DOI: 10.14529/jsfi150405

Parallel Programming Models for Dense Linear Algebra on Heterogeneous Systems

M. Abalenkovs[1], *A. Abdelfattah*[2], *J. Dongarra* [1,2,3], *M. Gates*[2], *A. Haidar*[2], *J. Kurzak*[2], *P. Luszczek*[2], *S. Tomov*[2], *I. Yamazaki*[2], *A. YarKhan*[2]

We present a review of the current best practices in parallel programming models for dense linear algebra (DLA) on heterogeneous architectures. We consider multicore CPUs, stand alone manycore coprocessors, GPUs, and combinations of these. Of interest is the evolution of the programming models for DLA libraries – in particular, the evolution from the popular LAPACK and ScaLAPACK libraries to their modernized counterparts PLASMA (for multicore CPUs) and MAGMA (for heterogeneous architectures), as well as other programming models and libraries. Besides providing insights into the programming techniques of the libraries considered, we outline our view of the current strengths and weaknesses of their programming models – especially in regards to hardware trends and ease of programming high-performance numerical software that current applications need – in order to motivate work and future directions for the next generation of parallel programming models for high-performance linear algebra libraries on heterogeneous systems.

Keywords: Programming models, runtime, HPC, GPU, multicore, dense linear algebra.

Introduction

Parallelism in today's computer architectures is pervasive – not only in systems from large supercomputers to laptops, but also in small portable devices like smart phones and watches. Along with parallelism, the level of heterogeneity in modern computing systems is also gradually increasing. Multicore CPUs are often combined with discrete high-performance accelerators like graphics processing units (GPUs) and coprocessors like Intel's Xeon Phi, but are also included as integrated parts with system-on-chip (SoC) platforms, as in the AMD Fusion family of application processing units (APUs) or the NVIDIA Tegra mobile family of devices. To extract full performance from systems like these, the heterogeneity makes the parallel programming for technical computing problems extremely challenging.

Furthermore, when considering the future of research and development on parallel programming models, the two factors that should be looked at first are the projected changes in system architectures that will define the landscape on which HPC software will have to execute, and the nature of the applications and application communities that this software will ultimately have to serve. In the case of system architectures, we can get a reasonable picture of the road that lies ahead by examining the roadmaps of major hardware vendors, as well as the next generation of supercomputing platforms that are in the works, discussed in Section 1. In the case of applications, we can extract trends of what may be needed from applications of high current interest like big-data analytics, machine learning, and more, in Section 5.

A traditional way to take advantage of parallelism without tedious parallel programming is through the use of already parallelized HPC libraries. Among them, linear algebra (LA) libraries, and in particular DLA, are of high importance since they are fundamental to a wide range of scientific and engineering applications. Hence, these applications will not perform well unless LA

[1] University of Manchester, Manchester, UK
[2] University of Tennessee, Knoxville, TN, USA
[3] Oak Ridge National Laboratory, Oak Ridge, TN, USA

libraries perform well. Thus, by concentrating on best practices in parallel programming models for DLA on heterogeneous architectures, we will indirectly address other areas, as long as the DLA libraries discussed and their programming models are interoperable with third party tools and standards.

To describe how to program parallel DLA, we first focus on the generic parallel programming models today (Section 2), and second, on the evolution in the programming models for DLA libraries – from the popular LAPACK and ScaLAPACK to their modernized counterparts in PLASMA, for multicore CPUs, and MAGMA, for heterogeneous architectures, as well as in other programming models and libraries (Section 3). The current best practices are summarized by the task based programming model, reviewed in Section 4. Besides providing insights into the programming techniques of the libraries considered, we outline our view of the current strengths and weaknesses of their programming models – especially in regards to hardware trends and ease of programming high-performance numerical software that current applications need – in order to motivate work and future directions for the next generation of parallel programming models for high-performance LA libraries on heterogeneous systems (Section 5).

