Scalable Dense Linear Algebra on Heterogeneous Hardware

George Bosilca [a] Aurelien Bouteiller [a] Anthony Danalis [a] Thomas Herault [a]
Jakub Kurzak [a] Piotr Luszczek [a] Stanimire Tomov [a] Jack J. Dongarra [a,b,c]

[a] *Innovative Computing Laboratory – The University of Tennessee Knoxville*
[b] *Oak Ridge National Laboratory*
[c] *University of Manchester*

Abstract. Design of systems exceeding 1 Pflop/s and the push toward 1 Eflop/s, forced a dramatic shift in hardware design. Various physical and engineering constraints resulted in introduction of massive parallelism and functional hybridization with the use of accelerator units. This paradigm change brings about a serious challenge for application developers, as the management of multicore proliferation and heterogeneity rests on software. And it is reasonable to expect, that this situation will not change in the foreseeable future. This chapter presents a methodology of dealing with this issue in three common scenarios. In the context of shared-memory multicore installations, we show how high performance and scalability go hand in hand, when the well-known linear algebra algorithms are recast in terms of Direct Acyclic Graphs (DAGs), which are then transparently scheduled at runtime inside the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) project. Similarly, Matrix Algebra on GPU and Multicore Architectures (MAGMA) schedules DAG-driven computations on multicore processors and accelerators. Finally, Distributed PLASMA (DPLASMA), takes the approach to distributed-memory machines with the use of automatic dependence analysis and the Direct Acyclic Graph Engine (DAGuE) to deliver high performance at the scale of many thousands of cores.

1. Introduction and Motivation

Among the various factors that drive the momentous changes occurring in the design of microprocessors and high end systems [29], three stand out as especially notable:

1. the number of transistors per chip will continue the current trend, i.e. double roughly every 18 months, while the speed of processor clocks will cease to increase;
2. the physical limit on the number and bandwidth of the CPU pins is becoming a near-term reality;
3. a strong drift toward hybrid and/or heterogeneous systems for Peta-scale and beyond is taking place.

While the first two involve fundamental physical limitations that current technology trends are unlikely to overcome in the near term, the third is an obvious consequence of the first two, combined with the economic necessity of using many thousands of computational units to scale up to Peta-scale and beyond.

More transistors and slower clocks require multicore designs and an increased parallelism. The fundamental laws of traditional processor design – increasing transistor density, speeding up clock rate, lowering voltage – have now been stopped by a set of physical barriers: excess dissipation of heat, excessive power consumption, too much energy leaked, and useful signals being overcome by noise. Multicore designs are a natural evolutionary response to this situation. By putting multiple processor cores on a single die, chip architects can overcome the previous limitations, and continue to increase the number of gates per chip without increasing the power densities. However, since excess heat production means that frequencies cannot be further increased, deep-and-narrow pipeline models will tend to recede as shallow-and-wide pipeline designs become the norm. Moreover, despite obvious similarities, multicore processors are not equivalent to multiple-CPUs or to SMPs. Multiple cores on the same chip can share various caches (including TLB – Translation Look-aside Buffer), while competing for memory bandwidth. Extracting performance from such configurations of resources means, that programmers must exploit increased thread-level parallelism (TLP), and efficient mechanisms for inter-processor communication and synchronization to manage resources effectively. The complexity of fine grain parallel processing will no longer be hidden in hardware by a combination of increased instruction level parallelism (ILP) and pipeline techniques, as it was with the superscalar designs. It will have to be addressed at an higher level: software; either directly in the context of the applications or in the programming environment. As code and performance portability remain essential, the programming environment has to change drastically.

A thicker memory wall means, that communication efficiency becomes crucial. The pins that connect the processor to main memory have become a strangle point, which, with both the rate of pin growth and the bandwidth per pin slowing down, is not flattening out. Thus the processor to memory performance gap, which is already approaching a thousand cycles, is expected to grow by 50% per year according to some estimates [38]. At the same time, the number of cores on a single chip is expected to continue to double every 18 months, and since limitations on space will keep the cache resources from growing as quickly, cache-per-core ratio will continue to diminish. Problems with memory bandwidth and latency, and cache fragmentation will, therefore, tend to become more severe, and that means that communication costs will present an especially notable problem. To quantify the growing cost of communication, we can note that time per flop, network bandwidth (between parallel processors), and network latency are all improving, but at significantly different rates: 59%/year, 26%/year and 15%/year, respectively [38]. Therefore, it is expected to see a shift in algorithms' properties, from computation-bound, i.e. running close to the peak performance today, toward communication-bound in the near future. The same holds for communication between levels of the memory hierarchy: memory bandwidth is improving 23%/year, and memory latency only 5.5%/year. Many familiar and widely used algorithms and libraries will become obsolete, especially dense linear algebra algorithms which try to fully exploit all these architecture parameters. They will need to be reengineered and rewritten in order to fully exploit the power of the new architectures.

In this context, the PLASMA project [47] has developed new algorithms for dense linear algebra on shared memory systems based on tile algorithms. Widening the scope of these algorithms from shared to distributed memory, and from homogeneous architectures to heterogeneous ones, has been the focus of a follow-up project, DPLASMA. DPLASMA introduces a novel approach to schedule dynamically dense linear algebra algorithms on distributed systems. Similarly to PLASMA, to whom it shares most of the mathematical algorithms, it is based on tile algorithms, and takes advantage of DAGUE [14], a new generic distributed Direct Acyclic Graph Engine for high performance computing. The DAGUE engine features a DAG representation independent of the problem-size, overlaps communications with computation, prioritizes tasks, schedules in an architecture-aware manner and manages micro-tasks on distributed architectures featuring heterogeneous many-core nodes. The originality of this engine resides in its capability to translate a sequential nested-loop code into a concise and synthetic format, which it can interpret and then execute in a distributed environment. We consider three common dense linear algebra algorithms, namely: Cholesky, LU and QR factorizations, part of the DPLASMA library, to investigate through the DAGUE framework their data driven expression and execution in a distributed system. It has been demonstrated, through performance results at scale, that this approach has the potential to bridge the gap between the peak and the achieved performance, that is characteristic in the state-of-the-art distributed numerical software on current and emerging architectures. However, one of the most essential contributions, in our view, is the simplicity with which new algorithmic variants may be developed and how they can be ported to a massively parallel heterogeneous architecture without much consideration, at the algorithmic level, of the underlying hardware structure or capabilities. Due to the flexibility of the underlying DAG scheduling engine and the powerful expression of parallel algorithms and data distributions, the DAGUE environment is able to deliver a significant percentage of the peak performance, providing a high level of performance portability.

2. Multithreading in the PLASMA Library

Parallel Linear Algebra Software for Multicore Architectures (PLASMA) is a numerical software library for solving problems in dense linear algebra on systems of multicore processors and multi-socket systems of multicore processors [2]. PLASMA offers routines for solving a wide range of problems in dense linear algebra, such as: non-symmetric, symmetric and symmetric positive definite systems of linear equations, least square problems, singular value problems and eigenvalue problems (currently only symmetric eigenvalue problems). PLASMA solves these problems in real and complex arithmetic and in single and double precision. PLASMA is designed to give high efficiency on homogeneous multicore processors and multi-socket systems of multicore processors. As of today, the majority of such systems are on-chip symmetric multiprocessors with classic *super-scalar* processors as their building blocks (x86 and alike) augmented with short-vector SIMD extensions (SSE, AVX, and the like). PLASMA has been designed to supersede LAPACK [4], principally by restructuring the software to achieve much greater efficiency on modern computers based on multicore processors.

The interesting part of PLASMA from the mutithreading perspective is the variety of scheduling mechanism utilized by PLASMA. In the next subsection, the main design principles of PLASMA are introduced. Section 2.3 discusses the different multithreading mechanisms employed by PLASMA. Section 2.4.1 introduces PLASMA's most powerful mechanism of dynamic runtime task scheduling with the QUARK scheduler and briefly iterates through QUARK's extensions essential to the implementation of a production-quality numerical software. Section 2.4.2 highlights the advantages of dynamic scheduling for parallel task composition by showing how PLASMA implements explicit matrix inversion using the Cholesky factorization. Section 2.4.3 covers the concept of task aggregation when large clusters of tasks are offloaded to a GPU accelerator. Finally, section 2.4.4 discusses the very important case of using QUARK for exploiting nested parallelism.

2.1. Design Principles behind PLASMA

The main motivation behind the PLASMA project are performance shortcomings of LAPACK [4] and ScaLAPACK [9] on shared memory systems, specifically systems consisting of multiple sockets of multicore processors. The three crucial elements that allow PLASMA to achieve performance greatly exceeding that of LAPACK and ScaLAPACK are: the implementation of *tile algorithms*, the application of *tile data layout*, and the use of *dynamic scheduling*. Although some performance benefits can be delivered by each one of these techniques on its own, it is only the combination of all of them that delivers maximum performance and highest hardware utilization.

Tile data layout insists on storing the matrix by square tiles of relatively small size, such that each tile occupies a continuous memory region, and a tile fits entirely in one of the cache levels associated with a single core. This way a tile can be loaded to the cache memory efficiently and the risk of evicting it from the cache memory before it is completely processed is minimized. Of the three types of cache misses, *compulsory*, *capacity* and *conflict*, the use of tile layout minimizes the number of conflict misses, since a continuous region of memory will completely fill out a set-associative cache memory before an eviction can happen. Also, from the standpoint of multithreaded execution, the probability of *false sharing* is minimized. It can only affect the cache lines containing the beginning and the ending of a tile.

Dynamic scheduling assigns work to cores based on the availability of data for processing at any given point in time and is also referred to as *data-driven* scheduling. This concept is related closely to the idea of expressing computation through a task graph, often referred to as the DAG (*Direct Acyclic Graph*), and the flexibility of exploring the DAG at runtime. Thus, to a large extent, dynamic scheduling is synonymous with *runtime scheduling*. An important concept here is the one of the *critical path*, which defines the upper bound on the achievable parallelism. This is in direct opposition to the *fork-and-join* or *data-parallel* programming models, where artificial synchronization points expose serial sections of the code, where multiple cores are idle, while sequential processing takes place.

2.2. PLASMA Software Stack

Starting from the PLASMA, Version 2.2, released in July 2010, the library is built on top of standard software components, all of which are either available as open source or are standard OS facilities. Some of them can be replaced by packages provided by hardware vendors for efficiency reasons. Figure 1 presents the current structure of PLASMA's software stack. Following is a brief bottom-up description of individual components.

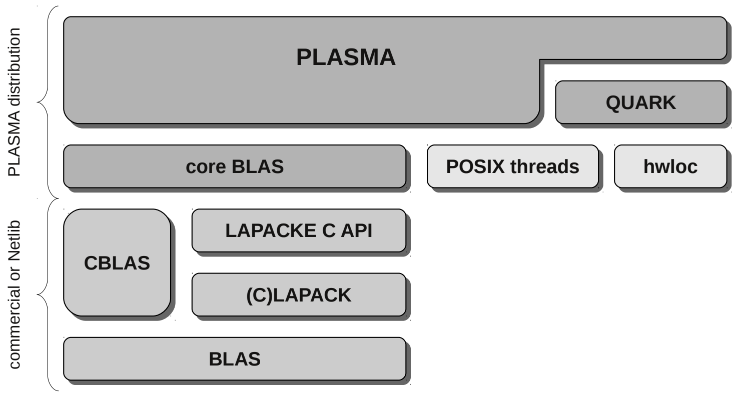


Figure 1. The software stack of PLASMA, Version 2.3.

Basic Linear Algebra Subprograms (BLAS) [7] is a *de facto* standard and a set of basic linear algebra operations, such as vector and matrix multiplication. CBLAS is the C language interface to BLAS [18]. Most commercial and academic implementations of BLAS also provide CBLAS. *Linear Algebra PACKage* (LAPACK) [4] is a software library for numerical linear algebra, a direct predecessor of PLASMA, providing routines for solving linear systems of equations, linear least square problems, eigenvalue problems and singular value problems. CLAPACK [19] is a version of LAPACK available from Netlib, created by automatically translating FORTRAN LAPACK to C with the help of the F2C [24] utility. It provides the same, FORTRAN, calling convention as the "original" LAPACK. LAPACKE C API [31] is a proper C language interface to LAPACK (or CLAPACK).

