Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems

Tomov, Stanimire and Dongarra, Jack and Baboulin, Marc

2009

MIMS EPrint: **2009.7**

ISSN 1749-9097

Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems

Stanimire Tomov[1], Jack Dongarra[1,2,3], and Marc Baboulin[1,4]

[1] University of Tennessee (USA)
[2] Oak Ridge National Laboratory (USA)
[3] University of Manchester (UK)
[4] University of Coimbra (Portugal)

October 14, 2008

Abstract. If multicore is a disruptive technology, try to imagine hybrid multicore systems enhanced with accelerators! This is happening today as accelerators, in particular Graphics Processing Units (GPUs), are steadily making their way into the high performance computing (HPC) world. We highlight the trends leading to the idea of hybrid manycore/GPU systems, and we present a set of techniques that can be used to efficiently program them. The presentation is in the context of Dense Linear Algebra (DLA), a major building block for many scientific computing applications. We motivate the need for new algorithms that would split the computation in a way that would fully exploit the power that each of the hybrid components offers. As the area of hybrid multicore/GPU computing is still in its infancy, we also argue for its importance in view of what future architectures may look like. We therefore envision the need for a DLA library similar to LAPACK but for hybrid manycore/GPU systems. We illustrate the main ideas with an LU-factorization algorithm where particular techniques are used to reduce the amount of pivoting, resulting in an algorithm achieving up to 388 GFlop/s for single and up to 99.4 GFlop/s for double precision factorization on a hybrid Intel Xeon (2x4 cores @ 2.33 GHz) – NVIDIA GeForce GTX 280 [5] (240 cores @ 1.30 GHz) system.
Keywords: hybrid computing, dense linear algebra, parallel algorithms, LU factorization, multicore processors, graphics processing units.

1 Introduction

Computing technology is currently undergoing a transition driven by power and performance limitations that provide more and more on-die x86 cores each year. The current standard is quad core chips and the development roadmap indicates that 8, 16, 32 core chips will follow in the coming years. There is now widespread recognition that performance improvement on CPU-based systems in the near future will come from the use of multicore platforms.

But multicore architectures are not the only proposed way forward. IBM, for example, introduced its own heterogeneous multicore architecture, the CELL Broadband Engine. Other innovative solutions proposed include GPUs, FPGAs, and ASICs. GPUs stand out in a unique way from all these innovative solutions because they are produced as commodity processors and their floating point performance has significantly outpaced that of CPUs in recent years (see Figure 1). Moreover GPUs have become easier to program, which allows developers to effectively exploit their computational power, as is evident for example in NVIDIA's Compute Unified Device Architecture (CUDA) [29]. Joining the hybrid architectures trend, Intel recently announced its plans for a graphics accelerated chip, Larrabee, a hybrid between multicore CPU and a GPU [24].

The problems and the challenges for developers in the new computational landscape are daunting. Many familiar and widely used algorithms and libraries will become obsolete and would have to be rethought and rewritten in order to take advantage of the new architectures. In many cases, the optimal solution may well be a hybrid solution combining the strengths of each platform. The

[5] Note that this is just the first NVIDIA card that supports double precision arithmetic

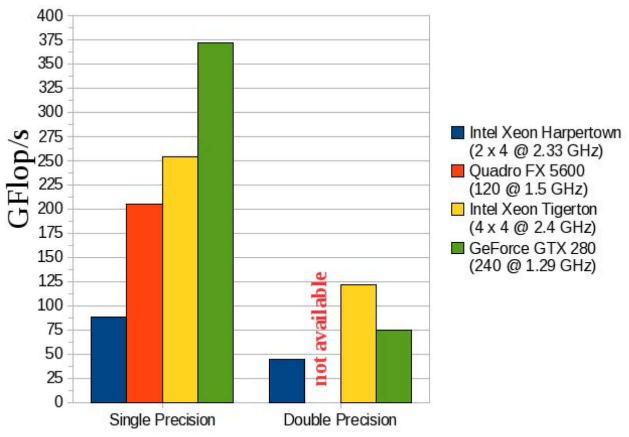


Fig. 1. Peak measured performances of matrix-matrix multiplications on current multicore (from Intel) and GPU (from NVIDIA) architectures.

ultimate goal of our work in this area is to achieve exactly that, namely to design linear algebra algorithms and frameworks for hybrid multi/manycores and GPUs systems that would exploit the power that each of them offers.

This paper is organized as follows. In Section 2, we give an overview on the GPUs used for HPC, their evolution and future trends, along with their current use for DLA. In Section 4.1, we concentrate on programming concepts for DLA on hybrid multicore/GPU systems. Then Section 4 gives an example of application, which further illustrates the concept of hybrid multicore/GPU computing for DLA. Finally, Section 5 gives conclusions and future directions.

2 GPUs for HPC

Driven by the demands of the game industry, graphics hardware has substantially evolved over the years to include both more functionality and programmability. This, combined with the graphics cards' impressive floating-point performance, have enabled and motivated their use in applications well beyond graphics. In this section, we give an overview of the GPUs evolution over the years along with future trends, and use in the area of dense linear algebra.