1. Hardware trends in heterogeneous system designs

Some future trends in the heterogeneous system designs are evident from USA's DOE plans for the next generation of supercomputers. The aim is to deploy three different platforms by 2018, each with over 150 petaflops of peak performance [18]. Two of them, named Summit and Sierra, will be based on IBM OpenPOWER and NVIDIA GPU accelerators, and the third, Aurora, will be based on the Xeon Phi platform. Summit and Sierra will follow the hybrid computing model by coupling powerful latency-optimized processors with highly parallel throughput-optimized accelerators. They will rely on IBM Power9 CPUs, NVIDIA Volta GPUs, an NVIDIA NVLink interconnect to connect the hybrid devices within each node, and a Mellanox Dual-Rail EDR Infiniband interconnect to connect the nodes. The Aurora system, on the other hand, will offer a more homogeneous model by utilizing the Knights Hill Xeon Phi architecture, which, unlike the current Knights Corner, will be a stand-alone processor and not a slot-in coprocessor, and will also include integrated Omni-Path communication fabric. All platforms will benefit from recent advances in 3D-stacked memory technology, and promise major performance improvements:

- CPU memory bandwidth is expected to be between 200 GB/s and 300 GB/s using HMC;
- GPU memory bandwidth is expected to approach 1 TB/s using HBM;
- GPU memory capacity is expected to reach 60 GB (NVIDIA Volta);
- NVLink is expected to deliver from 80 up to 200 GB/s of CPU-to-GPU bandwidth;
- In terms of computing power, the Knights Hill is expected to be between 3.6 and 9 teraflops, while the NVIDIA Volta is expected to be around 10 teraflops.

The hybrid computing model is here to stay, and memory systems will become ever more complicated. Moreover, the interconnection technology will be seriously lagging behind the computing power. Aurora's nodes may have 5 teraflops of computing power with a network injection bandwidth of 32 GB/s; Summit's and Sierra's nodes are expected to have 40 teraflops of computing power and a network injection bandwidth of 23 GB/s. This creates a gap of two orders of magnitude for Aurora and three orders of magnitude for Summit and Sierra, leaving data-heavy workloads in the lurch and motivating the search for algorithms that minimize data movement, and programming models that facilitate their development.

The gap between compute power and interconnect levels will continue to widen, diversify, and deepen not just between nodes, as pointed out, but also within the node's entire memory hierarchy, as evident from hardware developments such as a significant increase in the *last level cache* (LLC) size in the CPU, and L2 cache in the GPU, use of NVLink with a higher interconnect bandwidth, and 3D-stacked memories (see also Figure 1, Left). These trends will thus require some support from the programming model for hierarchical tasks (and their communications) for blocking computations over the memory hierarchies [44] in order to reduce data movement. In fact, the community agrees that the biggest challenges to future application performance lie with efficient node-level execution that can use all the resources in the node [58], thus motivating this paper's focus to be on the programming models at the node level.

2. General purpose parallel programming models

A thorough survey of general purpose parallel programming models is beyond the scope of this writing and would exceed space constraints imposed on the content. However, there already exist overview-style publications that provide a comprehensive list of programming models for manycore [58] and heterogeneous hardware [42]. The success of the accelerator hardware resulted in an unfortunate proliferation of software solutions for easing the programming burden that the new systems bring to bear. Even with the recent coalescing of the plethora of products around major standardization themes, the end user is still left with the choice of one of many *open* specifications: OpenCL [38], OpenMP 4 [48, 49], and OpenACC 2 [46]. Despite similarity in the name, there is a vast disparity between the levels of abstraction that these standards offer to the programmer. OpenCL defines a software library that exposes a very low-level view of the hardware. The other two standards, OpenMP and OpenACC, are language-based solutions that hide tremendous amounts of detail and offer a much more succinct means of expressing the offloading paradigm of programming. An additional consideration is the fact that many algorithms, and in particular, algorithms in the area of DLA, are designed to use the Basic Linear Algebra Subprograms (BLAS) [20] standard, and therefore it is important to note that, as of today, there is no fully compliant BLAS implementation in OpenACC or OpenMP. Instead, there is a possibility of calling vendor BLAS which might be, but not necessarily are, implemented with OpenCL. Thus, from the very specific perspective of library development, only OpenCL offers a sufficient breadth of features essential for performance, portability, and maintainability, without the burden of extraneous software dependencies. Unfortunately, the actual performance achieved by OpenCL implementations on some platforms does not match less portable interfaces that are perfected for the hardware by the vendor [21]. A lower level, and consequently closer to hardware, model is based on dataflow and streams [51]. On the NVIDIA hardware, the CUDA Toolkit a is widely known and used software stack that exposes the underlying hardware through a simple abstraction of streams that run on processing pipelines that are efficiently scheduled with the help of a hardware-based scheduler. On the Xeon Phi hardware accelerator from Intel, the streaming model for offload computing is also available [45] with the additional advantage of offloading across not just the Intel discrete accelerators but also the CPU processors. Finally for completeness, we would also like to mention C++ AMP [13, 23], which is a standard that targets Microsoft's DirectX and DirectCompute software stacks. AMP is much closer conceptually to OpenMP and OpenACC in terms of offering a single source code solution with restricted syntax for the offload regions to account for limited capabilities of the accelerator hardware. Note that OpenMP is much more permissive in terms of syntax due to Xeon Phi's much more CPU-like design. The recent porting efforts might make AMP much more relevant outside of the Windows and Visual Studio ecosystems as there is a port to LLVM that is advancing in functionality due to efforts by MultiCoreWare and overseen by the HSA Foundation[4].

