An Implementation of the Tile QR Factorization for a GPU and Multiple CPUs

– LAPACK Working Note 229

Jakub Kurzak
Rajib Nath
Peng Du
Department of Electrical Engineering and Computer Science, University of Tennessee

Jack Dongarra
Department of Electrical Engineering and Computer Science, University of Tennessee
Computer Science and Mathematics Division, Oak Ridge National Laboratory
School of Mathematics & School of Computer Science, University of Manchester

ABSTRACT

The tile QR factorization provides an efficient and scalable way for factoring a dense matrix in parallel on multicore processors. This article presents a way of efficiently implementing the algorithm on a system with a powerful GPU and many multicore CPUs.

Contents

1 Background 1

2 Motivation 1

3 Implementation 2
- 3.1 CPU Kernels . . . 3
- 3.2 GPU Kernel . . . 3
- 3.3 Scheduling . . . 5
- 3.4 Communication . . . 5

4 Results and Discussion 6

5 Conclusions 7

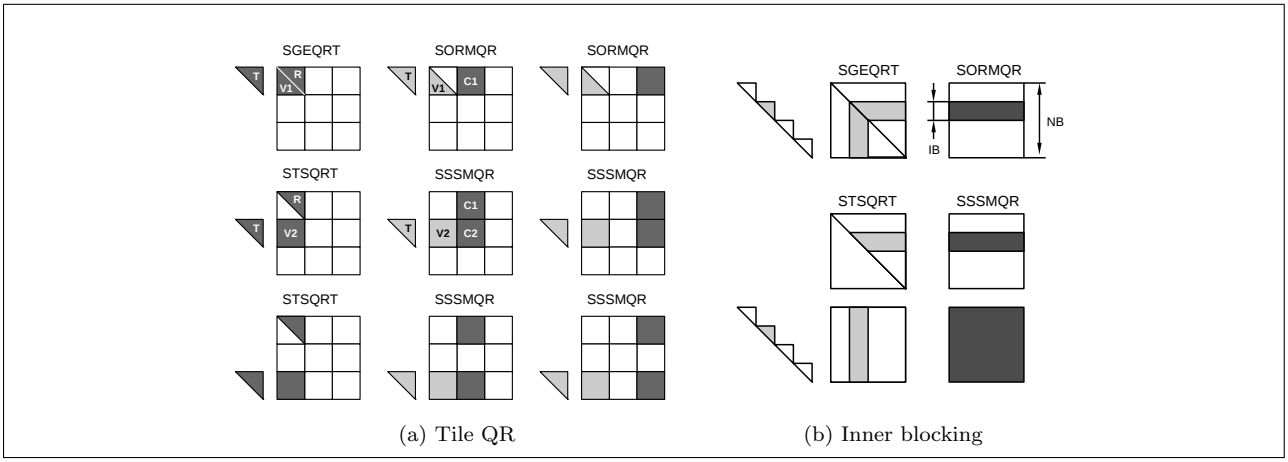
6 Future Work 7

1 Background

In recent years a tiled approach in applying Householder transformations has proven to be a superior method for computing the QR factorization of a dense matrix on multicore processors, including "standard" (x86 and alike) processors [1–3] and also the Cell Broadband Engine [4]. The basic elements contributing to the success of the algorithm are: processing the matrix by tiles of relatively small size, relying on laying out the matrix in memory by tiles, and scheduling operations in parallel in a dynamic, data-driven fashion.

2 Motivation

The efforts of implementing dense linear algebra on multicore and accelerators have been pursued in two different directions, one that emphasizes the efficient use of multicore processors [1–3], exemplified by the PLASMA project [5], and another that emphasizes the use of accelerators [6, 7], exemplified by the MAGMA project [8]. While the former makes great usage of multicores, it is void of support for accelerators. While the the latter makes great usage of GPUs, it seriously underutilizes CPU resources.