2.1 GPU Evolution and Future Trends

The game industry, and the large market that it enjoys, have pushed the GPUs over the years in excelling in graphics rendering. Graphics rendering can be described as an ordered sequence of graphics operations that are performed in a pipelined fashion, starting from a complex scene model until a final image is produced. In real-time rendering this pipelined computation has to be done fast over and over again. The limit for example is about 60 frames per second (fps) as rates above that are indistinguishable for the human eye, but for smooth movement about 30 is enough (e.g. TV has a refresh rate of 30 fps). It is clear that this type of computation

1. Requires an enormous computational power;
2. Allows for high parallelism, and
3. Stresses more on high bandwidth than low latency, as latency requirements can be compensated for by the deep graphics pipeline.

These three computational characteristics have marked the GPU evolution over the years, as described below. Moreover, as it is obvious that this pattern of computation is common with many other applications, GPUs' evolution has benefited a large number of applications, turning it into a General Purpose GPU (GPGPU), as often referred to in the literature.

Old graphics cards had a fixed function graphics pipeline, meaning that the operations and the sequence in which they were applied over the stream of data were configured on a very low level and were practically impossible to change by software developers. In August 1999, NVIDIA released the GeForce 256 card, which allowed a certain degree of programmability of its pipeline. In February 2001, NVIDIA released the GeForce 3 GPU, which is considered to be the first fully programmable GPU. Here fully programmable means that developers were able to provide their own transformations and lightning operations (vertex shaders) to be performed on vertices and their own pixel shaders to determine the final pixels color. Both the vertex and pixel shaders can be thought of as small programs which, when enabled, replace the corresponding fixed function pipeline. The programs are executed for every vertex/pixel and can change their attributes. Originally the vertex and pixel shaders had to be written in assembly language, but as the constantly increasing functionality provided by the graphics cards allowed more complex shaders to be written, higher level programming languages were developed, e.g. the High-Level Shading Language (HLSL), NVIDIA's Cg, etc. Moreover, as the GPUs seemed to be developing into more and more powerful programmable stream processors [22, chapter 29] (where the graphics data can be represented as streams and the shaders as kernels applied to each element of those streams), other high level languages emerged that concentrated on supporting a general purpose streaming programming model, and thus removing the need to know graphics in order to use GPUs for general purpose computations. Examples are the Brook [3] and Sh [15] languages along with their commercialized generalizations correspondingly in PeakStream and RapidMind.

Currently, GPU improvements continue, due to ever increasing computational requirements. Additionally, as better games mean not only **faster** but also more **realistic graphics**, or in other words more accurate and complex physics simulations, the requirements for improving GPUs' arithmetic precision has also been high. This need for more and more computational power, accuracy, and ability to implement complex simulations has pushed the GPUs development for higher speed, higher precision arithmetic, and more programmability to the point where current GPUs have reached a theoretical peak performance of 1 Tflop/s in single precision, support fully the IEEE double precision arithmetic standard [16], and have a programming model (e.g. see CUDA [17]) that according to some opinions may even revive the *quest for a free lunch* [12]. And indeed, CUDA, as an architecture and programming language, is not only easier to use but also have added and exposed to the user more hardware resources than what other languages, previous generations cards and even current NVIDIA competitors offer (like AMD). For example, CUDA extends the previous vision that GPUs are going to evolve towards more powerful stream processors [22], by providing not only the data parallelism inherent for stream processors, but also multithreading parallelism. CUDA provides also multiple levels of memory hierarchy, support for pointers, asynchronicity, etc [17]. These features have cemented even further the important role of GPUs in today's general purpose computing, and HPC use in accelerating real applications [19, 28, 29]. With the introduction of CUDA, software developers do not have to know about graphics in order to use GPUs for general purpose computing. As CUDA numerical libraries become rapidly available (e.g. CUDA FFT and BLAS libraries are included in the CUDA Toolkit) user may not even have to learn CUDA to benefit from the GPUs.

But as GPUs have moved "closer" to CPUs in terms of functionality and programmability, CPUs have also acquired functionality similar to the one in GPUs. For example Intel's SSE and PowerPC's AltiVec instructions offer a vector programming model similar to GPUs' (see the argument in [27] that modern GPUs should be viewed as multithreaded multicore vector units). Moreover, there are the *AMD Fusion* plans to integrate a CPU and GPU on a single chip, and other hybrids between multicore x86 and a GPU, as in Intel's recent announcement about the Larrabee system [24]. These trends, especially in view of the GPU's success in entering the general purpose computing market, will unquestionably stir contentions with CPU manufacturers, and add to the uncertainty to what the future architectures would look like. As of now it is not clear what part of a future computer can provide the crucial GPU functionality, or even if an increasingly important GPU with parallel computing capabilities could be part of a CPU [18]. It is clear though that future architectures will continue featuring hybrid designs where software designers would have to explore and use in their software both GPU and CPU features in order to fully exploit the underlying architectures.

More on the subject can be found e.g in [18].