3. Special purpose parallel programming models

3.1. LAPACK and ScaLAPACK

The LAPACK's programming model [4] is based on expressing algorithms in terms of BLAS calls, described below. Subsequently, LAPACK can achieve high efficiency, provided that highly efficient machine-specific BLAS implementations are provided, e.g., by the manufacturer. Since the 1980s this model has turned out to be very successful for cache-based shared-memory vector and parallel processors with multi-layered memory hierarchies.

ScaLAPACK is the distributed-memory implementation of LAPACK, where BLAS is replaced by a parallel BLAS (PBLAS). This design makes it possible for the ScaLAPACK routines to be quite similar to their LAPACK counterparts. Efficiency within a node is extracted through the use of BLAS. Load balance and scalability is largely achieved by the way matrices are distributed over the processes – namely, in a 2D block cyclic distribution [11].

3.1.1. BLAS

The original Level 1 BLAS [40] and an extended, Level 2 BLAS for matrix-vector operations [19] were adopted as standard and used in a wide range of numerical software, including LINPACK [16]. Unfortunately, while successful for the vector-processing machines at the time, Level 2 BLAS was not a good fit for the cache-based machines that emerged in the 1980s because the operations are memory bound and thus, do not use the multi-layered memory hierarchies; thereby spending too much time moving data instead of doing useful floating-point operations. To effectively use caches, it was preferable to express computations as matrix-matrix operations. Matrices were split into small blocks so that basic operations were performed on blocks that fit into cache memory. This approach avoids excessive movement of data to and from memory and gives a surface-to-volume effect for the ratio of operations to data movement. Subsequently, Level 3 BLAS was proposed [20], covering the main types of matrix-matrix operations, and LINPACK was redesigned into LAPACK to use the new Level 3 BLAS where possible.