(a) Tile QR (b) Inner blocking

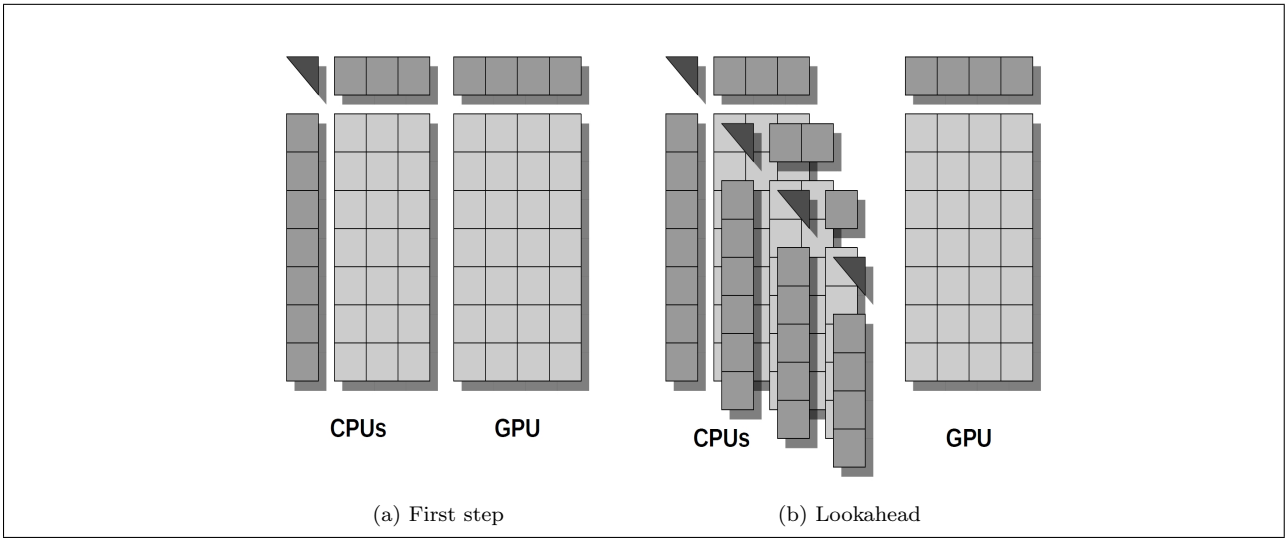
Figure 1: Tile QR with inner blocking.

The main problem of existing approaches to accelerating dense linear algebra using GPUs is that GPUs are used like monolithic devices, i.e., like another "core" in the system. The massive disproportion of computing power between the GPU and the standard cores creates problems in work scheduling and load balancing. As an alternative, the GPU can be treated as a set of cores, each of which can efficiently handle work at the same granularity as a standard CPU core.

3 Implementation

All aspects of the tile QR factorization have been documented very well in recent literature. Only a minimal description is presented here for the sake of further discussion. Figure 1a shows the basics of the algorithm and introduces the four sequential kernels relied upon. One potential deficiency of the algorithm is the introduction of extra floating point operations not accounted for in the standard $4/3N^3$ formula. These operations come from accumulation of the Householder reflectors as reflected in the triangular T matrices in Figure 1a and amount to 25 % overhead if the T matrices are full triangles. The problem is remedied by internal blocking of the tile operations as shown in Figure 1b, which produces T matrices of triangular block-diagonal form and makes the overhead negligible.

The basic concept of the implementation presented here is laid out in Figure 2. It relies upon running the three complex kernels (SGEQRT, STSQRT, SORMQR) on CPUs and only offloading the performance critical SSSMQR kernel to the GPU. It is done in such a way that the Streaming Multiprocessor (SM) of the GPU is responsible for a similar amount of work as one CPU core. In one step of the factorization, the CPUs factorize one panel of the matrix (the SGEQRT and STSQRT kernels), update the top row of the *trailing submatrix* and also update a number of initial columns of the trailing submatrix (through a CPU implementation of the SSSMQR kernel). The GPU updates the trailing submatrix through a GPU implementation of the SSSMQR kernel (Figure 2a). As soon as some number of initial columns is updated, the CPUs can also initialize follow-up panel factorizations and updates, a concept known as *a lookahead* (Figure 2b). This way, when the GPU is finished with one update, the next panel is immediately ready for the following update, which keeps the GPU occupied all the time (avoiding GPU idle time). Also, at each step of the factorization, the GPU part shrinks by one column, and when the size of the trailing submatrix reaches the width of the lookahead, the work is continued by the CPUs only.