2.2 GPUs for DLA

Due to the high ratio of floating point calculations to data required, many DLA algorithms have been of very high performance (e.g. close to the machine's peak) on standard CPU architectures. Therefore, although there has been some work in the field, special purpose architectures like GPUs, or even reconfigurable architectures like FPGAs, have not been able to significantly accelerate them up until recently. For example Fatahalian et al. [9] study SGEMM using shaders and their conclusion is that CPU implementations outperform most GPU implementations, and only the ATI X800XT produced comparable results (close to 12 GFlop/s) with a 3GHz Pentium 4. Similar results were produced by Galoppo et al [10] on LU factorization. Their results were in general outperformed by LAPACK routines using ATLAS on 3.4GHz Pentium IV. Again using shaders their best result on LU with partial pivoting was approximately 5.7 GFlop/s on an NVIDIA 7800.

But this has changed as CPUs move to multi/manycores with an exponentially growing gap between processor speed and memory (and bandwidth shared between cores), while GPUs have consistently outpaced them in both performance, which has been approximately doubling every year *vs* every year and a half for CPUs, and bandwidth/throughput (relying on deep pipeline, and sacrificing latency, as discussed in Subsection 2.1). A simple illustration of this fact can be seen on Figure 1 where we give the matrix-matrix multiplication performances of two modern multicore processors and two GPUs. Note that in single precision the GTX 280 is about an order of magnitude faster than a quad core processor (2.33 GHz) and still 75 GFlop/s faster even then a quad-socket quad-core Intel Xeon Tigerton processor (running at 2.4 GHz). In double precision the difference is not that distinct yet but still the GTX 280, being just the first card to support double precision arithmetic, outperforms a single quad-core (2.33 GHz) about 3 times.

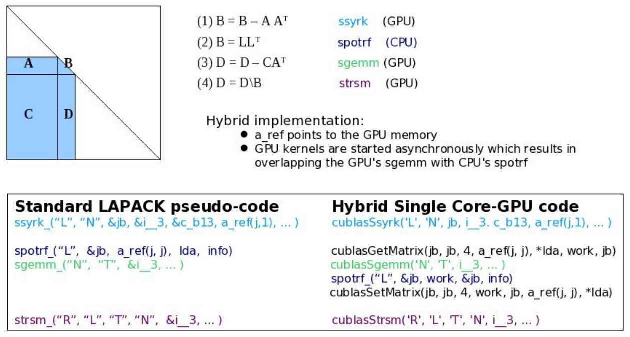
The first CUDA GPU results that significantly outperformed standard CPUs on single precision DLA started appearing at the beginning of 2008. To mention a few, in January, a poster by V. Volkov and J. Demmel [26] describes an SGEMM kernel (along with others) that significantly outperformed the one released by NVIDIA in the CUBLAS library (125 GFlop/s in CUBLAS *vs* more than 180 GFlop/s in their implementation in single precision). Moreover, the improved kernels were used in developing an LU factorization running at up to 140 GFlop/s. In March, S. Tomov [25] presented at PPSC08 a Cholesky factorization running at up to 160 GFlop/s in single precision using Volkov's sgemm kernel (later described in LAPACK Working Note 200 [2]). In May, V. Volkov and J. Demmel [27] described LU, QR, and Cholesky factorizations running at up to 180 GFlop/s in single precision (with QR a little bit more). The first results on a pre-released next generation G90 NVIDIA card were presented at UGC2008 in May, where Dongarra et al. [7] reported Cholesky running at up to 327 GFlop/s in single precision. Using again the newest generation card, in this paper, we describe an LU algorithm running at up to 331 GFlop/s in single precision and 70 Gflop/s in double precision when using a single core of the host CPU (and up to correspondingly 388 and 99.4 GFlop/s when using the entire host, a Intel Xeon 2 x 4 Cores @ 2.33 GHz).

The first results just described, formed the general understanding on how to program DLA using CUDA. Namely, there are three main ideas that define the approach:

1. Use BLAS level parallelism, where the matrix resides on the GPU, and the CPU is running for example LAPACK style code, e.g. represented as a sequence of CUBLAS kernels, using the GPU pointer arithmetic;
2. Offload small kernels, inefficient for the GPU to the CPU;
3. Use asynchronicity between CPU and GPU whenever possible in the offload/load process.

This is illustrated for Cholesky factorization (so called left-looking version) on Figure 2 (the case reported in [7]). The matrix to be factorized is allocated on the GPU memory and the code is as in LAPACK with BLAS calls replaced by CUBLAS, which represents the first idea from the list above. As steps two and three of the algorithm are independent, and the Cholesky factorization of B would have been inefficient for the GPU (small problem of size 128 x 128 for example, i.e. cannot have enough parallelism to utilize 240 cores GPU), B is offloaded and factorized on the GPU, which illustrates the second idea. Finally, steps 2 and 3 of the algorithm are done in parallel as calls to CUDA are asynchronous, namely as the CPU calls cublasSgemm the execution continues, i.e. to

SPOTRF, without waiting for the completion of cublasSgemm, which illustrates the third idea. In addition to overlapping just the computation, for cards that support it, sending B to the CPU and moving the result back could be overlapped with the GPU computation (of cublasSgemm in this case) when asynchronous copy calls are used. Note the ease of programming this algorithm while achieving an impressive performance.