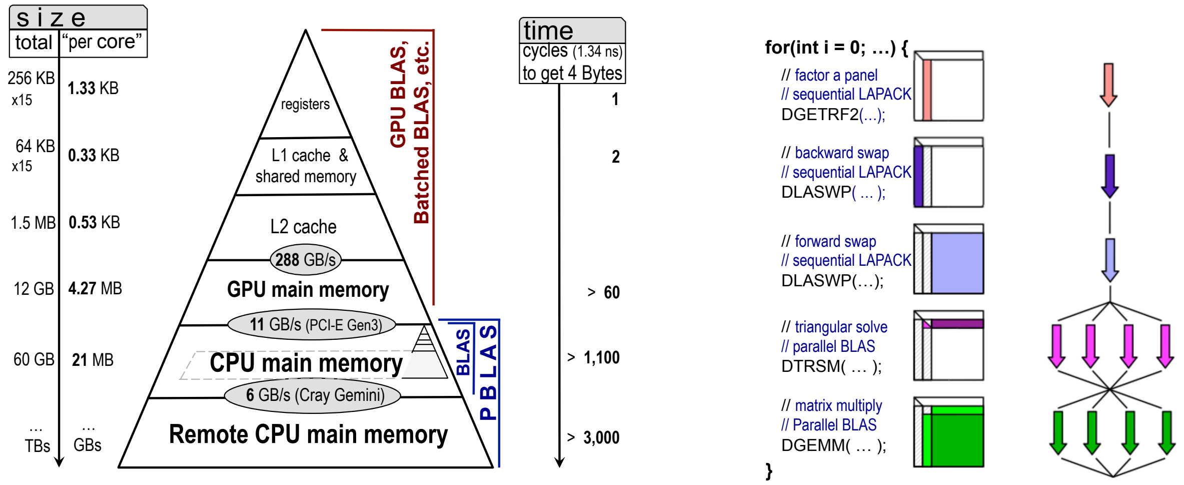
To account for the deep memory hierarchies today, efficient Level 3 BLAS implementations feature multilevel blocking where the matrix-matrix computations are split hierarchically into blocks that fit into corresponding levels of the memory hierarchy. Thus, we point out that a programming model based on using Level 3 BLAS provides support for hierarchical tasks, as needed and highlighted in Section 1, and is still the best parallel programming model (or at least the main component in custom models; discussed below) for DLA at the present time.

3.1.2. PBLAS

Custom parallel programming models for DLA are usually built on top of BLAS, and their purpose is to ease parallel programming efforts and facilitate obtaining scalable high-performance. The use of BLAS is one main component in obtaining high-performance. Another, which is orthogonal to the benefits of using BLAS, is the support of hierarchical tasks for the

[4] http://www.hsafoundation.com/bringing-camp-beyond-windows-via-clang-llvm/

memory hierarchies not covered by BLAS. In the case of PBLAS [12], this is the node-to-node interconnection layer, denoted as the *Remote CPU main memory* layer in Figure 1. PBLAS is still a very successful parallel programming abstraction for DLA on distributed-memory systems. As pointed out, the biggest challenges to future application performance lie within efficient node-level execution, where nodes are becoming highly parallel and heterogeneous, which is our main focus. With that in mind, we leave the distributed-memory case with PBLAS as an excellent example of how to organize DLA to capture one more level of hierarchy (internodal) over BLAS, and thus allowing coding algorithms for distributed-memory systems to look like the ones for shared-memory systems built on top of BLAS (as in LAPACK).



```
for(int i = 0; ...) {
    // factor a panel
    // sequential LAPACK
    DGETRF2(...);

    // backward swap
    // sequential LAPACK
    DLASWP( ... );

    // forward swap
    // sequential LAPACK
    DLASWP(...);

    // triangular solve
    // parallel BLAS
    DTRSM( ... );

    // matrix multiply
    // Parallel BLAS
    DGEMM( ... );
}
```

Figure 1. Left: Memory hierarchy of a CUDA core of an NVIDIA K40c GPU with 2,880 CUDA cores. **Right**: A fork-join parallelism example in an LU factorization using parallel BLAS.

3.2. The fork-join parallel model

The use of parallel BLAS – for either shared-memory systems (e.g., single GPU, or multicore CPUs system) or distributed-memory systems (as in PBLAS) – results in a parallel execution model often referred to as *fork-join*. The naming arises from the fact that in this model a sequence of BLAS calls is implicitly synchronized after each individual BLAS call (join), and the routines by themselves run in parallel (fork), as illustrated in Figure 1, Right. This is a powerful model as it is simple to use and develops scalable high-performance parallel algorithms. However, the overhead of synchronization and idle processors/cores can be large for small problems. This motivates the search for improved models such as the task approach in PLASMA, or the batched approach in MAGMA [28] for very small problems and even tiny problems (sub-vector/warp in size) that may require a few tasks to be grouped for execution on a single core [1].

