufacturers promise to increment the already large on-board GDDR bandwidth and to improve the double precision support of future devices and this could imply interesting implications in scientific computing.

As discussed in Section 5, our current research direction is to investigate other kinds of linear algebra operations, such as the sparse matrix-vector product involved in many types of computations, specifically in explicit time-marching schemes for fluid dynamics based on the Lattice-Boltzmann model. Our future work will also focus on the best exploitation of further evolutions of the newer GPUs generations.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[2] D. A. Bader and K. Madduri. *GTgraph: A suite of synthetic graph generators*. `http://hpcrd.lbl.gov/~kamesh/GTgraph/`

[3] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, **16**(1):1–17, 1990.

[4] J. Gebis and D. Patterson. Embracing and extending 20-th century instrucion set architectures. *IEEE Computer*, **40**(4):68–75, 2007.

[5] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *Proc. of HiPC 2007*, LNCS Vol. 4873, Springer, 2007.

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

[7] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Edition 2.0,* 2008. `http://www.nvidia.com`

[8] NVIDIA Corporation. *CUBLAS Library, Programming Guide, version 2.1*, Sept. 2008.

[9] W. J. van der Laan. Cubin utilities. `http://www.cs.rug.nl/~wladimir/decuda/`

[10] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. *Proc. of 2008 ACM/IEEE Conf. on Supercomputing*, Austin, TX, Nov. 2008.