A
B
C
D
(1) $B = B - A A^T$ ssyrk (GPU)
(2) $B = LL^T$ spotrf (CPU)
(3) $D = D - CA^T$ sgemm (GPU)
(4) $D = D\backslash B$ strsm (GPU)

Hybrid implementation:
- a_ref points to the GPU memory
- GPU kernels are started asynchronously which results in overlapping the GPU's sgemm with CPU's spotrf

Standard LAPACK pseudo-code	Hybrid Single Core-GPU code
ssyrk_("L", "N", &jb, &i__3, &c_b13, a_ref(j,1), ...)	cublasSsyrk('L', 'N', jb, i__3. c_b13, a_ref(j,1), ...)
spotrf_("L", &jb, a_ref(j, j), lda, info)	cublasGetMatrix(jb, jb, 4, a_ref(j, j), *lda, work, jb)
sgemm_("N", "T", &i__3, ...)	cublasSgemm('N', 'T', i__3, ...)
	spotrf_("L", &jb, work, &jb, info)
	cublasSetMatrix(jb, jb, 4, work, jb, a_ref(j, j), *lda)
strsm_("R", "L", "T", "N", &i__3, ...)	cublasStrsm('R', 'L', 'T', 'N', i__3, ...)

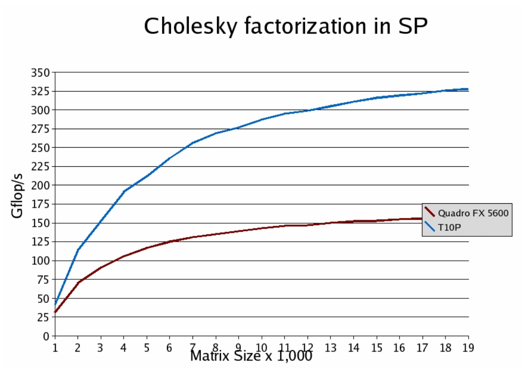


Fig. 2. Left Looking Cholesky factorization: implementation (Left) and performance running the algorithm on an NVIDIA T10P and NVIDIA Quadro FX 5600 (Right) in single precision arithmetic.

As a final remark in this section, we would like to point out that as DLA can still be efficiently implemented for multi/manycores it becomes increasingly interesting on using multi/manycores along with accelerators, where the algorithms are split and mapped between the two architectures in a way that better fits architectural with algorithmical requirements [7].

3 DLA for Hybrid Multicore/GPU Systems

No matter if designing DLA algorithms for multicores or GPUs, the requirements for efficient execution are the same, namely algorithms should be not only of **high parallelism** but also of **high enough ratio of floating point calculations to data required** to mask slow memory speeds. When we combine the two architectures, algorithms should in addition be properly split between the two, namely there should be load balancing throughout the execution, and a mapping of the computation in which the strengths of each platform are properly matched to the requirements of the algorithm. As these are complicated issues and there is no single solution for them we briefly outline below some of the main efforts in the area.

A way to provide algorithms of **high parallelism** is to split the computation in tasks and dependencies among them, leading for example to the concept of representing *algorithms as Directed Acyclic Graphs (DAGs)* where the nodes represent the sub-tasks and the edges the dependencies (described in Section 3.1). In Section 3.2 we present how to apply the *algorithms as DAGs* concept to hybrid systems.

Designing algorithms of **higher ratio of floating point calculations to data required** is a subject of current research in the field of DLA. A classical example is the transition from algorithms based on optimized Level 1 BLAS (from the LINPACK and EISPACK libraries) to reorganized DLA algorithms that use block matrix operations in their innermost loops, which actually formed LAPACK's design philosophy. Current examples include work on LU and QR factorizations, in particular in the so called *tiled* [4] and *communication avoiding* [11] algorithms. The LU algorithm described in Section 4 is yet another example.

Splitting algorithms into tasks, either within a single or hybrid architecture, raises also the question of properly scheduling those tasks for execution. The order is not important for correctness, as long as dependencies are not violated, but the benefits for performance can be significant as proper scheduling can ensure more asynchronicity and hiding/overlapping less efficient tasks with efficient ones. This is discussed more in Section 3.3.

3.1 Algorithms as DAGs

As architectures evolved from sequential to ones requiring more and more parallelism, e.g. multicores, it became evident that a fundamental concept in programming those new architectures will be a flexible control over the data and program execution flow scheduling. Instrumental in developing it is for example the concept of representing algorithms and their execution flows as DAGs, where, as already mentioned, nodes represent the sub-tasks and the edges the dependencies among them. Figure 3 gives a typical example of how a DLA algorithm may look like when represented as a DAG. The nodes in red in this case represent the sequential part of the algorithm and the ones in green the tasks that can be done in parallel. Ideally the scheduling should be asynchronous and dynamic, so that the tasks in red are overlapped, without violating any dependencies, with the tasks in green. This can be done for example by defining a "critical path", that is the most time-consuming sequence of basic operations that must be carried out sequentially even allowing for all possible parallelism, and scheduling for execution the tasks from the critical path as soon as possible (i.e. when all dependencies have been computed).