3.3. PLASMA and the task approach

Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [2] was developed to address the performance deficiency of the LAPACK library on multicore processors. LAPACK's shortcomings stem from the fact that its algorithms are coded in a serial fashion, with parallelism only possible inside the set of BLAS, which forces a fork-join style of multithreading. In contrast, algorithms in PLASMA are coded in a way that allows for much more powerful dataflow parallelism [10, 39]. Currently, PLASMA supports a substantial subset of LAPACK's functionality, including solvers for linear systems of equations, linear least squares problems, singular value problems, symmetric eigenvalue problems and generalized symmetric eigenvalue problems. It also provides a full set of Level 3 BLAS routines for matrices stored in tile layout and a highly optimized (cache-efficient and multithreaded) set of routines for layout translation [24]. PLASMA is based on three principles: tile algorithms, tile matrix layout and task-based scheduling, all of which are geared towards efficient multithreaded execution.

Initially, PLASMA routines were implemented using the *Single Program Multiple Data* (SPMD) programming style, where each thread executed a statically scheduled preassigned set of tasks, while coordinating with other threads using progress tables, with busy-waiting as the main synchronization mechanism. While seemingly very rudimentary, this approach actually produced very efficient, systolic-like, pipelined processing patterns, with good data locality and load balance. At the same time, the codes were hard to develop, error prone and difficult to maintain, which motivated a move towards dynamic scheduling.

The multicore revolution of the late 2000's brought the idea back and put it in the mainstream. One of the first task-based multithreading systems that received attention in the multicore era, was the Cilk programming language, originally developed at MIT in the 1990's. Cilk offers nested parallelism based on a tree of threads that is geared towards recursive algorithms. About the same time, the idea of superscalar scheduling gained traction, based on scheduling tasks by resolving data hazards in real time, in a similar way that superscalar processors dynamically schedule instructions on CPUs.

This superscalar technique was pioneered by a software project from the Barcelona Supercomputing Center, which went through multiple names as its hardware target was changing: GridSs, CellSs, SMPSs, OMPSs, StarSs, where "Ss" stands for superscalar [9, 22, 50]. A similar development was the StarPU project from INRIA, which applied the same methodology to systems with GPU accelerators, named for its capability to schedule work to "C"PUs and "G"PUs, hence the name *PU, transliterated into StarPU [7]. Yet another scheduler was developed at Uppsala University, called SuperGlue [53]. Finally, a system called QUARK was developed at the University of Tennessee [59], and used for implementing the PLASMA numerical library [2]. At some point, all the projects mentioned received extensions for scheduling in distributed memory.

The OpenMP community has been swiftly moving forward with standardization of new scheduling techniques for multicores. First, the OpenMP 3.0 standard [47] adopted the Cilk scheduling model, then the OpenMP 4.0 standard [48] adopted the superscalar scheduling model. Not without significance is the fact that the GNU compiler suite was also quick to follow with high quality implementations of the new extensions. Motivated by these recent developments, PLASMA is currently in the process of being completely ported to the OpenMP standard.

3.4. MAGMA and the hybrid approach

The MAGMA library implements hybrid DLA routines for heterogeneous systems using both CPUs and accelerators, such as a GPU or Intel Xeon Phi. To best utilize a heterogeneous system requires understanding the strengths and weaknesses of each processor. We will briefly review the architectures and how those impact computations.

GPUs use a *single instruction multiple thread* (SIMT) model. The computation is arranged in a grid of thread blocks, which is further divided into a grid of threads. Within each thread block, threads are executed in sets; for instance a set of 32 threads known as a *warp* in NVIDIA's CUDA architecture. In each thread block, threads are not independent, but in SIMT fashion follow the same execution path. Threads may take different branches, but this incurs a significant penalty as all threads must wait for both sides of the branch to finish. Ideally, most of the time all threads execute the same instruction on different data.

As with CPUs, reusing data in registers and caches is important for maximizing the ratio of floating point operations to data transfers. This architecture means GPUs are most efficient at doing bulk computations with regular data access – exactly the kind of computations in dense matrix-matrix operations (Level 3 BLAS). For instance, the cuBLAS dgemm achieves 75% of the theoretical peak on a Kepler GPU. Tasks with little parallelism, significant branching, or that require global synchronization will perform relatively poorly on GPUs.