A set of serial kernels called "core BLAS" comprises a building block for PLASMA's parallel algorithms. PLASMA scheduling mechanisms coordinate the execution of these kernels in parallel on multiple cores. PLASMA relies on POSIX threads for access to the systems multithreading capabilities and on the hwloc [28] library for the control of thread affinity. PLASMA employs *static scheduling*, where threads have their work statically assigned and coordinate execution through progress tables, but can also rely on the QUARK [50] scheduler for dynamic runtime scheduling of work to threads.

2.3. Multithreading in PLASMA

PLASMA has to be initialized with the call to the `PLASMA_Init` function, before any calls to its computational routines can be made. When the user's thread (also referred to as the master thread) calls the `PLASMA_Init` function and specifies N as the number of cores to use, PLASMA launches $N-1$ additional (worker) threads and puts them in a waiting state. In the master thread, control returns to the user. When the user calls PLASMA's computational routine, the worker threads are woken up and all the threads, including the user's master thread, enter the routine. Upon completion, worker threads return back to the waiting state, while the master thread returns from the call. A call to the `PLASMA_Finalize` function terminates the worker threads.

PLASMA is *thread-safe*. Multiple PLASMA instances, referred to as *contexts* may remain active at the same time without conflicts, with the restriction that one user thread can be associated with one context only, i.e., one thread can only call `PLASMA_Init` once, before it calls `PLASMA_Finalize`. Currently contexts are managed implicitly. Context handle is not provided to the user. The typical usage scenario is to have a serial code (with a single thread) and launch one instance of PLASMA to use all the cores in the system. However, it is also possible for the user to spawn, e.g., four threads, each of which creates a four-thread instance of PLASMA, in order to exploit 16-way parallelism. In such a case, each PLASMA instance synchronizes its four threads, while the user has to manage synchronization among the four PLASMA instances. PLASMA provides mechanisms for managing *thread affinity*, i.e. controlling the placement of threads on the physical cores.

PLASMA's statically scheduled routines follow the *Single Program Multiple Data* (SPMD) programming paradigm. Each thread knows its thread ID (`PLASMA_RANK`) and the total number of threads within the context (`PLASMA_SIZE`) and follows a specific execution path based on that information. Synchronization is implemented though shared progress tables and busy waiting. If a thread cannot progress, it yields the core to the OS with a POSIX function (`sched_yield`). PLASMA's dynamically scheduled routines follow the *superscalar* programming paradigm, whereby the code is written sequentially and parallelized at runtime through the analysis of the flow of data between different tasks. When a static routine is called, all the threads simply enter the SPMD code of the routine. When a dynamic routine is called, the master thread enters the superscalar routine and begins queueing tasks using QUARK's task queueing calls. At the same time, all the worker threads enter QUARK's *worker loop*, where they keep executing the queued tasks until notified by the master. The master also participates in the execution of tasks, unless overwhelmed with the job of queueing.

PLASMA's computational routines are implemented using either static or dynamic scheduling or both. PLASMA provides a capability of switching at runtime between static or dynamic scheduling. If a sequence of user's calls invokes a mixed set of routines with static implementations only and dynamic implementation only, PLASMA switches the scheduling mode automatically. Statically scheduled routines are separated with barriers and each switch from static scheduling to dynamic scheduling, and vice versa. This invokes a global barrier which blocks all threads in a given PLASMA context. However, any continuous sequence of dynamically scheduled routines is free of barriers.

2.4. *PLASMA Scheduling*

By now, multicore processors are ubiquitous in both low-end consumer electronics and high-end servers and supercomputer installations. This led to the emergence of a large number of multithreading frameworks, both academic and commercial. Each one embraces the idea of task scheduling. They include: Cilk [11], OpenMP (with its tasking features) [42], Intel Threading Building Blocks [45]. These are just a few prominent examples in widespread use. One especially important category are multithreading systems based on *dataflow* principles, which represent the computation as a *Direct Acyclic Graph* (DAG) and schedule tasks at runtime through resolution of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). PLASMA's scheduler – QUARK – is an example of such a scheduling system. Two other, very similar, academic projects are also available: StarSs [6] from Barcelona Supercomputer Center and StarPU [5] from INRIA Bordeaux. While all three systems have their strength and weaknesses, QUARK has vital extensions for use in a numerical library such as PLASMA.

The QUARK scheduler targets multicore, multi-socket shared memory systems. The main design principle behind QUARK is the implementation of the dataflow model, which makes the scheduling decision to honor data dependencies between tasks in the complete task graph. The second principle is constrained use of resources, with bounds on space and time complexity. The dataflow model is implemented through analysis of data hazards, discussed later on. The constrained use of resources is accomplished by exploration of the task graph through a *sliding window* technique.

2.4.1. Dynamic Scheduling with QUARK

Even relatively small problems in dense linear algebra can easily generate DAGs with hundreds of thousands or even millions of tasks. Generation and exploration of the entire DAG of such size would not be feasible due to the aforementioned constrain on space complexity. Instead, as execution proceeds, tasks are continuously generated and executed. At any given point in time, only a relatively small number of tasks (on the order of one thousand) is stored in the task pool. The size of the sliding window is a tunable parameter. This allows for trading the time and space overhead for scheduling flexibility.

Parallelization using QUARK relies on two steps: transforming function calls to task definitions and replacing function calls with task queueing constructs. Figure 2 shows how PLASMA defines QUARK *dgemm* (matrix multiply) task using a call to CBLAS. Similarly to Cilk and SMPSs, functions implementing parallel tasks must be side-effect free (cannot use global variables, etc.) The task definition takes QUARK handle as its only parameter, declares all arguments as local variables, fetches them from QUARK using the `quark_unpack_args_` construct and executes the work (which in this example is just calling the `cblas_dgemm` function).

```
void CORE_dgemm_quark(Quark *quark) {
    int transA, transB;
    int m, n, k;
    double alpha, beta;
    double *A, *B, *C;
    int lda, ldb, ldc;

    quark_unpack_args_13(quark, transA, transB, m, n, k,
                         alpha, A, lda, B, ldb, beta, C, ldc);
    cblas_dgemm( CblasColMajor,
                 (CBLAS_TRANSPOSE)transA, (CBLAS_TRANSPOSE)transB,
                  m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);
}
```

Figure 2. QUARK dgemm (matrix multiply) task definition in PLASMA.

In order to change a function call to a task invocation, one needs to: replace the function call with a call to `QUARK_Insert_Task()`, pass the task name (pointer) as the first parameter, precede each parameter with its size and follow with direction. Array (or pointer) arguments are preceded with the size of memory they occupy, in bytes, and followed with one of the directions: `INPUT`, `OUTPUT` or `INOUT`. Scalar arguments are passed by a reference (pointer), preceded with the size of their data type, and followed by `VALUE` in place of the direction. Although scalar arguments are passed by reference, passing of scalars has the *pass by value* semantics. (A copy of each scalar argument is made at the time of call to `Insert_Task()`.)

```
QUARK_Insert_Task(quark, CORE_dgemm_quark, task_flags,
    sizeof(PLASMA_enum),        &transA,    VALUE,
    sizeof(PLASMA_enum),        &transB,    VALUE,
    sizeof(int),                &m,         VALUE,
    sizeof(int),                &n,         VALUE,
    sizeof(int),                &k,         VALUE,
    sizeof(double),             &alpha,     VALUE,
    sizeof(double)*nb*nb,        A,                  INPUT,
    sizeof(int),                &lda,       VALUE,
    sizeof(double)*nb*nb,        B,                  INPUT,
    sizeof(int),                &ldb,       VALUE,
    sizeof(double),             &beta,      VALUE,
    sizeof(double)*nb*nb,        C,                  INOUT,
    sizeof(int),                &ldc,       VALUE,
    0);
```

Figure 3. QUARK dgemm (matrix multiply) task invocation (queueing) in PLASMA.

QUARK schedules tasks at runtime through the resolution of data hazards (dependencies): *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). The *Read After Write* (RaW) hazard, often referred to as the *true dependency*, is the most common dependency. It defines the relation between a task writing ("creating") the data and the task reading ("consuming") the data. In this case, the latter task has to wait until the former task completes.

The *Write After Read* (WaR) hazard is caused by a situation, where a task attempts to write data before a preceding task is finished reading the data. In such case, the writer has to wait until the reader completes. The dependency is not referred to as a true dependency, because it can be eliminated by renaming of the data. The renaming is accomplished by making an explicit copy in the main memory. Although the dependency is unlikely to appear often in dense linear algebra, it has been encountered and has to be handled by the scheduler to ensure correctness.

The *Write After Write* (WaW) hazard is caused by a situation, where a task attempts to write data before a preceding task is finished writing the data. The final result is expected to be the output of the latter task, but if the dependency is not preserved (and the former task completes after the latter one), incorrect output will result. This is an important dependency in hardware design of processor pipelines, where resource contention can be caused by a limited number of registers. The situation is, however, quite unlikely for a software scheduler, where the occurrence of the WaW hazard means, that some data is produced and overwritten before it is consumed. As was the case for the WaR hazard, the WaW hazard can be removed by renaming.

2.4.2. Parallel Composition

One vital feature of QUARK, or any other superscalar scheduler, is the *parallel composition*, i.e., the ability to construct a larger task graph from a set of smaller task graphs. The benefit comes from exposing of more parallelism in the combined task graph than each of the components possess alone. PLASMA's routine for computing an inverse of a symmetric positive definite matrix is a great example [1].

The appropriate direct method to compute the solution of a symmetric positive definite system of linear equations consists of computing the Cholesky factorization of that matrix and then solving the resulting triangular systems. It is not recommended to use the inverse of a matrix in this case. However, some applications need to explicitly form the inverse of the matrix. A canonical example is the computation of the variance-covariance matrix in statistics. Higham [27, p.260,§3] lists more such applications.

The matrix inversion presented here follows closely the one in LAPACK and ScaLAPACK. The inversion is performed in-place, i.e., the data structure initially containing matrix A is gradually overwritten with the result and eventually A^{-1} replaces A. (No extra storage is used.) The algorithm involves three steps: computing the Cholesky factorization ($A = LL^T$), inverting the L factor (computing L^{-1}) and, finally, computing the inverse matrix $A^{-1} = L^{-T}L^{-1}$. In LAPACK the three steps are performed by the functions: POTRF, TRTRI and LAUUM. In PLASMA the steps are performed by functions which process the matrix by tiles define the work in terms of tile operations (tile kernels), and use QUARK to queue, schedule and execute the kernels. Figure 4 shows an excerpt of PLASMA implementation of the three functions. Each one includes four loops with three levels of nesting. All work in expressed through elementary BLAS operations encapsulated in `core_blas` kernels: POTRF, TRSM, SYRK, GEMM, TRTRI, TRMM, LAUUM.

The scheduler addresses three important problems in parallel software development: complexity, productivity, and performance. A superscalar scheduler eliminates the complexity of writing parallel software, by automatically parallelizing algorithms defined sequentially and guaranteeing parallel correctness of sequentially expressed algorithms. For some workloads in dense linear algebra, manual parallelization is relatively straightforward. A good example here is the Cholesky factorization (when considered alone) and its statically scheduled implementation in PLASMA [30]. For other operations this becomes nontrivial. Designing a static schedule for the combined three operations of the matrix inversion would be much harder. Finally, for some operations, it becomes prohibitively complex. A good example here are PLASMA routines for band reductions through bulge chasing [34,33].

Another benefit of the scheduler is productivity. Thanks to the fact, that sequential correctness guarantees parallel correctness due to the superscalar approach, when the data dependencies are correctly followed. the scheduler facilitates very rapid development of numerical software. Whereas manual design of parallel codes is labor intensive and error prone, causing long development times. Yet another benefit is the ability to do rapid prototyping of new algorithms and analysis of their parallel performance without the tedious work of parallelization. Finally, the scheduler also provides for increased performance, by identifying parallel scheduling opportunities, where a human programmer would miss them. Also, it is resilient to fluctuations in task execution time, OS jitter and adverse effects of resource sharing. Such adverse effects will cause a graceful degradation rather than a catastrophic performance loss.

```
void plasma_pdpotrf_quark(...) {
    for (k = 0; k < M; k++) {
        QUARK_CORE_dpotrf(...);
        for (m = k+1; m < M; m++)
            QUARK_CORE_dtrsm(...);
        for (m = k+1; m < M; m++) {
            QUARK_CORE_dsyrk(...);
            for (n = k+1; n < m; n++)
                QUARK_CORE_dgemm(...);
        }
    }
}
void plasma_pdtrtri_quark(...) {
    for (n = 0; n < N; n++) {
        for (m = n+1; m < M; m++)
            QUARK_CORE_dtrsm(...);
        for (m = n+1; m < M; m++)
            for (k = 0; k < n; k++)
                QUARK_CORE_dgemm(...);
        for (m = 0; m < n; m++)
            QUARK_CORE_dtrsm(...);
        QUARK_CORE_dtrtri(...)
    }
}
void plasma_pdlauum_quark(...) {
    for (m = 0; m < M; m++) {
        for(n = 0; n < m; n++) {
            QUARK_CORE_dsyrk(...);
            for(k = n+1; k < m; k++)
                QUARK_CORE_dgemm(...);
        }
        for (n = 0; n < m; n++)
            QUARK_CORE_dtrmm(...);
        QUARK_CORE_dlauum(...);
    }
}
```

Figure 4. PLASMA functions, POTRF, TRTRI, LAUUM, computing the inverse of a symmetric positive definite matrix using QUARK for superscalar parallelization.