We used the concept of representing algorithms and their execution flows as DAGs in the context of developing DLA for multicores [5].

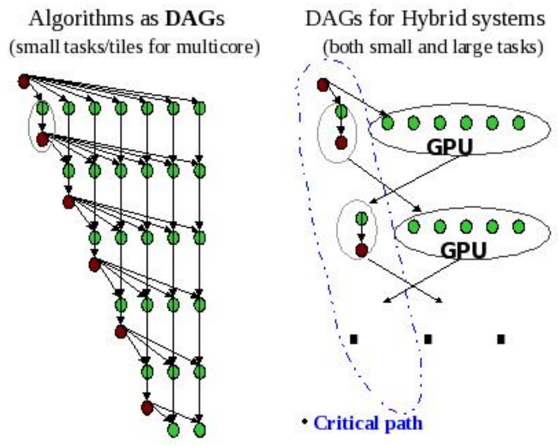


Fig. 3. Algorithms as DAGs.

3.2 DAGs for Hybrid Systems

The need for flexible control over the data and program execution flow scheduling for hybrid multicore/GPU systems is even higher than the one for multicore taken alone. Similarly to multicore, we can apply the DAGs concept to the hybrid case. One of the differences is the task granularity. For multicore, for example, small tasks worked well, in the so called tiled algorithms for multicore [4]. Here, as the GPU task would be one GPU kernel invocation (with number of cores 120 and higher per GPU, e.g. 240 in the GTX 280) we need the GPU tasks to be larger than in the multicore case, as shown on Figure 3, Right. Tasks from the critical path can be smaller and in general executed on the GPU's host.

3.3 Scheduling Tasks

The tasks scheduling is of crucial importance for the efficient execution of an algorithm. Proper scheduling, for example scheduling tasks from the critical path to be executed as soon as possible, results in techniques that have been used in the past. In particular these are the "look-ahead" techniques that have been extensively applied to the LU factorization. Such methods can be used to remedy the problem of synchronizations introduced by non-parallelizable tasks by overlapping their execution with the execution of more efficient ones [8]. It has been applied also in the context of GPUs in [27] as well as here. Illustration is given on Figure 3, Right, where for example if we overlap the execution of the 2 circled tasks on the critical path (by the host), with the execution of the green tasks circled and marked GPU (by the GPU), we get a hybrid version of the look-ahead technique.

4 An Example of DLA Algorithms for Hybrid Systems

To further motivate and illustrate the main ideas on the hybrid multicore/GPU approach for DLA, we give an example of algorithm for hybrid systems along with its performance results.

4.1 An LU Factorization Design for Hybrid Multicore/GPU Systems

We consider a right looking block LU factorization and design an algorithm for hybrid multicore/GPU systems. The approach is based on splitting the computation as shown on Figure 4. The numbers are under the assumption that the host has 8 cores, which is the case for our test system. Having an $N \times N$ matrix, the splitting is such that the first $N - 7nb$ columns reside on the

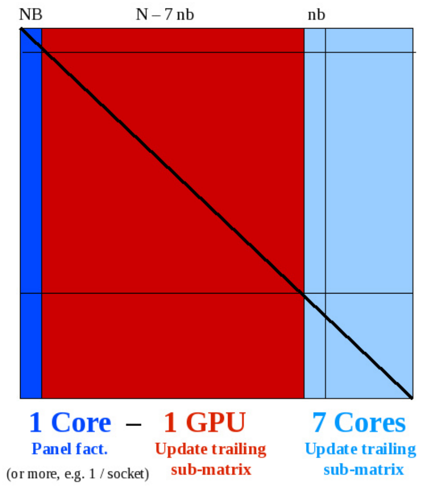

NB
N – 7 nb
nb
1 Core
–
1 GPU
7 Cores
Panel fact.
(or more, e.g. 1 / socket)
Update trailing sub-matrix
Update trailing sub-matrix

Fig. 4. Load splitting for a hybrid LU factorization.

GPU memory and the last $7nb$ on the host where nb is certain block size. One of the host cores and the GPU factor the first $N - 7nb$ columns of the matrix in the following way:

1. Current panel is downloaded to the CPU. For example the dark blue part of the matrix of NB columns, where NB is certain block size, is the panel for the first iteration.
2. The panel is factored on the CPU and the result is sent back to the GPU to update the trailing submatrix (colored in red for the first iteration).
3. The GPU first updates the first block column of the trailing matrix so that the CPU that processes the panels can proceed while the GPU updates the rest of the matrix (note that this is the look-ahead technique described in Section 3.3).

4. The rest of the host cores (7 in this case) update the last $7nb$ columns (one core per block of nb columns).
5. When the 1 Core- 1 GPU system finishes the factorization there is a synchronization with the other 7 cores, so that when they both finish their factorization/updates, a 1 Core- 1 GPU system finishes the factorization of the trailing $7bn \times 7nb$ matrix.

This algorithm is general enough to be applicable to many forms of LU factorizations, where the distinction can be made based on the form of pivoting that they employ.