The Xeon Phi coprocessor is somewhat more flexible than GPU architectures. The current Knights Corner model has up to 60 compute cores (plus one core that is reserved for the operating system), with 4 hyperthreads per core, and an 8 double-precision wide vector unit. It supports traditional CPU thread models such as pthreads and OpenMP. Still, achieving top performance requires all the threads executing instructions on different data using the vector unit. As with the GPU, the Xeon Phi excels at performing dense matrix-matrix operations.

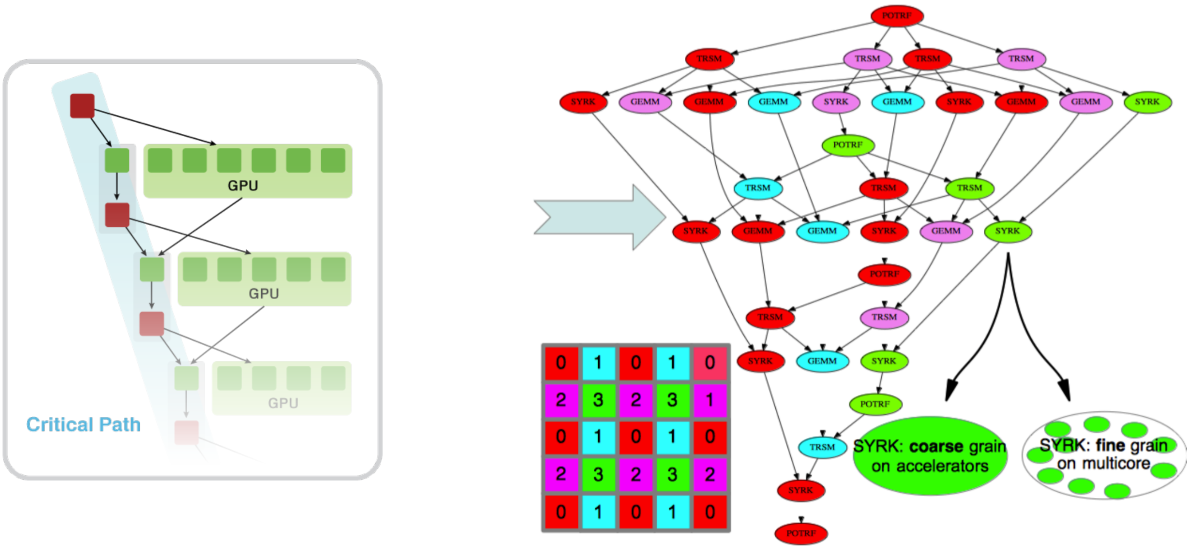
In contrast, CPUs have fewer cores and smaller vector units. They also have more complicated hardware optimizations such as out-of-order execution, branch prediction, and speculative execution. This means that CPUs perform much better at tasks with limited parallelism and significant branching.

Hence, the typical hybrid algorithm splits the overall computation into small tasks to execute on the CPU, and large update tasks to execute on the accelerator [17, 25, 29, 54], as shown in Figure 2a. For instance, in *LU* and *QR* factorizations, each step is split into a panel factorization of n_b columns, followed by a trailing matrix update. The panel factorization is assigned to the CPU (red tasks Figure 2a), and includes such decisions as selecting the maximum pivot in each column or computing a Householder reflector for each column. The trailing matrix update is assigned to the accelerator (green tasks in Figure 2a), and involves some form of matrix-matrix multiply. The block size, n_b, can be tuned to adjust the amount of work on the CPU vs. on the accelerator. Optimally, during the trailing matrix update, a look-ahead panel is updated first and sent back to the CPU (green tasks in the *critical path* in Figure 2a). Asynchronous data transfers are used to copy data between the CPU and accelerator while the accelerator continues computing. The CPU performs the next panel factorization while the accelerator continues with the remainder of the trailing matrix update. In this way, the inputs for the next trailing matrix update are ready when the current update finishes. The goal is to keep the accelerator – which has the highest performance – always busy.

This model can be extended to distributed computing, as shown in Figure 2b [30]. Here, a task scheduler assigns the trailing matrix updates, such as the SYRK Level 3 BLAS call, to either the accelerator or the multicore CPU, depending on the availability of resources.