(a) First step (b) Lookahead

Figure 2: Splitting the work between the CPUs and the GPU.

3.1 CPU Kernels

CPU implementations of all four kernels are taken directly from the publicly available *core BLAS* component of the PLASMA library. Ideally, core BLAS would be implemented as monolithic kernels optimized to the maximum for a given architecture. However, this amounts to a prohibitive coding effort, mainly due the challenges of SIMD'zation for vector extensions ubiquitous in modern processors. Instead, these kernels are currently constructed from calls to BLAS and LAPACK, which is a suboptimal way of implementing them, but the only feasible one known to the authors. They are known to typically deliver about 75 % of the core's peak, while straight matrix multiplication delivers up to 95 %.

3.2 GPU Kernel

The main building block of the SSSMQR kernel is matrix multiplication. The process of coding fast matrix multiplication for a GPU relies on a classic autotuning approach similar to the one utilized in the ATLAS library [9, 10], where a code generator creates multiple variants of code and the best one is chosen through benchmarking. This is the approach taken by the MAGMA library and here the authors leverage this work by using MAGMA SGEMM (matrix multiply) kernels as building blocks for the SSSMQR kernel [11, 12]. One shortcoming of this (initial) work is that the kernels were developed for the Nvidia G80 (Tesla) architecture and are used for the Nvidia GF100 (Fermi) architecture.

The two required operations are $C = C - A^T \times B$ and $C = C - A \times B$. Figure 3 shows MAGMA implementations of these kernels. The first one is implemented as a 32×32 by $32 \times k$ matrix multiplication using a thread block of size 8×8 (Figure 3a). The second one is implemented as a 64×16 by $16 \times k$ matrix multiplication using a thread block of size 64×1 (Figure 3b).

Figure 4 shows the process of constructing the SSSMQR kernel. MAGMA SGEMM kernels allow for building an SSSMQR kernel for tile sizes $NB = 32, 64, 96, ...$ with inner blocking $IB = 32, 64, 96, ...$, such that IB divides NB (see the first paragraph of section 3 and Figure 1 for the explanation of inner blocking). It has been empirically tested that the combination (IB, NB) of (32, 256) provides the best performance on the GPU and is also a good combination for the CPUs.

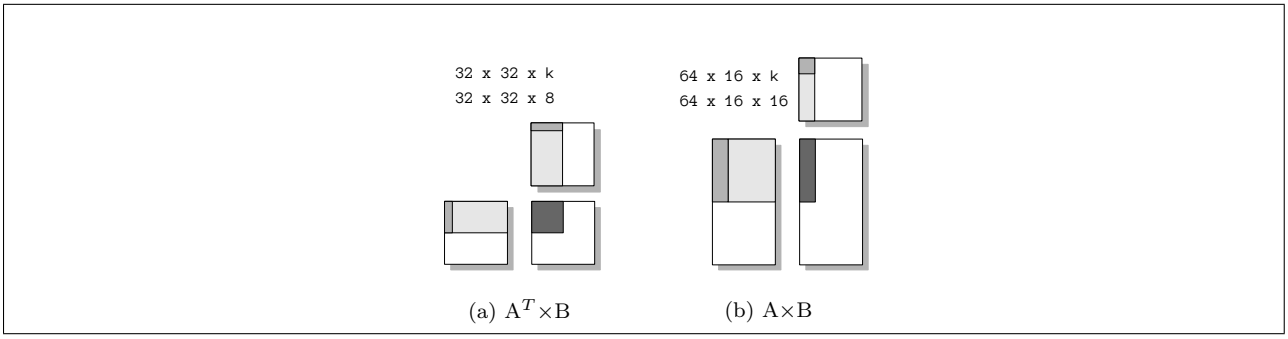


Figure 3: GPU SGEMM kernels.