Figures 5 and 6 further strengthen those points. Figures 5 shows the three DAGs of the three components of the matrix inversion and the aggregate DAG of the entire inversion. The DAG aggregation is a natural behavior of QUARK and happens automatically upon invocation of the three routines on Figure 4. The DAG created by stacking the DAGs of the three components on top of each other is taller and thinner, which means its sequential part (*the critical path*) is longer and its parallelism is more confined. The

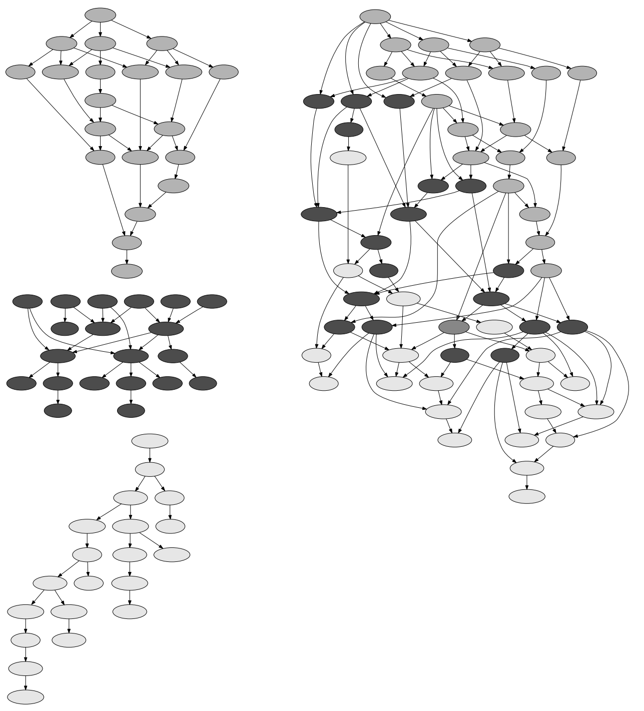


Figure 5. Three separate DAGs for the components of matrix inversion and their combined DAG.

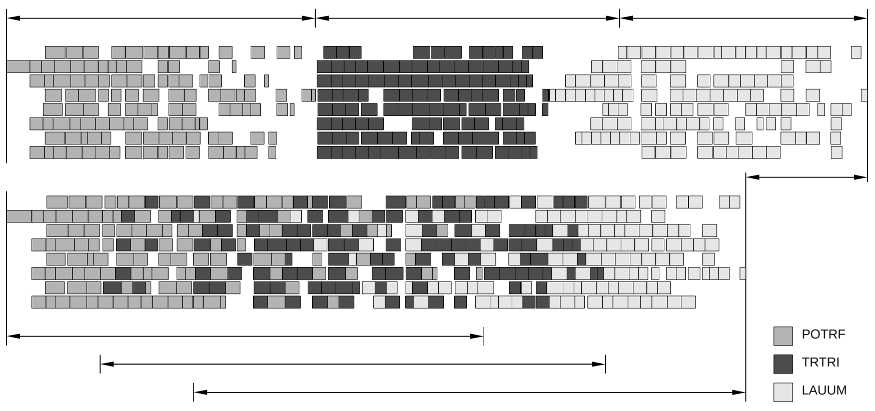


Figure 6. Trace of a small matrix inversion run on eight cores, with and without barriers between phases.

aggregate DAG is shorter and wider, meaning shorter critical path and more parallelism.

Figure 6 shows execution traces of the matrix inversion with and without barriers between phases. The parallelism of each separate phase is limited by its data dependencies, causing gaps in execution. However, when the barriers are removed, the three phases fill out each other's gaps, which results in a shorter overall execution time. Figure 7 shows

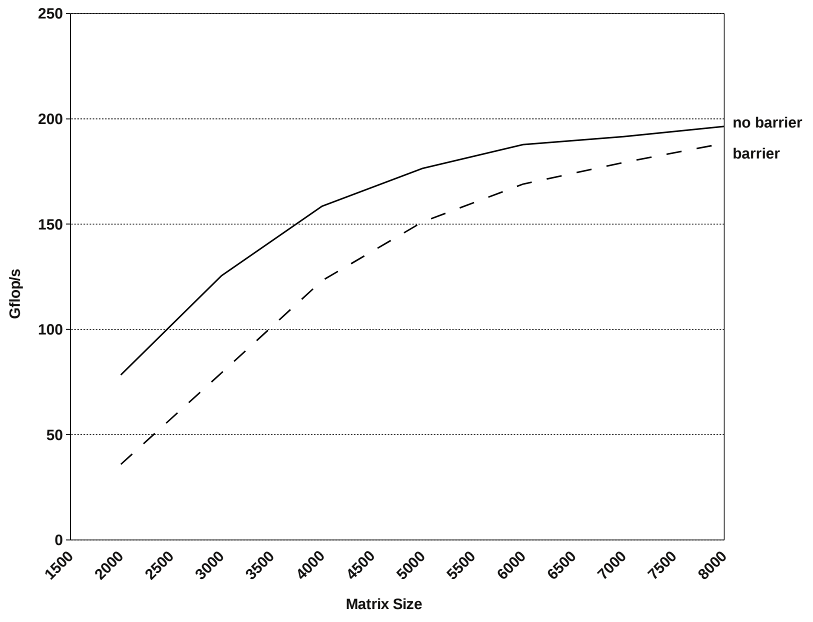


Figure 7. Performance difference between matrix inversion codes with and without barriers on a 48-core AMD-bases system.

the difference in performance, when using a large system with 48 cores.

2.4.3. Task Aggregation

Task aggregation is a feature of QUARK allowing for creation of large tasks representing agglomerates of smaller tasks. These tasks not only have a large number of "inherited" dependencies, but also the number of dependencies is not known at compile time, and determined at runtime instead. This feature has two main applications. One is to mitigate scheduling overheads of very fine granularity tasks, by increasing the level of granularity. Another is to offload work to data-parallel devices, such as GPUs. Here, the latter case will serve as an example.

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. The difficulty here comes from the fact that GPU cores cannot synchronize work in any other way than through a global barrier. In other words, the tasks offloaded to the GPU have to be independent.

The crucial concept here is the one of task aggregation. The GPU kernel is an aggregate of many CPU kernels, i.e., one call to the GPU kernel replaces many calls to CPU kernels. Because of the data-parallel nature of the GPU, the CPU calls constituting the aggregate GPU call cannot have dependencies among them. The GPU task call can be pictured in the DAG as a cluster of CPU tasks with arrows coming into

```
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_stsmqr, ...);
}
```

Figure 8. QUARK code for the QR factorization using CPU cores 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, ...);
    for (m = k+1; m < SIZE; m++)
        for (n = k+1; n < k+1+lookahead; n++)
            QUARK_Insert_Task(CORE_stsmqr, ...);
    task = QUARK_Task_Init(cuda_stsmqr, ...);
    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);
}
```

Figure 9. QUARK code for the QR factorization using CPUs and a GPU.

the cluster and out of the cluster, but no arrows connecting the task inside the cluster. In order to support this model, QUARK allows for queueing of tasks with a dynamic range of dependencies. Initially, a task with no dependencies is created through a call to `QUARK_Task_Init`. Then any number of dependencies can be added to the task using calls to `QUARK_Task_Pack_Arg`. Finally, the task can be queued with a call to `QUARK_Insert_Task_Packed`.

The tile QR factorization will serve as an example here. Figure 8 shows a QUARK implementation of the tile QR factorization using CPUs only. This particular example

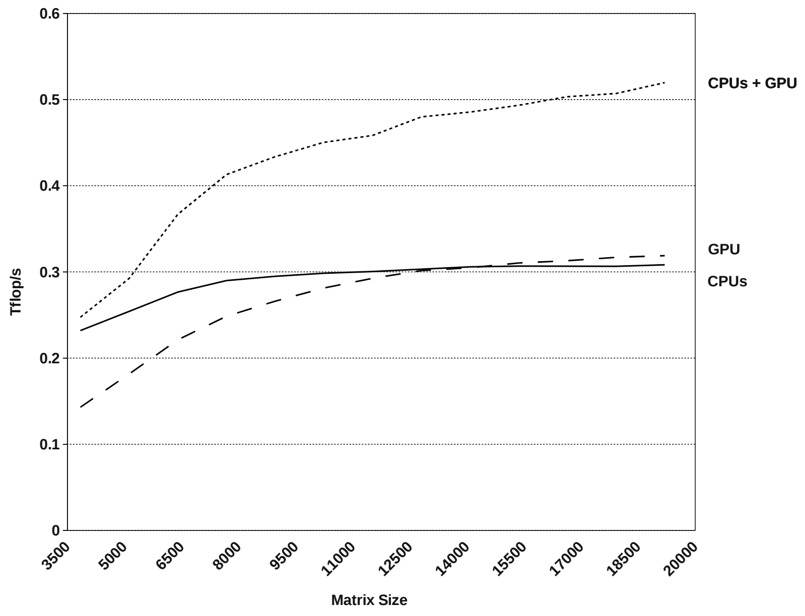


Figure 10. Performance of the QR factorization using 24 AMD-based cores and an NVIDIA Fermi GPU.

consists of five loops with three levels of nesting. It is built out of four `core_blas` kernels: GEQRT, TSQRT, ORMQR and TSMQR. More details can be found in PLASMA literature. Figure 9 shows modifications necessary to offload some of the tasks to the GPU. Here, the last loop nest is split into the CPUs part and the GPU part. The split is done along the *n* dimension. While the CPUs get *lookahead* columns of the matrix to process, the GPU gets *SIZE* − *lookahead* columns to process. The `cuda_stsmqr` kernel implements the GPU work, such that one tile of the matrix is (approximately) processed by one multiprocessor of the GPU. At the same time all dependencies corresponding to all the tile operations are created inside the double loop nest.

Although GPU acceleration in PLASMA is currently in a prototype stage, Figure 10 shows clearly that this approach allows to efficiently combine the power of a GPU and a big number of conventional CPU cores. In this particular case, the 14 cores (SMs) of the Fermi GPU combined with 24 conventional AMD cores were capable of delivering performance in excess of half a TeraFlop/s.

2.4.4. Nested Parallelism

A notable feature of QUARK is nested parallelism, which, among other uses, allowed for fine grained parallelization of the LU factorization [22] of a matrix panel using the recursive LU algorithm [25].

Figure 11 shows the initial factorization steps of a matrix subdivided into 9 tiles (a 3-by-3 grid of tiles). The first step is a recursive parallel factorization of the first panel consisting of three leftmost tiles. Only when this finishes, the other tasks may start executing, which creates an implicit synchronization point. To avoid the negative impact on parallelism, we execute this step on multiple cores to minimize the running time. How-

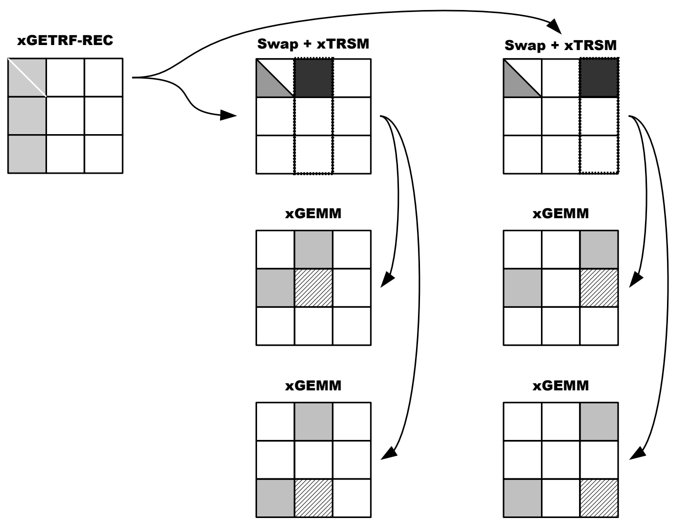


Figure 11. Execution breakdown for recursive tile LU factorization: factorization of the first panel using the parallel kernel is followed by the corresponding updates to the trailing submatrix.

ever, we use nested parallelism model, as most of the tasks are handled by a single core, and only the panel tasks are assigned to more than one core. Unlike similar implementations [17], we do not use all cores to handle the panel. There are two main reasons for this decision. First, we use dynamic scheduling, that enables us to hide the negative influence of the panel factorization behind more efficient work performed by concurrent tasks. And second, we have clearly observed the effect of diminishing returns when using too many cores for the panel. Consequently, we do not use them all, and, instead, we keep the remaining cores busy with other critical tasks.

The next step is pivoting to the right of the panel that has just been factorized. We combine in this step the triangular update (xTRSM in the BLAS parlance) because there is no advantage of scheduling them separately due to cache locality considerations. Just as the panel factorization locks the panel and has a potential to temporarily stall the computation, the pivot interchange has a similar effect. This is indicated by a rectangular outline encompassing the tile updated by xTRSM of the tiles below it. Even though so many tiles are locked by the triangular update, there is still a potential for parallelism because pivot swaps and the triangular update itself for a single column is independent of other columns. We can then easily split the operations along the tile boundaries and schedule them as independent tasks. This observation is depicted in Figure 11 by showing two xTRSM updates for two adjacent tiles in the topmost row of tiles instead of one update for both tiles at once.

The last step, shown in Figure 11, is an update based on the Schur complement. It is the most computationally intensive operation in the LU factorization and is commonly implemented with a call to a Level 3 BLAS kernel called xGEMM. Instead of a single call that performs the whole update of the trailing submatrix, we use multiple invocations of the routine because we use a tile-based algorithm. In addition to exposing more parallelism and the ability to alleviate the influence of algorithm's synchronization points