4.2 The issue of pivoting in LU factorizations

Pivoting is a well-known technique to ensure stability in matrix algorithms. In particular, the commonly used method of Gaussian elimination (GE) with partial pivoting (PP) is implemented in current linear algebra libraries for solving square linear systems $Ax = b$ resulting in very stable algorithms. In the LAPACK [1] implementation of GE, during pivoting rows are swapped at once, which inhibits the exploitation of more asynchronicity between block operations.

In a recent paper, [11] describes a pivoting strategy that minimizes the number of messages exchanged during the panel factorization and shows that this approach is stable in practice. For multicore, pairwise pivoting (PwP) is often considered (e.g in [4]) but this generates a significant overhead since the rows are swapped in pairs of blocks. Still for multithreaded architectures, [23] describes an algorithm by blocks for LU factorization that uses a pivoting technique referred to as incremental pivoting based on principles used for out-of-core solvers.

For implementation of PP LU on GPUs, [27] designs an algorithm using innovative data structures where for example storing the matrix in row-major layout helps in reducing the pivoting overhead from 56% of the total factorization time to $1-10\%$ (depending on the machine and on the problem size).

In the following, we show the performance of our hybrid design using an extension of the technique proposed in [2]. Briefly, the approach in [2] follows the idea of [20, 21] to transform the original matrix into a matrix that would be sufficiently "random" so that, with probability close to 1, pivoting is not needed. These transformations are in general chosen as unitary because they are numerically stable and they keep the condition number of the matrix unchanged (when using the 2-norm). The random transformation proposed in [21] is based on the Discrete Fourier Transform and the transformation proposed in [20] is referred to as Random Butterfly Transformation (RBT) which consists of preconditioning a given matrix using particular random matrices referred to as butterfly matrices or products of them. We will refer to the resulting method as RBT NP LU. The easiest way to think of the method is as performing LU with no pivoting (NP) on a preconditioned matrix, where the cost of preconditioning is negligible compared to the cost of the factorization itself.

Similarly to the Cholesky factorization, where no pivoting is required for symmetric and positive definite matrices, the NP LU can be of direct use for diagonally dominant matrices as this case does not require pivoting. For general matrices, the RBT transformation helps but in general the accuracy is reduced and requires to add iterative refinement in the working precision (see [2] where solutions of linear systems using PP LU, RBT NP LU and QR are compared for some matrices from Higham's collection [13]). Another technique to improve the stability is to add "limited" pivoting (LP), e.g within the block size or more. In Figure 5, we compare for different matrix sizes the error in the LU factorization obtained when we do partial pivoting (PP LU), pairwise pivoting (PwP LU), limited pivoting (LP LU) and no pivoting at all (NP LU). For NP LU, we plot, for each matrix size, the maximum and minimum values for the norm of the residual for LU obtained for a sample of matrices. PP LU and PwP LU correspond to the LU factorization as it is implemented respectively in LAPACK and a preliminary version in PLASMA [4]. The accuracy of LP LU is computed when pivoting within the first NB rows and within the first $NB + 64$ rows of the panel (or less if this exceeds the rows in the panel). NP LU(NB + invert) corresponds to the case where we pivot within the first NB rows; the obtained L factor is explicitly inverted and the inverse is used in updating the trailing sub-matrix on the right of the current block. Note that the computational cost of adding limited pivoting is affordable because it does not change the Level 3 BLAS nature of the current

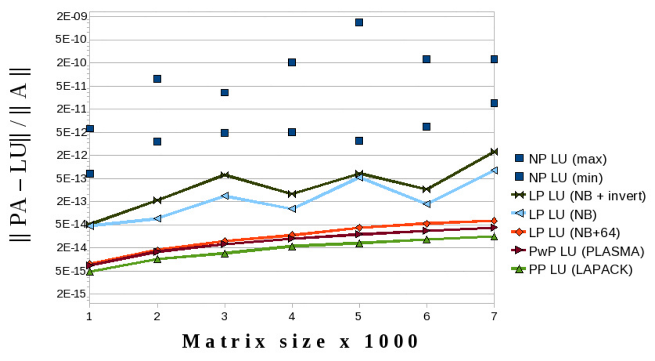


Fig. 5. Accuracy of double precision LU factorizations on matrices of $N(0,1)$ distribution.

implementation (performance results are given in the next section). More theoretical work is required to verify/prove that the proposed combination of a global 'preconditioner' (in this case the RBT transformation) and local/limited pivoting can lead to optimally reduced amount of pivoting while still preserving the accuracy of standard algorithms like the PP LU.