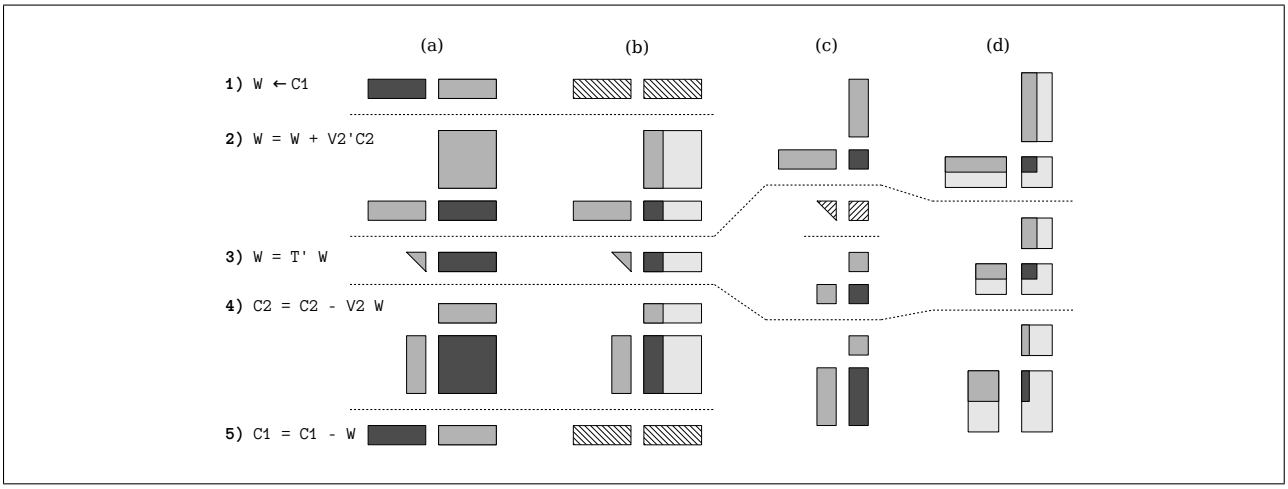


Figure 4: Construction process of the SSSMQR GPU kernel.

The construction of the kernel can be explained in the following steps. Figure 4a shows the starting point. (This would be a CPU implementation of the kernel.) Operation 1 is a memory copy, which is trivial to implement in CUDA, and will not be further discussed. Same applies to operation 5, which is an AXPY operation, also trivial to implement. The first step is a vertical split of all operations (Figure 4b) to provide more parallelism. (What is being developed here is an operation for one thread block, and multiple thread blocks will run on a single Streaming Multiprocessor.) The next step is a conversion of the in-place triangular matrix multiplication (operation 3) to an out-of-place square matrix multiplication (Figure 4c). The last step is using MAGMA SGEMM kernels to implement operations 2, 3 and 4. The last step is done by incorporating the SGEMM kernels into the body of the SSSMQR kernel and a number of manual code adjustments such as reshaping pointer arithmetics and reshaping the thread block, a somewhat tedious process. A quicker alternative would be to rely on automatic function inlining. It turns out, however, that doing so results in a higher register usage, which leads to lower occupancy and lower overall performance. At the same time, forcing register usage with a compiler flag causes register spills to the memory and, again, lower performance.

```
for (k = 0; k < SIZE; k++)
{
  QUARK_Insert_Task(CORE_sgeqrt, ...

  for (m = k+1; m < SIZE; m++)
    QUARK_Insert_Task(CORE_stsqrt, ...

  for (n = k+1; n < SIZE; n++)
    QUARK_Insert_Task(CORE_sormqr, ...

  for (m = k+1; m < SIZE; m++)
    for (n = k+1; n < SIZE; n++)
    {
      QUARK_Insert_Task(CORE_sssmqr, ...
    }
}
```