```
function xGETRFR(M, N, column) {
  if N == 1 {                                   single column, recursion stops
    idx = split_IxAMAX(...)                     compute local maximum of modulus
    gidx = combine_IxAMAX(idx)                  combine local results
    split_xSCAL(...)                            scale local data
  } else {
    xGETRFR(M, N/2, column)                     recursive call to factor left half
    xLASWP(...)                                 pivoting forward
    split_xTRSM(...)                            triangular solve
    split_xGEMM(...)                            Schur's complement
    xGETRFR(M, N-N/2, column+N/2)               recursive call to factor right half
    xLASWP(...)                                 pivoting backward
  }
}
```

Figure 12. Pseudo-code for the recursive panel factorization.

(such as the panel factorization), by splitting the Schur update operation we are able to obtain better performance than a single call to a parallelized vendor library [3].

One thing not shown in Figure 11 is pivoting to-the-left, because it does not occur in the beginning of the factorization. It is only necessary for the second and subsequent panels. The swaps originating from different panels have to be ordered correctly but are independent for each column, which is the basis for running them in parallel. The only inhibitor of parallelism then is the fact that the swapping operations are inherently memory-bound because they do not involve any computation. On the other hand, the memory accesses are done with a single level of indirection, which makes them very irregular in practice. Producing such memory traffic from a single core might not take advantage of the main memory's ability to handle multiple outstanding data requests and the parallelism afforded by NUMA hardware. It is also noteworthy to mention that the tasks performing the pivoting behind the panels are not located on the critical path and therefore, are not essential for the remaining computational steps in the sense that they could be potentially delayed toward the end of the factorization.

3. MAGMA

We aim to extend some of the state-of-the-art algorithms and techniques for high performance linear algebra on GPU architectures. The main goal of the development is to map the algorithmic requirements to the architectural strengths of the GPUs. For example, GPUs are many-core architectures and hence require the dense linear algorithms to exhibit high levels of parallelism. New data structures must be designed to facilitate parallel and coalesced memory accesses. The introduction of shared memory and multiple levels of memory hierarchy in the NVIDIA Fermi GPUs [39] has enabled the benefits of memory reuse. This, in turn, made blocking techniques applicable for achieving a more efficient use of the memory hierarchy. Furthermore, we have developed a *hybridization* technique, that is motivated by the fact, that today's GPUs are used only through explicit requests from the CPU. We thus employ CPU scheduling methods to queue the computational tasks and requests, that are meant to be executed on the GPU.

The scope of our work comprises basic dense linear algebra kernels as well as higher-level routines such as linear, least-squares, eigenvalue, and singular-value solvers. For dense linear algebra operations, we highlight some Level 2 and 3 BLAS and their use in basic factorizations and solvers.

We also show that basic linear algebra kernels achieve a high fraction of the theoretical performance peak on GPUs. Implementations of most of the dense linear algebra codes described here are available through the MAGMA library [46] (Matrix Algebra on GPU and Multicore Architectures).

3.1. Acceleration Techniques for GPUs

3.1.1. Algorithmic Blocking

Algorithmic blocking is a well known linear algebra optimization technique, where the computation is organized to operate on blocks: carefully chosen submatrices of the original matrix. The main idea is, that the blocks are small enough to fit into a particular memory hierarchy level so that once loaded there, the blocks' data may be continuously reused in all of the arithmetic operations required for that data. This technique may be applied on GPUs by using GPUs' shared memory. As it is demonstrated throughout the paper, the application of blocking is crucial for the performance of numerous kernels and high level algorithms for both sparse and dense linear algebra.

3.1.2. Algorithmic Hybridization

Algorithmic Hybridization refers to a methodology, that we follow in developing high level linear algebra algorithms:

- Represent the algorithm as a collection of basic kernels/tasks together with the dependencies among them as shown in Figure 13):
  - ∗ Use parametrized task granularity to facilitate auto-tuning;
  - ∗ Use performance models to facilitate the task splitting and mapping.
- Schedule the execution of the basic kernels and tasks over the multicore processor and the GPU:
  - ∗ Schedule small, non-parallelizable tasks on the CPU and large, parallelizable ones on the GPU;
  - ∗ Define the algorithm's *critical path* and prioritize its scheduling and execution.

3.1.3. Data Structures

A great deal of research in GPU computing focuses on appropriate data structures. Data should be organized in a way that facilitates parallel memory accesses (see also Section 3.1.4 below), so that applications may achieve high memory throughput, made possible by the use of GPUs. This is especially true for the sparse matrix-vector multiplication kernel, where researchers have proposed numerous data formats. Even in dense linear algebra, where data structures for input/output interfaces are more standardized, there are numerous algorithms, where intermediate steps involve rearranging data to facilitate fast memory accesses (see Section 3.2).

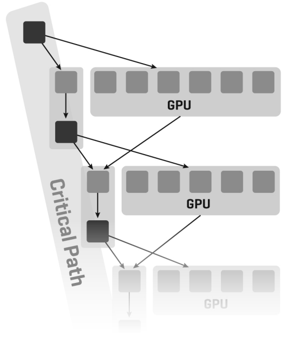


Figure 13. Algorithms as a collection of basic kernels, tasks, and dependencies among them (DAGs) for hybrid GPU-based computing.

3.1.4. Parallel Memory Access

Accesses to GPU's global memory are costly and, in general, not cached except on Fermi cards which are the latest generation of GPUs from NVIDIA [39]. For performance, it is crucial to have the right access pattern to get maximum memory bandwidth. There are two access requirements for high throughput [41,40]. The first is to organize global memory accesses in terms of parallel consecutive memory accesses – 16 consecutive elements at a time by the threads of a half-warp (16 threads). By doing so, memory accesses (up to 16 elements at a time) will be *coalesced* into a single memory access. This is demonstrated in the kernels' design throughout this section. Second, the data should be properly aligned. In particular, the data to be accessed by half-warp should be aligned at $16 * \texttt{sizeof}(\texttt{element})$, e.g., 64 bytes for single precision elements.

3.1.5. Pointer Redirecting

Pointer redirecting is a set of GPU specific optimization techniques, that allows for easy removal of performance oscillations in cases, where the input data is not aligned to directly allow coalescent memory accesses, or when the problem sizes are not divisible by the partitioning sizes required for achieving high performance [37]. For example, when applied to the dense matrix-matrix multiplication routine this can lead to two-fold increase in speed depending on the hardware configuration and routine parameters. Similarly, the dense matrix-vector multiplication can be accelerated more than two-fold in both single and double precision arithmetic.

3.1.6. Storage Data Padding

Storage Data Padding is similar to pointer redirecting and is another technique for obtaining high overall performance as well as for remove performance variability. This method is used when input data is not aligned to allow coalescent memory accesses, or when the problem sizes are not divisible by certain blocking sizes. The difference is that, with padding, the problem dimensions are increased and the extra space is filled with zeroes which we refer to as padding. A drawback of the padding is that it requires extra memory, and it may involve extra data copies. A comparison of the padding and pointer redirecting approaches for dense matrix-matrix multiplication shows, that for small matrix sizes the pointer redirecting gives better performance, while for larger matrices, the two approaches are almost identical. An advantage of using padding is the possibility of users to lay out the input data, so that the highest performance may be achieved without the library code performing any additional padding.

3.1.7. Auto-tuning

Automatic performance tuning for optimal performance is called auto-tuning. It is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS [49] and PHiPAC [8] were used to generate highly optimized BLAS.

Auto-tuning can also be applied to tuning linear algebra software on GPUs. Recent work in the area of dense linear algebra [32] shows that auto-tuning for GPUs is a very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures, and to substantially speed up even highly hand-tuned kernels.

There are two core components in a comprehensive auto-tuning system:

- **Code generator**: The code generator produces code variants according to a set of pre-defined, parametrized algorithms' templates. The code generator also applies certain state of the art optimization techniques.
- **Heuristic search engine**: The heuristic search engine runs the variants produced by the code generator and finds out the best ones using a feedback loop, e.g., the performance results of previously evaluated variants are used as a guidance for the search on currently unevaluated variants.

Auto-tuning is used to determine best performing kernels, partitioning sizes, and other parameters for the various algorithms described here.

3.2. Accelerating Dense Linear Algebra Kernels and Factorizations

Implementations of the BLAS interface are a major building block of dense linear algebra (DLA) libraries, and therefore must be highly optimized. This is true for GPU computing as well, especially after the introduction of shared memory in modern GPUs. This is important because, it enabled fast Level 3 BLAS implementations, which in turn made possible the development of DLA for GPUs to be based on GPU BLAS.

Despite the current success in developing highly optimized BLAS for GPUs [20, 48], the area is still new, and presents numerous opportunities for improvements. For example, we address cases of the matrix-matrix and the matrix-vector multiplications, along with discussion about the techniques used to achieve these improvements.

Figure 14 shows the performance for GEMM in double precision arithmetic on NVIDIA GTX 280. Note the performance oscillations that CUBLAS experiences for problem sizes not divisible by 32. This performance degradation can be removed using the pointer redirecting technique [37], as the figure clearly illustrates. We see an improvement of 24 Gflops/s in double and 170 GFlops/s in single precision arithmetic. We extended this technique to other Level 3 BLAS kernels to see similar performance improvements. Figure 15 shows the performance of both single and double precision GEMM

Feature	Tesla GTX 280	Fermi C2050
Frequency	602 MHz	1150 MHz
CUDA cores (vertex shaders)	240	448
Streaming Multiprocessors (SM)	30	14
Shared memory per SM	16 KiB	16 KiB or 48 KiB
L1 cache per SM	–	16 KiB or 48 KiB
L2 cache per SM	–	768 KiB
Theoretical peak in single precision	933 Gflop/s	1030 Gflop/s
Theoretical peak in double precision	80 Gflop/s	515 Gflop/s
Address width	32 bits	64 bits

Table 1. Detailed description of NVIDIA Tesla GTX 280 and the server version of NVIDIA Fermi GPU: C2050.

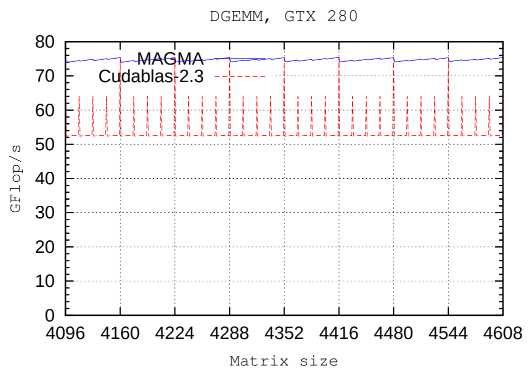


Figure 14. Performance of MAGMA's implementation of matrix-matrix multiplication routine DGEMM on NVIDIA GTX 280.

routines on NVIDIA Fermi C2050. Comparison of specification of both platforms is summarized in Table 1.

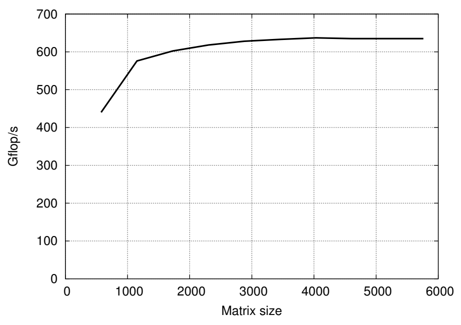
Padding can be applied as well, but in many cases copying user data may not be feasible. This is the case for algorithms that involve Level 2 BLAS that has to be applied for matrices of continuously decreasing sizes, e.g., in the bidiagonal reduction for the symmetric eigenvalue problem. This motivated us to extend the technique to Level 2 BLAS to obtain similar improvements.

Conceptually, above the level of basic computational kernels provided by BLAS, are the one-sided factorization codes. Most commonly used ones are Cholesky, LU, and QR factorizations. When the aforementioned hybridization techniques are combined with good quality BLAS, then, asymptotically, a high fraction of the peak performance may indeed be achieved as shown in Figure 16.

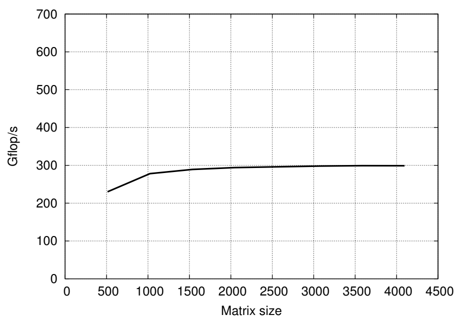
4. Programming Linear Algebra with DPLASMA and DAGuE

4.1. Representation of Linear Algebra Kernels in DAGUE

In this section, we present in details how some Linear Algebra operations have been programed with the DAGUE framework in the context of the DPLASMA library. We

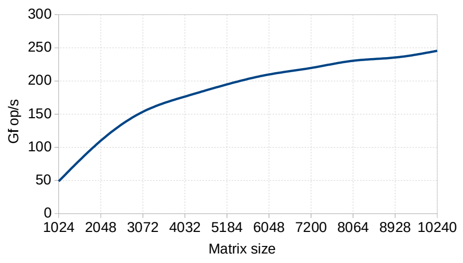


(a) SGEMM

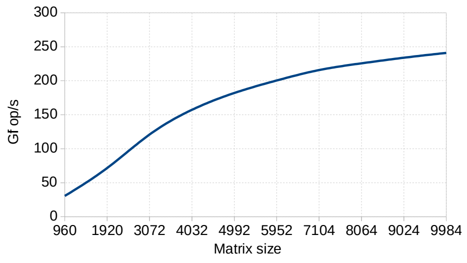


(b) DGEMM

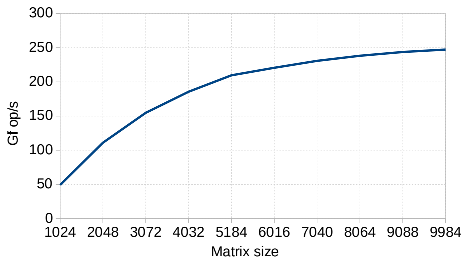
Figure 15. Performance of MAGMA's implementation of matrix-matrix multiplication routine DGEMM on NVIDIA Fermi C2050.



(a) Cholesky Factorization



(b) LU Factorization



(c) QR Factorization

Figure 16. Performance of MAGMA's one sided factorizations on Fermi C2050 in double precision.

use one of the most common one-sided factorizations as a walkthrough example, QR. We first present the algorithm, and its properties, then, we walk through all the steps a programmer must perform to get a fully functional QR factorization. We present how this operation is integrated in a parallel MPI application, how some kernels are ported to enable acceleration using GPUs, and some tools provided by the DAGuE framework to evaluate the performance and tune the resulting operation.

4.1.1. Tile Linear Algebra: PLASMA, DPLASMA

The PLASMA project has been designed to target shared memory multicore machines. Although the idea of tiles algorithm does not specifically resonates with the typical specificities of a distributed memory machine (where cache locality and reuse are of little significance when compared to communication volume), a typical supercomputer tends to be structured as a cluster of commodity nodes, which means many cores and sometimes accelerators. Hence, a tile based algorithm can execute more efficiently on each node, often translating into a general improvement for the whole system. The core idea of the DPLASMA project is to reuse the tile algorithms developed for PLASMA, but using the DAGuE framework to express them as parametrized DAGs that can be scheduled on large scale distributed systems of such form.

4.1.2. Tile QR algorithm

The QR factorization (or QR decomposition) offers a numerically stable way of solving full rank underdetermined, overdetermined, and regular square linear systems of equations. The QR factorization of an $m \times n$ real matrix A has the form $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix.

A detailed tile QR algorithm description can be found in [16]. Figure 8 shows the pseudocode of the Tile QR factorization. It relies on four basic operations implemented by four computational kernels for which reference implementations are freely available as part of either the BLAS, LAPACK or PLASMA [2].

- DGEQRT: The kernel performs the QR factorization of a diagonal tile and produces an upper triangular matrix R and a unit lower triangular matrix V containing the Householder reflectors. The kernel also produces the upper triangular matrix T as defined by the compact WY technique for accumulating Householder reflectors [43]. The R factor overrides the upper triangular portion of the input and the reflectors override the lower triangular portion of the input. The T matrix is stored separately.
- DTSQRT: The kernel performs the QR factorization of a matrix built by coupling the R factor, produced by DGEQRT or a previous call to DTSQRT, with a tile below the diagonal tile. The kernel produces an updated R factor, a square matrix V containing the Householder reflectors and the matrix T resulting from accumulating the reflectors V. The new R factor overrides the old R factor. The block of reflectors overrides the corresponding tile of the input matrix. The T matrix is stored separately.
- DORMQR: The kernel applies the reflectors calculated by DGEQRT to a tile to the right of the diagonal tile, using the reflectors V along with the matrix T.
- DSSMQR: The kernel applies the reflectors calculated by DTSQRT to the tile two tiles to the right of the tiles factorized by DTSQRT, using the reflectors V and the matrix T produced by DTSQRT.

4.2. Walkthrough QR Implementation

The first step to write the QR algorithm of DPLASMA is to take the sequential code presented in Figure 8, and process it through the DAGuE compiler (as described in section 4.4). This produces a JDF file, that then needs to be completed by the programmer.