4.3 Performance and Numerical Results.

Here we give the performance of our implementation of the RBT NP/LP LU algorithms and put it in the context of other LU factorizations and their performances on current systems. The parameters nb and NB have to be set so that there is load balance. For this particular algorithm the work done by the core processing the panels is proportional to the work for the other cores so we take $nb = NB$. Figure 6 shows the performance results. On the Left we have the performance for single

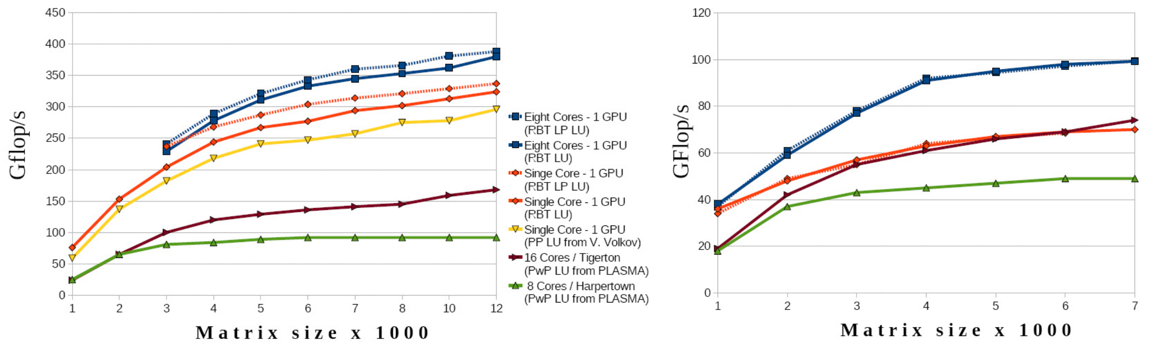


Fig. 6. Performance for the RBT LU algorithm on a hybrid Intel Xeon (2×4 @ 2.33 GHz) – GeForce GTX 280 (240 @ 1.30 GHz) for correspondingly single (Left) and double (Right) precision arithmetic.

precision arithmetic and on the right for double precision. The experiments are done on a hybrid Intel Xeon Harpertown processor (dual socked quad-core at 2.33 GHz) and an NVIDIA GeForce GTX 280 (240 Cores at 1.30 GHz). The algorithm denoted by 'RBT LP LU' is performing local pivoting within the block size and uses explicitly inverted lower triangular matrices resulting from the panel factorization to update the trailing matrix (as suggested in [27] for performance reasons). All the pivoting is done on the CPU. The pivoting and the following update on the submatrix on the right of the diagonal is replaced by flipping the corresponding rows of the explicitly inverted lower triangular matrix from the panel factorization (done on the CPU) and multiplying it by rectangular submatrix on the right of the current block. Note that the RBT LP LU is in general faster as well as more stable (see Figure 5) than the RBT NP LU.

Not just for the purpose of comparison, but more in order to put these results in the context of state-of-the-art results for homogenous multicore architectures, we have included also results from the pairwise pivoting LU from the PLASMA project for two multicore systems. Namely the GPU's host (Intel Xeon Harpertown) and an Intel Xeon Tigerton featuring quad-socket quad-cores at 2.4 GHz. Note that these are the same systems that we used to compare GEMM operations on Figure 1. In single precision we also compared the new algorithm with the best known so far 1 Core - 1 GPU code from V. Volkov.

Compared to V. Volkov's single precision PP LU our RBT NP LU code runs from 12% (for large matrices) to 26% (for small matrices) faster on 1 Core - 1 GPU system. In the context of just multicores, the hybrid implementation outperforms significantly even a 4 (socket) quad-cores system like the Intel's Tigerton (at 2.4GHz).

As already mentioned, one of the techniques to improve the method's accuracy is adding limited pivoting. The other technique to improve on the accuracy is to add iterative refinement in fixed precision. The cost of adding it can also be reduced by having explicitly available triangular matrix inverses that are byproduct of the factorization. For example, we developed blocked CUDA STRSV-like and DTRSV-like routines that replace triangular solves (within the block) with matrix multiplication to get a performance of correspondingly up to 14 GFlop/s (for matrix of size $14,000$) and 6.7 GFlop/s (for matrix of size $7,000$). Note that just using cublasStrsm or cublasDtrsm the performance would be correspondingly 0.24 GFlop/s and 0.09 GFlop/s, which will result in iterating cost that is higher than negligible compared to the factorization (cublasStrsv and cublasDtrsv are about 5 times faster but the matrix size should not exceed correspondingly 4070 and 2040). We note that these routines can be used for mixed-precision iterative refinement solvers [6, 14] as well, where iterating on the GPU would have negligible cost compared to the factorization itself.

5 Conclusions and Future Directions

GPUs have already evolved and quickly pass the point where many real world applications not only can be easily implemented for GPUs but also to significantly outperform current multicore systems. Still, there are applications – or at least parts of them – that do not map well to the GPU architecture, and would benefit much more of a hybrid architecture. We demonstrated that this is the case for DLA where the computation can be split in a way that would better exploit the power that each of the hybrid components offer. We made an overview describing this move to hybrid architectures, where major CPU manufactures start to include more GPU functionality in their designs, and where GPU manufacturers more CPU functionality. It's clear that both functionalities are needed, but it is not clear if any of them, and if yes, which one, could absorb the other. It is clear though that future architectures will continue featuring hybrid designs where software designers would need to explore and use in their software both GPU and CPU features in order to fully exploit the underlying architectures. Therefore, we envision that future commodity architectures would be hybrid systems that incorporate both GPU and CPU functionalities. Furthermore, as GPUs are getting more and more widely used in HPC the need for DLA for hybrid systems would grow. As DLA software development for multi/manycores, GPUs, as well as hybrids is still in its infancy, the area is wide open. This paper is just a first step in describing and giving a motivating example for the benefits of hybrid systems, motivating further work towards creating a self contained DLA library similar to LAPACK but for hybrid manycore/GPU systems.