(a) CPUs only

```
for (k = 0; k < SIZE; k++)
{
    QUARK_Insert_Task(CORE_sgeqrt, ...

    for (m = k+1; m < SIZE; m++)
        QUARK_Insert_Task(CORE_stsqrt, ...

    for (n = k+1; n < SIZE; n++)
        QUARK_Insert_Task(CORE_sormqr, ...

    task = QUARK_Task_Init(cuda_sssmqr, ...
    for (m = k+1; m < SIZE; m++)
        for (n = k+1+lookahead; n < SIZE; n++)
        {
            QUARK_Task_Pack_Arg(task, &C1, INOUT);
            QUARK_Task_Pack_Arg(task, &C2, INOUT);
            QUARK_Task_Pack_Arg(task, &V2, INPUT);
            QUARK_Task_Pack_Arg(task, &T,  INPUT);
        }
    QUARK_Insert_Task_Packed(task);
}
```

(b) CPUs + a GPU

Figure 5: Simplified QUARK code.

Since tiles in a column have to be updated in a sequence, each thread block updates a stripe of the trailing submatrix of width $IB = 32$. This creates enough parallelism to keep the GPU busy for matrices of size 4000 and higher.

3.3 Scheduling

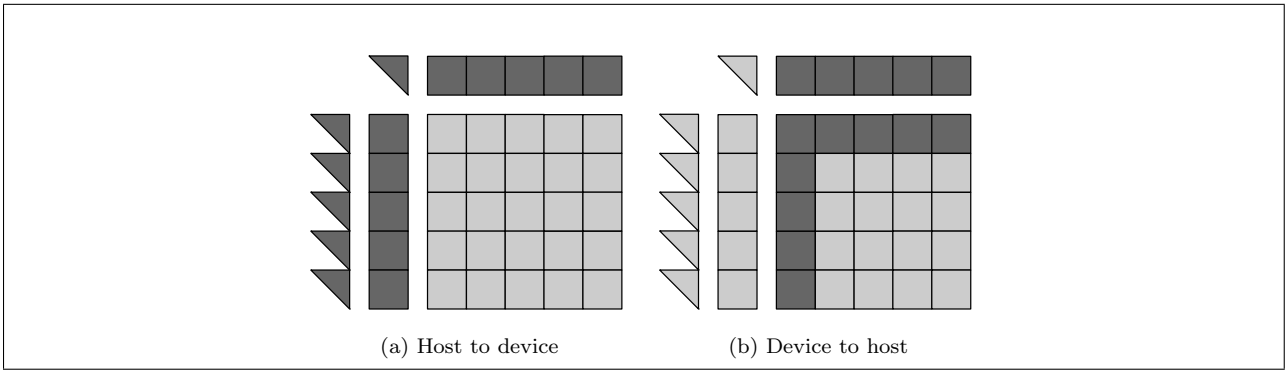
The next critical element of the implementation is dynamic scheduling of operations. Given the lookahead scheme presented in Figure 2b, keeping track of data dependencies and scheduling of operations manually would be close to impossible. Instead, the QUARK scheduler was used, the one used internally by the PLASMA library.

QUARK is a simple dynamic scheduler, very similar in design principles to projects like, e.g., Jade [13, 14], StarSs [15] or StarPU [16, 17]. The basic idea is the one of unrolling sequential code at runtime and scheduling tasks by resolving three basic data hazards: *Read After Write (RAW)*, *Write After Read (WAR)* and *Write After Write (WAW)*.

The crucial concept here is the one of task aggregation. The GPU kernel is an aggregate of many CPU kernels, i.e., one invocation of the GPU kernel replaces many invocations of CPU kernels. In order to use the dynamic scheduler, the GPU kernel inherits all data dependencies of the CPU kernel it aggregates. This is done by a simple extension to the dynamic scheduler, where a task is initialized without any dependencies and dependencies are added to it in a loop nest. Figure 5a shows QUARK code for multicores only and Figure 5b shows QUARK code for multicores and a GPU (with lookahead).