```
/* Prologue, dumped "as is" in the generated file */
extern "C" %{
  /**
   * TILE QR FACTORIZATION
   * @precisions normal z -> s d c
   */
#include <plasma.h>
#include <core_blas.h>

#include "dague.h"
/* ... */ /* more includes */
#include "dplasma/cores/cuda_stsmqr.h"
%}

/* Input variables used when creating the
 * algorithm object instance */
descA  [type = "tiled_matrix_desc_t"]
A      [type = "dague_ddesc_t *"]
descT  [type = "tiled_matrix_desc_t"]
T      [type = "dague_ddesc_t *" aligned=A]
ib     [type = "int"]
p_work [type = "dague_memory_pool_t *"
        size = "(sizeof(PLASMA_Complex64_t)*ib*(descT.nb))"]
p_tau  [type = "dague_memory_pool_t *"
        size = "(sizeof(PLASMA_Complex64_t)   *(descT.nb))"]

/* Tasks descriptions follow */
```

Figure 17. Samples from the JDF of the QR algorithm: prologue

```
/* Prologue precedes, other tasks */

ztsmqr(k,m,n)
  /* ... */ /* Execution space (autogenerated) */

  /* Variable names translation table (autogenerated) */
  /* J == A(k,n) */
  /* ... */ /* more translations */

  /* dependencies (autogenerated)
  RW  J <- (m==k+1) ? E zunmqr(m-1,n) : J ztsmqr(k,m-1,n)
        -> (m==descA.mt-1) ? J ztsmqr_out_A(k,n) : J ztsmqr(k,m+1,n)
  /* ... */ /* more dependencies */

  /* Task affinity with data (edited by programmer) */
  : A(m, n)

BODY  /* edited by programmer */
  /* computing tight tile dimensions
   * (tiles on matrix edges contain padding) */
  int tempnn = (n==descA.nt-1) ? descA.n-n*descA.nb : descA.nb;
  int tempmm = (m==descA.mt-1) ? descA.m-m*descA.mb : descA.mb;
  int ldak = BLKLDD( descA, k );
  int ldam = BLKLDD( descA, m );

  /* Obtain a scratchpad allocation */
  void* p_elem_A = dague_private_memory_pop( p_work );
  /* Call to the actual kernel */
  CODELET_ztsmqr(PlasmaLeft, PlasmaConjTrans, descA.mb,
                 tempnn, tempmm, tempnn, descA.nb, ib,
                 J /* A(k,n) */, ldak,
                 K /* A(m,n) */, ldam,
                 L /* A(m,k) */,  ldam,
                 M /* T(m,k) */,  descT.mb,
                 p_elem_A, ldwork );
  /* Release the scratchpad allocation */
  dague_private_memory_push( p_work, p_elem_A );
END
```

Figure 18. Samples from the JDF of the QR algorithm: task body

The first part of the JDF file contains a user defined prologue (presented in Figure 17). This prologue is copied directly in the generated C code produced by the JDF compiler, so the programmer can add suitable definitions and includes necessary for the body of tasks. An interesting feature is automatic generation of a variety of numerical precisions from a single source file, thanks to a small helper translator that does source-to-source pattern matching to adapt numerical operations to the target precision. The next section of the JDF file declares the inputs of the algorithm and their types. From these declarations, the JDF compiler creates automatically all the interface functions used by the main program (or the library interface) to create, manipulate and dispose of the DAGuE object representing a particular instance of the algorithm.

Then, the JDF file contains the description of all the task classes, usually generated automatically from the decorated sequential code. For each task class, the programmer needs to define 1) the data affinity of the tasks (`: A(m, n)` in Figure 24) and 2) user provided bodies, which are, in the case of linear algebra, usually as simple as calling a BLAS or PLASMA kernel. Sometimes, algorithmic technicalities result in additional work for the programmer: many kernels of the QR algorithm use a temporary scratchpad memory. This memory is purely local to the kernel itself, hence does not need to appear in the dataflow. However, to preserve Fortran compatibility, scratchpad memory needs to be allocated outside the kernels themselves, and passed as an argument. As a consequence, the bodies have to allocate and release these temporary arrays. We have designed a set of helper functions while designing DPLASMA, whose purpose is to ease the writing of linear algebra bodies; code presented in Figure 18 illustrates how the programmer can push and pop scratchpad memory from a generic system call free memory pool. The variables name translation table, dumped automatically by the sequential code dependency extractor, helps the programmer navigate the generated dependencies and select the appropriate variable as a parameter of the actual computing kernel.

4.2.1. Accelerator Port

The only action required from the linear algebra package to enable GPU acceleration is to provide the appropriate codelets in the body part of the JDF file. A codelet is a piece of code that encapsulates a variety of implementations of an operation for a variety of hardware. Just like CPU core kernels, GPU kernels are sequential and pure, hence, a codelet is an abstraction of a computing function suitable for a variety of processing units, either a single core or a single GPU stream (even though they can still contain some internal parallelism, such as vector SIMD instructions). Practically, that means that the application developer is in charge of providing multiple versions of the computing bodies. The relevant codelets, optimized for the current hardware, are loaded automatically during the algorithm initialization (one for the GPU hardware, one for the CPU cores, etc). Today, the DAGuE runtime supports only CUDA and CPU codelets, but the infrastructure can easily accommodate other accelerator types (MIC, OpenCL, FPGAs, Cell, . . .). If a task features multiple codelets, the runtime scheduler chooses dynamically (during the invocation of the automatically generated scheduling hook CODELET_kernelname) between all these versions, in order to execute the operation on the most relevant hardware. Because multiple versions of the same codelet kernel can be in use at the same time, the workload of this type of operations, on different input data, can be distributed on both CPU cores and GPUs simultaneously.