Acknowledgments. Part of this work was supported by the U.S. National Science Foundation, and the U.S. Department of Energy. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations. We thank also Jim Demmel and Vasily Volkov from UC Berkeley, and Massimiliano Fatica from NVIDIA for helpful discussions related to GPU computing.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.

2. Marc Baboulin, Jack Dongarra, and Stanimire Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures*, Technical Report UT-CS-08-615, University of Tennessee, 2008, LAPACK Working Note 200.
3. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan, *Brook for GPUs: stream computing on graphics hardware*, ACM Trans. Graph. **23** (2004), no. 3, 777–786.
4. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Technical Report UT-CS-07-600, University of Tennessee, 2007, LAPACK Working Note 191.
5. Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, and Stanimire Tomov, *The impact of multicore on math software*, In PARA 2006, Umea Sweden, 2006.
6. Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov, *Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy*, ACM Trans. Math. Softw. **34** (2008), no. 4.
7. Jack Dongarra, Shirley Moore, Gregory Peterson, Stanimire Tomov, Jeff Allred, Vincent Natoli, and David Richie, *Exploring new architectures in accelerating CFD for Air Force applications*, Proceedings of HPCMP Users Group Conference 2008 (July 14-17, 2008), http://www.cs.utk.edu/˜tomov/ugc2008_final.pdf.
8. Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet, *The linpack benchmark: Past, present, and future*, Concurrency and Computation: Practice and Experience **15** (2003), 820.
9. K. Fatahalian, J. Sugerman, and P. Hanrahan, *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*, HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA), ACM, 2004, pp. 133–137.
10. Nico Galoppo, Naga Govindaraju, Michael Henson, and Dinesh Manocha, *LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware*, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2005, p. 3.
11. L. Grigori, J. W. Demmel, and H. Xiang, *Communication avoiding Gaussian elimination*, Technical Report 6523, INRIA, 2008.
12. Wolfgang Gruener, *Larrabee, CUDA and the quest for the free lunch*, http://www.tgdaily.com/content/view/38750/113/, 08/2008, TGDaily.
13. N. J. Higham, *Accuracy and stability of numerical algorithms*, SIAM, 2002, Second edition.
14. Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra, *Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)*, sc **0** (2006), 50.
15. Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule, *Shader algebra*, SIGGRAPH '04: ACM SIGGRAPH 2004 Papers (New York, NY, USA), ACM, 2004, pp. 787–795.
16. NVIDIA, *Nvidia Tesla doubles the performance for CUDA developers*, Computer Graphics World (06/30/2008).
17. NVIDIA, *NVIDIA CUDA Programming Guide*, 6/07/2008, Version 2.0.
18. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, *GPU computing*, Proceedings of the IEEE **96** (2008), no. 5, 879–899.
19. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell, *A survey of general-purpose computation on graphics hardware*, Computer Graphics Forum **26** (2007), no. 1, 80–113.
20. D. S. Parker, *Random butterfly transformations with applications in computational linear algebra*, Technical Report CSD-950023, Computer Science Department, UCLA, 1995.
21. D. S. Parker and B. Pierce, *The randomizing FFT: an aternative to pivoting in Gaussian elimination*, Technical Report CSD-950037, Computer Science Department, UCLA, 1995.
22. Matt Pharr and Randima Fernando, *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*, Addison-Wesley Professional, 2005.
23. G. Quintana-Orti, E. S. Quintana-Orti, E. Chan, F. G. van Zee, and R. A. van de Geijn, *Programming algorithms-by-blocks for matrix computations on multithreaded architectures*, Technical Report TR-08-04, University of Texas at Austin, 2008, FLAME Working Note 29.
24. Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan, *Larrabee: a many-core x86 architecture for visual computing*, ACM Trans. Graph. **27** (2008), no. 3, 1–15.
25. S. Tomov, M. Baboulin, J. Dongarra, S. Moore, V. Natoli, G. Peterson, and D. Richie, *Special-purpose hardware and algorithms for accelerating dense linear algebra*,

http://www.cs.utk.edu/˜tomov/PP08_Tomov.pdf, Parallel Processing for Scientific Computing, Atlanta, March 12-14, 2008.

26. V. Volkov and J. W. Demmel, *Using GPUs to accelerate linear algebra routines*, Poster at PAR lab winter retreat, January 9, 2008, http://www.eecs.berkeley.edu/˜volkov/volkov08-parlab.pdf.
27. Vasily Volkov and James Demmel, *LU, QR and Cholesky factorizations using vector capabilities of GPUs*, Tech. Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.
28. *General-purpose computation using graphics hardware*, http://www.gpgpu.org.
29. *Nvidia cuda zone*, http://www.nvidia.com/object/cuda_home.html, NVIDIA.