3.4 Communication

If the CPUs and the GPU were sharing a common memory system, the solution would be complete at this point. Since this is not yet the case, data has to be transferred between the CPUs memory (the *host* memory) and the GPU memory (the *device* memory) through the slow PCI bus. Despite the disparity between the computing power of a GPU and the communication power of the PCI, a GPU can be used efficiently for dense linear algebra thanks to the *surface-to-volume effect* ($O(N^3)$ volume of computation and $O(N^2)$ volume of communication).



(a) Host to device (b) Device to host

Figure 6: *Wavefront* communication CPU-GPU communication.

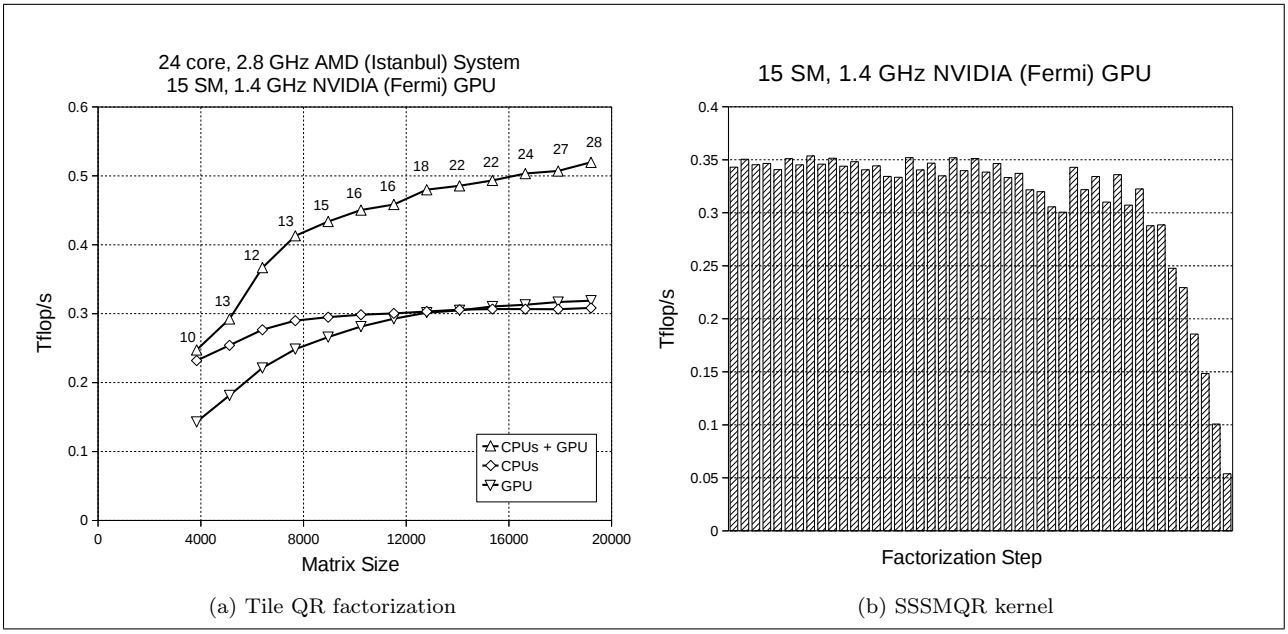
Here an approach is taken similar to the one of the MAGMA library. It can be referred to as *wavefront communication*, since at each step only a moving boundary region of the matrix is communicated. Initially, a copy of the entire matrix is made in the device memory. Then communication follows the scheme shown in Figure 6. Each GPU kernel invocation is preceded by bringing in to the device memory the panel, the column of T factors, and the top row associated with a given update (Figure 6a). Then, each GPU kernel execution is followed with sending back to the host memory the row brought in before the kernel execution, the first row and the first column of the update (Figure 6b). No additional communication is required when the factorization is completed. At that point the host memory contains the factorized matrix.