```
1   dague_object_t* dplasma_sgeqrf_New( tiled_matrix_desc_t *A,
2                                       tiled_matrix_desc_t *T )
3   {
4     dague_sgeqrf_object_t* d = dague_sgeqrf_new(*A, (dague_ddesc_t*)A,
5                                                 *T, (dague_ddesc_t*)T,
6                                                 ib, NULL, NULL);
7
8     d->p_tau = malloc(sizeof(dague_memory_pool_t));
9     dague_private_memory_init(d->p_tau, T->nb * sizeof(float));
10    /* ... */ /* similar code for p_work scratchpad */
11
12    /* Datatypes declarations, from MPI datatypes */
13    dplasma_add2arena_tile(d->arenas[DAGUE_sgeqrf_DEFAULT_ARENA],
14                           A->mb*A->nb*sizeof(float),
15                           DAGUE_ARENA_ALIGNMENT_SSE,
16                           MPI_FLOAT, A->mb);
17    /* Lower triangular part of tile without diagonal */
18    dplasma_add2arena_lower(d->arenas[DAGUE_sgeqrf_LOWER_TILE_ARENA],
19                            A->mb*A->nb*sizeof(float),
20                            DAGUE_ARENA_ALIGNMENT_SSE,
21                            MPI_FLOAT, A->mb, 0);
22    /* ... */ /* similarly, U upper triangle and T (IB*MB rectangle)*/
23
24    return (dague_object_t*)d;
25  }
```

Figure 19. User provided wrapper around the DAGuE generated QR factorization function

In the case of the QR factorization, we selected to add a GPU version of the STSMQR kernel, which is the matrix-matrix multiplication kernel used to update the remainder of the matrix, after a particular panel has been factorized (hence representing 80% or more of the overall compute time). We have extended a handmade GPU kernel [15], originally obtained from MAGMA [2]. This kernel is provided in a separate source file, and is developed separately as a regular CUDA function. Should future versions of CuBLAS enable running concurrent GPU kernels on several hardware streams, these vendor functions could be used directly.

4.2.2. Wrapper

As previously stated, scratchpad memory needs to be allocated outside of the bodies. Similarly, because we wanted the JDF format to be oblivious of the transport technology, datatypes, which are inherently dependent on the description used in the message passing system, need to be declared outside the generated code. In order for the generated library to be more convenient to use for end-users, we consider it good practice to provide a wrapper around the generated code that takes care of allocating and defining these required elements. In the case of linear algebra, we provide a variety of helper functions to allocate scratchpads (line 9 in Figure 19), and to create datatypes (lines 13, 18 in the Figure), like band matrices, square or rectangular matrices, etc. Again, the framework provided tools can create all floating point precisions from a single source.

4.2.3. Main Program

A skeleton program that initializes and schedules a QR factorization using the DAGuE framework is presented in Figure 20. Since DAGuE uses MPI as an underlaying communication mechanism, the test program is an MPI program. It thus needs to initialize and finalize MPI (lines 7 and 35) and the programmer is free to use any MPI functionality between DAGuE calls. A subset of the DAGuE calls are to be con-

```
 1  int main(int argc, char **argv) {
 2      dague_context_t *dague;
 3      two_dim_block_cyclic_t ddescA;
 4      two_dim_block_cyclic_t ddescT;
 5      dague_object_t* zgeqrf_object;
 6
 7      MPI_Init(&argc, &argv);
 8      /* ... */
 9
10      dague = dague_init(NBCORES, &argc, &argv);
11      dague_set_scheduler(dague, &dague_sched_LHQ);
12
13          /* Matrix allocation and random filling */
14      two_dim_block_cyclic_init(&ddescA, matrix_ComplexDouble, /* ... */);
15      ddescA.mat = dague_data_allocate(/* ... */);
16      dplasma_zplrnt(dague, &ddescA, 3872);
17      dplasma_zlaset(dague, PlasmaUpperLower, 0., 0., &ddescT);
18      /* ... */ /* Same for other matrices */
19
20      zgeqrf_object = dplasma_zgeqrf_New(&ddescA, &ddescT);
21      dague_enqueue(dague, zgeqrf_object);
22
23          /* Computation happens here */
24      dague_progress(dague);
25
26      dplasma_zgeqrf_Destruct(zgeqrf_object);
27
28      /* ... */
29
30      dague_data_free(ddescA.mat);
31      dague_ddesc_destroy((dague_ddesc_t*)&ddescA);
32      /* ... */
33
34      dague_fini(&dague);
35      MPI_Finalize();
36
37      return 0;
38  }
```

Figure 20. Skeleton of a DAGuE main program driving the QR factorization

sidered as collective operations from an MPI perspective: all MPI processes must call them in the same order, with a communication scheme that allows these operations to match. These operations are the initialization function (dague_init), the progress function (dague_progress) and the finalization function (dague_fini). dague_init will create a specified number of threads on the local process, plus the communication thread. Threads are bound on separate cores when possible. Once the DAGuE system is initialized on all MPI processes, each must choose a local scheduler. DAGuE provides four scheduling heuristics, but the one preferred is the Local Hierarchical Scheduler, developed specifically for DAGuE on NUMA many-core heterogeneous machines. The function dague_set_scheduler of line 11 sets this scheduler.

The next step consists of creating a data distribution descriptor. This code holds two data distribution descriptors: ddescA and ddescT. DAGuE provides three built-in data distributions for tiled matrices: an arbitrary index based distribution; a symmetric two dimensional block cyclic distribution, and a two dimensional block cyclic distribution. In the case of QR, the latter is used to describe the input matrix A to be factorized, and the workspace array T. Once the data distribution is created, the local memory to store this data should be allocated in the fields mat of the descriptor. To enable DAGuE to pin memory, and allow for direct DMA transfers (to and from the GPUs or some high performance networks), the helper function dague_data_allocate of line 14 is used. The workspace array T should be described and allocated in a similar way on line 18.

```
 1  int dplasma_sgeqrf( dague_context_t *dague, tiled_matrix_desc_t *A,
 2                                             tiled_matrix_desc_t *T )
 3  {
 4      dague_object_t *dague_sgeqrf = dplasma_sgeqrf_New(A, T);
 5
 6      dague_enqueue(dague, dague_sgeqrf);
 7      dplasma_progress(dague);
 8
 9      dplasma_sgeqrf_Destruct(dague_sgeqrf);
10      return 0;
11  }
```

Figure 21. DPLASMA SPMD interface for the DAGuE generated QR factorization function

Then, this test program uses DPLASMA functions to initialize the matrix A with random values (line 16), and the workspace array T with 0 (line 19). These functions are coded in DAGuE: they create a DAG representation of a map operation that will initialize each tile in parallel with the desired values, making the engine progress on these DAGs.

Once the data is initialized, a `zgeqrf` DAGuE object is created with the wrapper that was described above. This object holds the symbolic representation of the local DAG, initialized with the desired parameters, and bound to the allocated and described data. It is (locally) enqueued in the DAGuE engine on line 21.

To compute the QR operation described by this object, all MPI processes call to `dague_progress` on line 24. This enables all threads created on line 10 to work on the QR operation enqueued before in collaboration with all the other MPI processes. This call returns when all enqueued objects are completed, thus when the factorization is done. At this point, the `zgeqrd` DAGuE object is consumed, and can be freed by the programmer at line 26. The result of the factorization should be used on line 28, before the data is freed (line 30), and the descriptors destroyed (line 31). Line 32 should hold similar code to free the data and destroy the descriptor of T. Then, the DAGuE engine can release all resources (line 34) before MPI is finalized and the application terminates.

4.2.4. SPMD library interface

It is possible for the library to encapsulate all dataflow related calls inside a regular (ScaLAPACK like) interface function. This function creates an algorithm instance, enqueues it in the dataflow runtime and enables progress (lines 4, 6, 7 in Figure 21). From the main program point of view, the code is similar to a SPMD call to a parallel BLAS function; the main program does not need to consider the fact that dataflow is used within the linear algebra library. While this approach can simplify the porting of legacy applications, it prevents the program from composing DAG based algorithms. If the main program takes full control of the algorithm objects, it can enqueue multiple algorithms, and then progress all of them simultaneously, enabling optimal overlap between separate algorithms (such as a factorization and the associated solve); if it simply calls the SPMD interface, it still benefits from complete parallelism within individual functions, but it falls back to a synchronous SPMD model between different algorithms.

4.3. Correctness and Performance Analysis Tools

The first correctness tool of the DAGuE framework sits within the code generator tool, which converts the JDF representation into C functions. A number of conditions on the

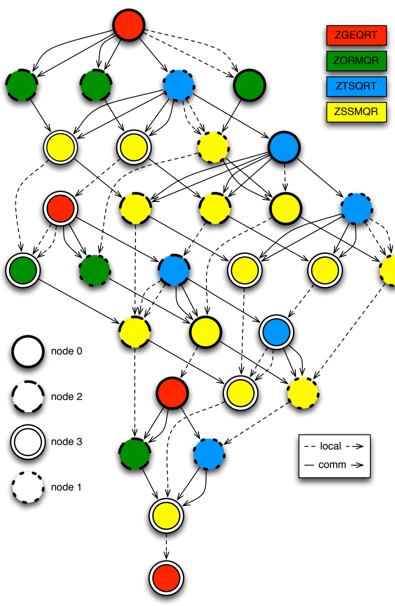
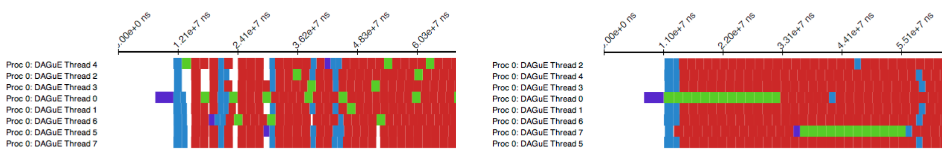


Figure 22. The runtime can output a graphical version of the DAG for the programmer to verify correctness, in this example, the output is from the execution of the QR operation on a 4×4 tiled matrix, on four nodes with eight cores per node.

dependencies and execution spaces are checked during this stage, and can detect many instances of mismatching dependencies (where the input of task A comes from task B, but task B has no outputs to task A). Similarly, conditions that are not satisfiable according to the execution space raise warnings, as is the case for pure input data (operations that read the input matrix directly, not as an output of another task) that do not respect the task-data affinity. There warnings help the programmer detect the most common errors when writing the JDF.

At runtime, algorithm programmers can generate the complete unrolled DAG, for offline analysis purposes. The DAGUE engine can output a representation of the DAG, as it is executed, in the `dot` input format of the GraphViz graph plotting tool. The programmer can use the resulting graphic representation (see Figure 22) to analyze which kernel ran on what resource, and which dependence released which tasks into their ready state. Using such information has proven critical when debugging the JDF representation (for an advanced user who wants to write her own JDF directly without using the DAGUE compiler), or to understand contentions and improve the data distribution and the priorities assigned to tasks.

The DAGUE framework also features performance analysis tools to help programmers fine-tune the performance of their application. At the heart of these tools, the profiling collection mechanism optionally records the duration of each individual task, communication, and scheduling decision. These measurements are saved in thread-specific memory, without any locking or other forms of atomic operations, and are then output at termination time in an XML file for offline analysis.



(a) FIFO Scheduler (b) Local Hierarchical Queue Scheduler

Figure 23. Gantt representation of a shared memory run of the QR factorization on 8 threads.

This XML file can then be converted by tools provided in the framework to portable trace formats (like OTF [35]), or simple spreadsheets, representing the start date and duration of each critical operation. Figure 23 presents two Gantt chart representations of the beginning of a QR DAGuE execution on a single node, 8 cores using two different scheduling heuristics: the simple FIFO scheduling and the scheduler of DAGuE (Local Hierarchical Queues). The efficiency of the Local Hierarchical Queues scheduler to increase the data locality, allow for maximal parallelism, and avoid starvations highlighted in these graphs. Potential starvations are easily spotted, as they appear as large stripes where multiple threads do not execute any kernel. Similar charts can be generated for distributed runs (not presented here), with a clear depiction of the underlying communications in the MPI thread, annotated by the data they carry and tasks they connect. Using these results, a programmer can assess the efficiency, on real runs, of the proposed data distribution, task affinity, and priority. Data distribution and task affinity will both influence the amount and duration of communications, as well as the amount of starvation, while Priority will mostly influence the amount of starvation.

In the case of the QR factorization, these profiling outputs have been used to evaluate the priority hints given to tasks, used by the scheduler when ordering tasks.The folklore knowledge about scheduling DAG of dense factorizations is that the priorities should always favor the tasks that are closer to the critical path. We have implemented such a strategy, and discovered that it is easily outperformed by a completely dynamic scheduling that does not respect any priorities. There is indeed a fine balance between following the absolute priorities along the critical path, which enables maximum parallelism, and favoring cache reuse even if it progresses a branch that is far from the critical path. We have found a set of beneficial priority rules (which are symbolic expressions similar to the dependencies) which favor progressing iterations along the ”k” direction first, but favoring only a couple iterations of the critical path over update kernels.

4.4. Dataflow Representation

The depiction of the data dependencies, of the task execution space, as well as the flow of data from one task to another is realized in DAGuE through an intermediary level language named Job Data Flow (JDF). This is the representation that is at the heart of the symbolic representation of folded DAGs, allowing DAGuE to limit its memory consumption while retaining the capability of quickly finding the successors and predecessors of any given task. Figure 24 shows a snippet from the JDF of the linear algebra one-sided factorization QR. More details about the QR factorization and how it is fully integrated into DAGuE is given in Section 4.1.

```
unmqr(k,n)
  k = 0..inline_c %{ return MIN((A.nt-2),(A.mt-1)); %}
  n = (k+1)..(A.nt-1)

  : A.mat(k,n)

  READ  E <- C geqrt(k)    [type = LOWER_TILE]
  READ  F <- D geqrt(k)    [type = LITTLE_T]
  RW    G <- (k==0) ? B DAGUE_IN_A(0, n) : M tsmqr(k-1, k, n)
          -> (k<=A.mt-2) ? L tsmqr(k, k+1, n) : P DAGUE_OUT_A(k, n)

BODY
  ...
END
```

Figure 24. Sample Job Data Flow (JDF) representation

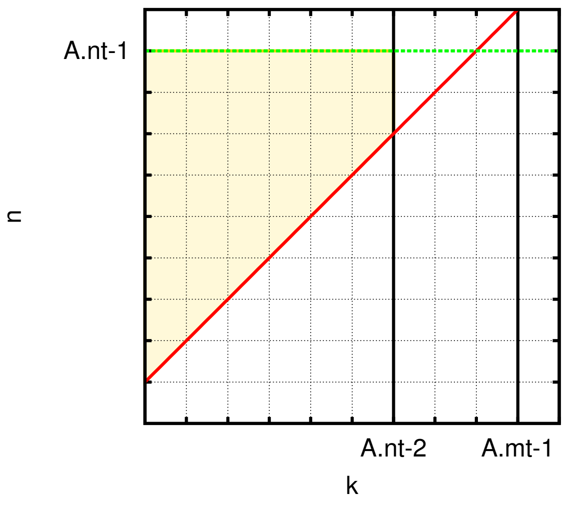


Figure 25. 2D Execution space of UNMQR(k,n)

Figure 24 shows the part of the JDF that corresponds to the task class unmqr(k,n). We use the term "*task class*" to refer to a parameterized representation of a collection of tasks that all perform the same operation, but on different data. Any two tasks contained in a task class are differing in their values of the parameters. In the case of unmqr(k,n), the two variables "k" and "n" are the parameters of this task class and along with the ranges provided in the following two lines, define the 2-D polygon that constitutes the execution space of this task class. A graphic representation of this polygon is provided by the shaded area in Figure 25.[1] Each lattice point included in this polygon (i.e., each point with integer coordinates) corresponds to a unique task of this task class. As is implied by the term "`inline_c`" in the first range, the ranges of values that the parameters can take do not have to be bound by constants, but can be the return value of arbitrary C code that will be executed at runtime.

Below the definition of the execution space, the line:

[1]For this depiction A.nt-2 was arbitrarily chosen to be smaller than A.mt-1, but in the general case they can have any relation between them.

```
                    : A.mat(k,n)
```

defines the affinity of each task to a particular block of the data. The meaning of this notation is that the runtime must schedule task unmqr(k_i, n_i) on the node where the matrix tile A[k_i][n_i] is located, for any given values k_i and n_i. Following the affinity, there are the definitions of the dependence edges. Each line specifies an incoming, or an outgoing edge. The general pattern of a line specifying a dependence edge is:

```
    (READ|WRITE|RW) IDa (<-|->) [(condition) ?] IDb peer(params)
                                           [: IBc peer(params)] [type]
```

The keywords `READ`, `WRITE` and `RW` specify if the corresponding data will be read, written, or both by the tasks of this task class. The direction of the arrow specifies whether a given edge is input, or output. A right pointing arrow specifies an output edge, which, for this example, means that each task, unmqr(k_i,n_i), of the task class unmqr(k, n) will modify the given data and the task (or tasks) specified on the right hand side of the arrow will need to receive the data from task unmqr(k_i,n_i), once this task has been completed. Conversely, a left pointing arrow specifies that the corresponding data needs to be received from the task specified on the right hand side of the arrow. The input and output identifiers (`IDa` and `IDb`) are used, in conjunction with the tasks on the two ends of an edge, to uniquely identify an edge. On the right hand side of each arrow there is a) an optional, conditional ternary operator “?:”; b) a unique identifier and an expression that specifies the peer task (or tasks) for this edge; c) an optional type specification. When a ternary operator is present, there can be one, or two identifier-task pairs as the operands of the operator. When there are two operands, the condition specifies which operand should be used as the peer task (or tasks). Otherwise, the condition specifies the values of the parameters for which the edge exists. For example, the line:

```
RW G <- (k==0) ? B DAGUE_IN_A(0,n) : M tsmqr(k-1,k,n)
```

specifies that, given specific numbers k_i and n_i, task unmqr(k_i,n_i) will receive data from task `DAGUE_IN_A(0,`n_i`)`, if, and only if, k_i has the value zero. Otherwise, unmqr(k_i, n_i) will receive data from task tsmqr($k_i - 1$, k_i, n_i)). Symmetrically, the JDF of task class `DAGUE_IN_A(i,j)` contains the following edge:

```
RW B -> (0==i) & (j>=1) ? G unmqr(0,j)
```

that uniquely matches the aforementioned incoming edge of unmqr(k,n) and specifies that for given numbers I and J, task `DAGUE_IN_A(`I,J`)` will send data to unmqr(0,J) if and only if I is equal to zero and J is greater or equal to one.

The next component of an edge specification is the task, or tasks that constitute this task’s peer for this dependence edge. All the edges shown in the example of Figure 24 specify a single task as the peer of each task of the class unmqr(k,n) (i.e., for each specific pair of numbers k_i and n_i). The JDF syntax also allows for expressions that specify a range of tasks as the receivers of the data. Clearly, since unmqr(k,n) receives from geqrt(k) (as is specified by the first edge line in Figure 24), for each value k_i, task geqrt(k_i) must send data to multiple tasks from the task class unmqr(k,n) (one for each value of n, within n’s valid range). Therefore, one of the edges of task class geqrt(k) will be as follows:

```
RW C -> (k<=A.nt-2) ? E unmqr(k, (k+1)..(A.nt-1))
```

In this notation, the expression `(k+1)..(A.nt-1)` specifies a range which guides the DAGUE runtime to broadcast the corresponding data to several receiving tasks. At first glance it might seem that the condition "`k<=A.nt-2`" limiting the possible values for the parameter "*k*" in the outgoing edge of geqrt(k) (shown above) is not sufficient since it only bounds *k* by `A.nt-2` while in the execution space of unmqr(k,n), *k* is also upper bound by `A.mt-1`. However, this additional restriction is guaranteed since the execution space of geqrt(k) (not shown here) bounds *k* by `A.mt-1`. In other words, in an effort to minimize wasted cycles at runtime, we limit the conditions that precede each edge to those that are not already covered by the conditions imposed by the execution space.

Finally, the last component of an edge specification is the type of the data exchanged during possible communications generated by this edge. This is an optional argument and it corresponds to an MPI datatype, specified by the developer. The type is used to optimize the communication by avoiding the transfer of data that will not be used by the task (the datatype does not have to point to a contiguous block of memory). This feature is particularly useful in cases where the operations, instead of being performed on rectangular data blocks, are applied on a part of the block, such as the upper, or lower triangle in the case of QR.

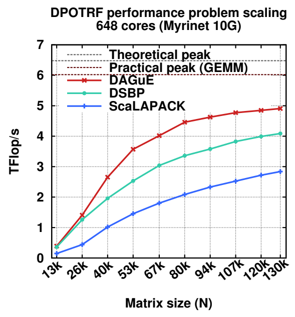
Following the dependence edges, there is the body of the task class. The body specifies how the runtime can invoke the corresponding codelet that will perform the computation associated with this task class. The specifics of the body are not related to the dataflow of the problem, so they are omitted from Figure 24 and are discussed in Section 4.1.

4.5. Performance Evaluation of the linear algebra layer DPLASMA

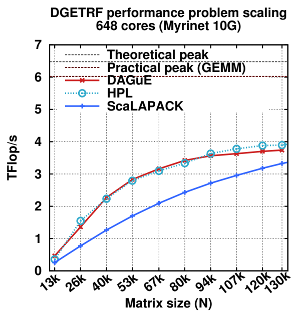
The performance of the DAGUE runtime have been extensively studied in related publications [13,14,15]. The goal here is to illustrate the performance results that can be achieved by the porting of linear algebra code to the DAGUE framework. Therefore, we present a summary of these result, to demonstrate that the tool chain achieves its main goals of overall performance, performance portability, and capability to process different non-trivial algorithms.

The experiments we summarize here have been conducted on three different platforms. The Griffon platform is one of the clusters of Grid'5000 [12]. We used 81 dual socket Intel Xeon L5420 quad core processors at 2.5 GHz to gather 648 cores. Each node has 16GB of memory, and is interconnected to the others by a 20 Gbs Infiniband network. Linux 2.6.24 (Debian Sid) is deployed on these nodes. The Kraken system of the University of Tennessee and National Institute for Computational Science (NICS) is hosted at the Oak Ridge National Laboratory. It is a Cray XT5 with 8,256 compute nodes connected on a 3D torus with SeaStar. Each node has a dual six-core AMD Opteron cadenced at 2.6GHz. We used up to 3,072 cores in the experiments we present here. All nodes have 16GB of memory, and run the Cray Linux Environment (CLE) 2.2.

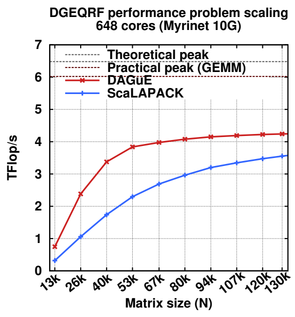
The benchmark consists of three popular dense matrix factorizations: Cholesky, LU and QR. The Cholesky factorization solves the problem $Ax = b$, where A is symmetric and positive definite. It computes the real lower triangular matrix with positive diagonal elements L such that $A = LL^T$. The QR factorization has been presented in previous sections, to explain the functionality and behavior of DAGUE. It offers a numerically stable way of solving full rank underdetermined, overdetermined, and regular square



(a) Cholesky Factorization

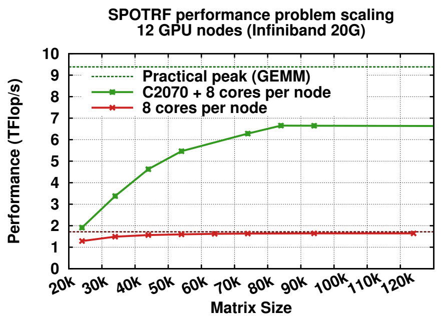


(b) LU Factorization

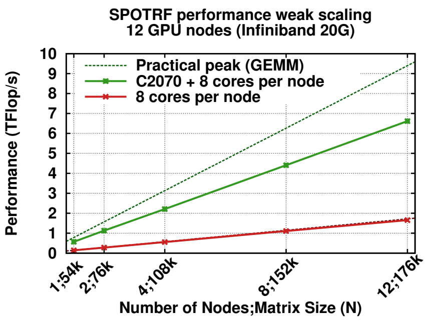


(c) QR Factorization

Figure 26. Performance comparison on the Griffon platform with 648 cores.



(a) Efficiency at 12 nodes



(b) Scalability

Figure 27. Performance of DAGuE Cholesky on the Dancer GPU accelerated cluster.

linear systems of equations. It computes Q and R such that $A = QR$, Q is a real orthogonal matrix, and R is a real upper triangular matrix. The LU factorization with partial pivoting of a real matrix A has the form $PA = LU$ where L is a real unit lower triangular matrix, U is a real upper triangular matrix, and P is a permutation matrix.

All three of these operations are implemented in the ScaLAPACK numerical library [10]. In addition, some of these factorizations have more optimized versions, we used the state of the art version for each of the existing factorizations to measure against. The Cholesky factorization has been implemented in a more optimized way in the DSBP software [26], using static scheduling of tasks, and a specific, more efficient, data distribution. The LU factorization with partial pivoting is also solved by the well known High Performance LINPACK benchmark (HPL) [23], used to measure the performance of high performance computers. We have distributed the initial data following a classical 2D-block cyclic distribution used by ScaLAPACK, and used the DAGuE runtime engine to schedule the operations on the distributed data. The kernels consist of the BLAS operations referenced by the sequential codes, and their implementation was the most efficient available on each of the machine.

Figure 26 presents the performance measured for DAGuE and ScaLAPACK, and when applicable DSBP and HPL, as a function of the problem size. 648 cores on 81

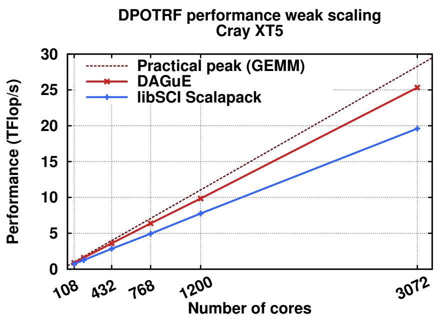


Figure 28. Scalability on the Kraken platform.

multi-core nodes have been used for the distributed run, and the data was distributed according to a 9×9 2D block-cyclic grid for DAGUE. A similar distribution was used for ScaLAPACK, and the other benchmarks when appropriate, and the block size was tuned to provide the best performance on each setup. As the figures illustrate, on all benchmarks, and for all problem sizes, the DAGUE framework was able to outperform ScaLAPACK, and perform as well as the state of the art, hand-tuned codes for specific problems. The DAGUE solution goes from the sequential code to the parallel run completely automatically, but is still able to outperform DSBP, and competes with the HPL implementation on this machine.

Figure 27a presents the performance of the DAGUE Cholesky algorithm on a GPU cluster, featuring 12 Fermi C2070 accelerators (one per node). Without GPU accelerators, the DAGUE runtime extracts the entire available performance; asymptotic performance matches the performance of the GEMM kernel on this processor, which is an upper bound to the effective peak performance. When using one GPU accelerator per node, the total efficiency reaches as much as 73% of the GEMM peak, which is a 54% efficiency of the theoretical peak (typical GPU efficiency is lower than CPU efficiency; the HPL benchmark on the TianHe-1A GPU system reaches a similar 51% efficiency, which compares with 78% on the CPU based Kraken machine). Scalability is a concern with GPU accelerators, as they provoke a massive imbalance between computing power and network capacity. Figure 27b presents the Cholesky factorization weak scalability (number of nodes vary, problem size is growing accordingly to keep memory load per node constant) on the GPU enabled machine. The figure outlines the perfect weak scalability up to 12 GPU nodes.

Last, Figure 28 compares the performance of the DAGUE implementation of these three operations with the libSCI implementation, specifically tuned by Cray for this machine. The value represented is the relative time overhead of DAGUE compared to libSCI for different matrix sizes and the number of nodes on the QR factorization (similar weak scaling as in the previous experiment, N=454000 on 3072 cores). On this machine, the DAGUE runtime can effectively use only 11 of 12 cores per node for compute tasks; due to kernel scheduler parameters (long, non-preemptive time quantum), the MPI thread must be exclusively pinned to a physical core to avoid massive and detrimental message jitter. Even considering that limitation, which is only technical and could be overcome by a native port of the runtime to the Portals messaging library instead of MPI, the DAGUE implementation competes favorably with the extremely efficient libSCI QR factorization.

The DAGuE approach demonstrates an excellent scalability, up to a massive number of nodes, thanks to the distributed evaluation of the DAG not requiring centralized control nor complete unrolling of the DAG on each node.

On different machines, the DAGuE compiler coupled with the DAGuE runtime significantly outperformed standard algorithms, and competed closely, usually favorably, with state-of-the-art optimized versions of similar algorithms, without any further tuning process involved when porting the code between radically different platform types. Another significant fact to be highlighted is the sizes of the problem where DAGuE achieves peak performance. In all graphs in Figure 26 one can notice that while Scalapack asymptotically reaches peak performance, for some of the algorithms DAGuE achieves the same level of performance on data 4 times smaller (in the case of Cholesky, Scalapack achieves 3TFlop/s on Griffon when $N = 130K$, while DAGuE reaches the same level for $N = 44K$).

5. Conclusion

Although hardware architectural paradigm shifts are threatening the scientific productivity of dense linear algebra codes, we have demonstrated that slightly changing the execution paradigm, and using a dataflow representation extracted from a decorated sequential code, dense matrix factorization can reach excellent performance. The DPLASMA package aims at providing the same functionalities as the ScaLAPACK legacy package, but using a more modern approach, based on tile algorithm and dataflow representation, that enables better cache reuse and asynchrony, which are paramount features to perform on multicore nodes. Furthermore, the DAG dataflow representation enables the algorithm to adapt easily to a variety of differing and heterogeneous hardware, without involving a major code refactoring for each target platform. We describe how the DPLASMA project uses the DAGuE framework to convert a decorated sequential code (which can be executed efficiently on multicore machines, but not on distributed memory systems), into a concise DAG dataflow representation. This representation is then altered by the programmer to add data distribution and task affinity on distributed memory. The resulting intermediate format is then compiled into a series of runtime hooks incorporating a DAG scheduler that automatically orchestrates the resolution of remote dependencies, orchestrates the execution to favor cache locality and other scheduling heuristics, and accounts for the presence of heterogeneous resources such as GPU accelerators. This description gives insight to linear algebra programmers as to the methods, challenges and solutions involved in porting their code to a dataflow representation. The performance analysis section demonstrates the vast superiority of the DAG based code over legacy programming paradigms on newer multicore hardware.

6. Summary

The tumultuous changes occurring in the computer hardware space, such as flatlining of processor clock speeds after more than 15 years of exponential increases, mark the end of the era of routine and near automatic performance improvements that the research community had previously enjoyed [44]. Three main factors converged to force processor architects to turn to multicore and heterogeneous designs and, consequently, bring an end to the "free ride." First, system builders have encountered intractable physical barriers – too much heat, too much power consumption, and too much leaking voltage – to further increases in clock speeds. Second, physical limits on the number of pins and bandwidth on a single chip mean that the gap between processor performance and memory performance, which was already bad, has gotten increasingly worse. Consequently, the design trade-offs made to address the previous two factors rendered commodity processors, absent any further augmentation, inadequate for the purposes of extreme scale systems for advanced applications. And finally, the exponential growth of transistor count on the heels of the stubbornly alive Moore's law [36] and Dennard's scaling law [21]. This daunting combination of obstacles forced the designers of new multicore and hybrid systems to explore architectures that software built on the old model are unable to effectively exploit without radical modification.

To develop software that will perform well on extreme scale systems with thousands of nodes and millions of cores, the list of major challenges that must now be confronted is formidable:

- dramatic escalation in the costs of intrasystem communication between processors and/or levels of memory hierarchy;
- increased hybridization of processor architectures (mixing CPUs, GPUs, etc.), in varying and unexpected design combinations;
- cooperating processes must be dynamically and unpredictably scheduled for asynchronous execution due to high levels of parallelism and more complex constraints;
- software will not run at scale without much better resilience to faults and increased robustness; and
- new levels of self-adaptivity will be required to enable software to modulate process speed in order to satisfy limited energy budgets.

The software project presented above meets the aforementioned challenges and allows the users to run their computationally intensive codes at scale and to achieve a significant percentage of peak performance on the contemporary hardware systems that may soon break the barrier of 100 Pflop/s. This is achieved by finding and integrating solutions to problems in two critical areas: novel algorithm design as well as management of parallelism and hybridization.

References

[1] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *Proceedings of the 9th International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2011.

[2] E. Agullo, J. Demmel, Jack J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:12-37, 2009.

[3] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992.

[5] 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).

[6] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual, Version 2.0*, 2008.

[7] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, August 2001.

[8] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, Jack J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming, Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPOPP'95*, pages 207–216, Santa Barbara, CA, July 19-21 1995. ACM.

[12] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *IEEE International Symposium on Parallel and Distributed Processing, 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11)*, pages 1432–1441, Anchorage, AK, May 2011.

[14] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38:37-51, 2011.

[15] George Bosilca, Aurelien Bouteiller, Thomas Herault, Pierre Lemarinier, Narapat Ohm Saengpatsa, Stanimire Tomov, and Jack J. Dongarra. Performance portability of a GPU enabled factorization with the DAGuE framework. In *Proceedings of the IEEE Cluster 2011 Conference (PPAC Workshop)*, pages 395–402. IEEE, September 2011.

[16] A. Buttari, J. Langou, J. Kurzak, and Jack J. Dongarra. Parallel tiled QR factorization for multicore architectures. Technical report, Innovative Computing Laboratory, University of Tennessee, 2007.

[17] Anthony M. Castaldo and R. Clint Whaley. Scaling LAPACK panel operations using parallel cache assignment. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 223–232, 2010.

[18] C interface to the BLAS. `http://www.netlib.org/blas/blast-forum/cblas.tgz`.

[19] CLAPACK (f2c'ed version of LAPACK). `http://www.netlib.org/clapack/`.

[20] *CUDA CUBLAS Library*.

[21] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid State Circuits*, 9(5):256–268, 1974. `http://dx.doi.org/10.1109/JSSC.1974.1050511`DOI: 10.1109/JSSC.1974.1050511.

[22] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In Koen De Bosschere, Erik H. D'Hollander, Gerhard R. Joubert, David Padua, Frans Peters, and Mark Sawyer, editors, *Advances in Parallel Computing, Special Issue Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22, pages 429–436. IOS Press, STM Publishing House, Amsterdam, The Netherlands, 2012. ISBN 978-1-61499-040-6 (print), ISBN 978-1-61499-041-3 (online).

[23] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present and

future. *Concurrency Computat.: Pract. Exper.*, 15(9):803–820, 2003.

[24] f2c. `http://www.netlib.org/f2c/`.

[25] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[26] Fred G. Gustavson, Lars Karlsson, and Bo Kågström. Distributed SBP Cholesky factorization algorithms with near-optimal scheduling. *ACM Trans. Math. Softw.*, 36(2):1–25, 2009.

[27] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[28] Portable hardware locality (HWLOC). `http://www.open-mpi.org/projects/hwloc/`.

[29] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, Department of Computer Science and Engineering, University of Notre Dame, September 28 2008.

[30] 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.

[31] LAPACK C interface. `http://www.netlib.org/lapack/lapacke.tgz`.

[32] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for GPUs. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[33] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. *ACM TOMS*, 39(3), 2013. In publication.

[34] Piotr Luszczek, Hatem Ltaief, and Jack Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *Proceedings of IPDPS 2011*, Anchorage, Alaska, May 16-20 2011.

[35] Allen D. Malony and Wolfgang E. Nagel. The open trace format (OTF) and open tracing for HPC. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[36] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 19 1965.

[37] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proc. of High Performance Computing for Computational Science (VECPAR' 10)*, June 22-25, 2010.

[38] National Research Council Committee on the Potential Impact of High-End Computing on Illustrative Fields of Science and Engineering. *The Potential Impact of High-End Capability Computing on Four Illustrative Fields of Science and Engineering*. National Academies Press, Washington, DC, 2008.

[39] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html, 2009.

[40] NVIDIA. *NVIDIA CUDA^{TM} Best Practices Guide Version 3.0*. NVIDIA Corporation, February4 2010.

[41] NVIDIA. *NVIDIA CUDA^{TM} Programming Guide Version 3.0*. NVIDIA Corporation, February20 2010.

[42] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, 2008.

[43] R. Schreiber and C. van Loan. A storage-efficient WY representation for products of Householder transformations. *J. Sci. Stat. Comput.*, 10:53–57, 1991.

[44] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[45] Intel Threading Building Blocks. `http://www.threadingbuildingblocks.org/`.

[46] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma, 11/2009.

[47] University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.2*, November 2009.

[48] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[49] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[50] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK users' guide: Queuing and runtime for

kernels. technical report UT-ICL-11-02, University of Tennessee Innovative Computing Laboratory, Knoxville, Tennessee 37996, April 2011.