4 Results and Discussion

Performance experiments were run using 4 sockets with 6-core AMD Opteron^{TM}
8439 SE (Istanbul) processors clocked at 2.8 GHz and an Nvidia GTX 480 (Fermi architecture) graphics card clocked at 1.4 GHz. The core BLAS kernels relied on Intel MKL 11.1 for performance, which turned out to be faster than the AMD ACML library. GCC version 4.1.2 was used for compilation of the CPU code and CUDA SDK 3.1 for compilation of the the GPU code and GPU runtime. The system was running Linux kernel 2.6.32.3 x86_64.

Figure 7a shows the performance results for CPUs-only runs, GPU-only runs and runs using both the 24 CPU cores and the GPU. GPU-only runs are basically CPU+GPU runs with *lookahead* = 1. This way the GPU is occupied most of the time, but the CPUs only perform the minimal part of the update to be able to factorize one consecutive panel, while the GPU performs the update so that the GPU does not stall waiting for the panel to be factorized. The CPU+GPU runs are runs with a deep level of lookahead, which keeps the CPUs occupied while the GPU performs the update. The optimal level of the lookahead was tuned manually and is reflected by the number on top of each performance point. Figure 7b shows the performance of each invocation of the GPU SSSMQR kernel throughout the largest factorization of a 19200×19200 (75×75 tiles) matrix with lookahead of 28 (since the number of stripes of $75 - 28 - 1 = 46$).

Interestingly, for this setup, the CPU-only and GPU-only runs deliver very similar performance (slightly above 300 Gflop/s). One can clearly see the performance advantage of using both the CPUs and the GPU, delivering together the performance of 520 Gflop/s. Once again, the authors admit to using suboptimal GPU kernels for the Fermi architecture. (The development of optimal Fermi kernels is underway.)



(a) Tile QR factorization

(b) SSSMQR kernel

Figure 7: Performance results.

5 Conclusions

It has been clearly shown that a system equipped with a high number of conventional cores and a GPU accelerator can be efficiently utilized for a classic dense linear algebra workload. The necessary components are a dynamic scheduler capable of task aggregation (accepting tasks with a very high number of dependencies) and a custom GPU kernel (not readily available in the CUBLAS library). Although a custom kernel is required, it can be built from blocks already available in a CUDA BLAS implementation, such as the one provided by MAGMA.

6 Future Work

The immediate objectives of the authors are to develop an optimized Fermi kernel for the SSSMQR operation (which should at least double the GPU performance) and generalize the work to multiple GPUs. One can observe that the latter can be accomplished by splitting the trailing submatrix vertically among multiple GPUs. In this case the wavefront communication will involve communication between each GPU and the CPUs and also communication between each pair of GPUs due to the shrinking size of the trailing submatrix and the necessity to shift the boundaries between the GPUs to balance the load.

References

[1] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

[2] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.

[3] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computat.: Pract. Exper.*, 21(1):15–44, 2009. DOI: 10.1002/cpe.1467.

[4] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*, 00:1–12, 2008. DOI: 10.3233/SPR-2008-0268.

[5] PLASMA. `http://icl.cs.utk.edu/plasma/`.

[6] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. DOI: 10.1016/j.parco.2009.12.005.

[7] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of the 2010 IEEE International Parallel & Distributed Processing Symposium, IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[8] MAGMA. `http://icl.cs.utk.edu/magma/`.

[9] R. C Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parellel Comput. Syst. Appl.*, 27(1-2):3–35, 2001. DOI: 10.1016/S0167-8191(00)00087-9.

[10] ATLAS. `http://math-atlas.sourceforge.net/`.

[11] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Roube, LA, May 25-27 2009. Springer.

[12] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2010. Springer.

[13] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Programming Lang. Syst.*, 20(3):483–545, 1998. DOI: 10.1145/291889.291893.

[14] The Jade Parallel Programming Language. `http://suif.stanford.edu/jade.html`.

[15] J. Planas, R. M. Badia, E. Ayguad, and J. Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perf. Comput. Applic.*, 23(3):284–299, 2009. DOI: 10.1177/1094342009106195 .

[16] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computat. Pract. Exper.*, 2010. (to appear).

[17] StarPU. `http://runtime.bordeaux.inria.fr/StarPU/`.