4. Task based programming model

The task based parallel programming model is well established by now and is very successful for DLA, as illustrated with the PLASMA and MAGMA libraries. To provide parallelism, algorithms are split into computational tasks, which in the context of LAPACK algorithms translates to splitting BLAS calls into tasks. Various abstractions can be used to define the model, e.g., as illustrated for QUARK and OpenMP4 in Section 4.1. The resulting algorithms can be viewed as


GPU
GPU
GPU
Critical Path
POTRF
TRSM
SYRK
GEMM
SYRK: coarse grain on accelerators
SYRK: fine grain on multicore

0	1	0	1	0
2	3	2	3	1
0	1	0	1	0
2	3	2	3	2
0	1	0	1	0

Figure 2. **(a)** Hybrid algorithm uses a DAG approach where small tasks, typically on the critical path, are scheduled for execution on multicore CPUs, and large data-parallel tasks are scheduled on the accelerator. **(b)** Extension of the hybrid approach to distributed systems using 2D block cyclic layout as in ScaLAPACK, and local/node scheduling of hierarchical tasks for best fit to either multicore CPUs or accelerators

DAGs that can be scheduled for execution on the underlying hardware in various ways, discussed in Sections 4.2, 4.3, and 4.4.

4.1. Algorithms as DAGs

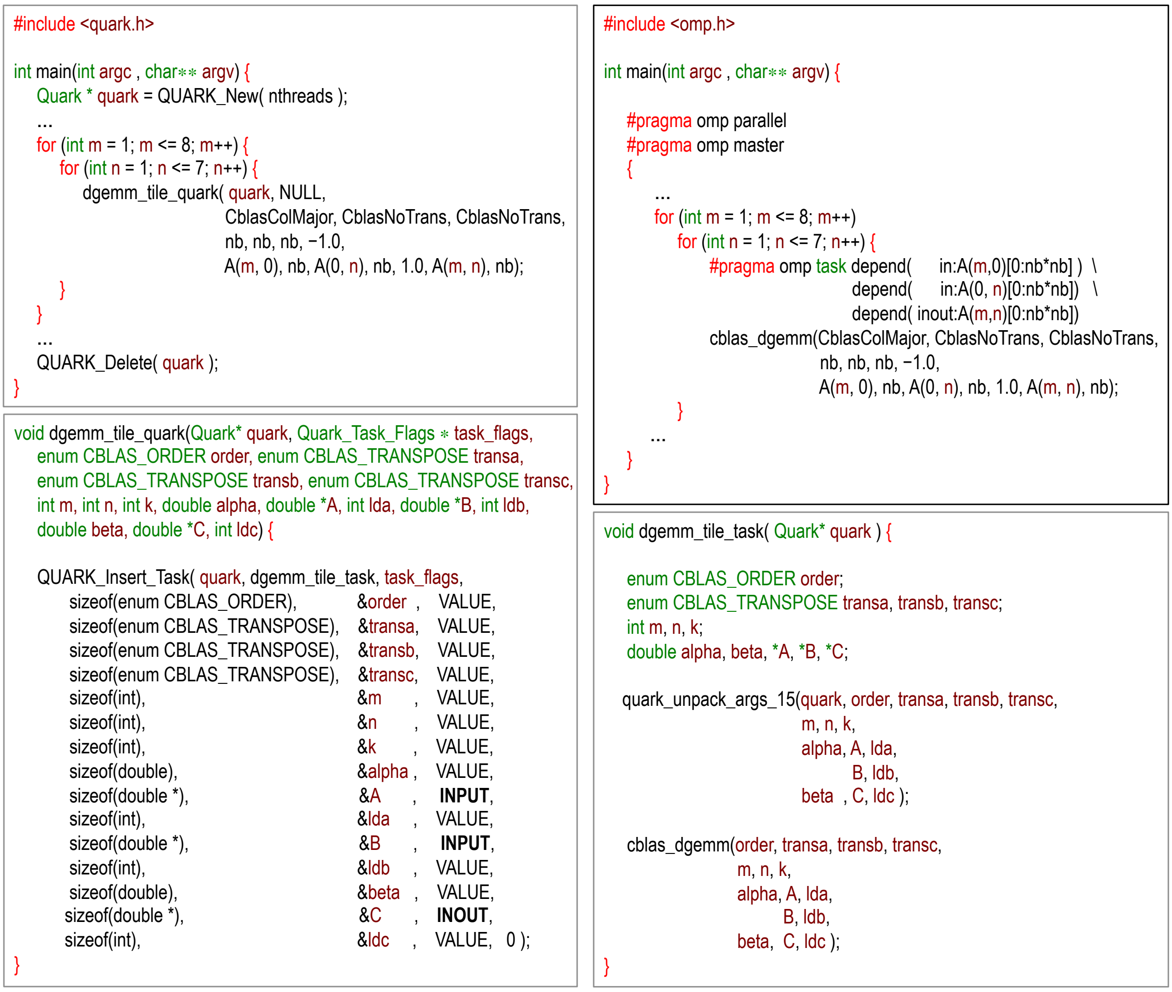
Task-based rank-k update To illustrate the DAG based programming model we present a short example of a general rank-k update (`gemm`) performed by means of two alternative approaches: (i) using the highly customizable runtime QUARK and (ii) applying the standard OpenMP4 runtime. The rank-k update expressed in Eq. (1) is an important building block of many DLA algorithms, including the LU factorization:

$$\mathbf{C} = \alpha\,\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}. \tag{1}$$

Here α and β are scalars, and $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices of sizes $m \times nb$, $nb \times n$, and $m \times n$, respectively. The rank nb is in general the algorithm's blocking size.

Figure 3 shows the example where the matrices are in tile format, and in particular, $\mathbf{A}$ is split into 8 tiles, while B is split into 7 tiles (as shown in Figure 5, Left), each of size $nb \times nb$. Thus, the computation in Eq. (1) is split into 56 tasks. In this case $\alpha = -1$ and $\beta = 1$.

DAG DAG is a well-established concept in the task-based programming world. To construct a DAG, the sequential code is split into basic independent operations, where each operation is performed by means of a new task usually assigned to a processing thread. Each task has data dependencies, specified through its arguments (with clauses like INPUT, INOUT, etc.). Prior to execution, the runtime analyzes data dependencies of each operation and constructs a task graph. The tasks to be performed become DAG nodes and the data dependencies become graph edges. The graph is unfolded on-the-fly at program execution time. Since the task graph



```
#include <quark.h>

int main(int argc , char∗∗ argv) {
    Quark * quark = QUARK_New( nthreads );
    ...
    for (int m = 1; m <= 8; m++) {
        for (int n = 1; n <= 7; n++) {
            dgemm_tile_quark( quark, NULL,
                                  CblasColMajor, CblasNoTrans, CblasNoTrans,
                                  nb, nb, nb, −1.0,
                                  A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
        }
    }
    ...
    QUARK_Delete( quark );
}
```

```
void dgemm_tile_quark(Quark* quark, Quark_Task_Flags ∗ task_flags,
    enum CBLAS_ORDER order, enum CBLAS_TRANSPOSE transa,
    enum CBLAS_TRANSPOSE transb, enum CBLAS_TRANSPOSE transc,
    int m, int n, int k, double alpha, double *A, int lda, double *B, int ldb,
    double beta, double *C, int ldc) {

    QUARK_Insert_Task( quark, dgemm_tile_task, task_flags,
        sizeof(enum CBLAS_ORDER),          &order  ,   VALUE,
        sizeof(enum CBLAS_TRANSPOSE),      &transa,    VALUE,
        sizeof(enum CBLAS_TRANSPOSE),      &transb,    VALUE,
        sizeof(enum CBLAS_TRANSPOSE),      &transc,    VALUE,
        sizeof(int),                       &m      ,   VALUE,
        sizeof(int),                       &n      ,   VALUE,
        sizeof(int),                       &k      ,   VALUE,
        sizeof(double),                    &alpha ,    VALUE,
        sizeof(double *),                  &A      ,   INPUT,
        sizeof(int),                       &lda    ,   VALUE,
        sizeof(double *),                  &B      ,   INPUT,
        sizeof(int),                       &ldb    ,   VALUE,
        sizeof(double),                    &beta   ,   VALUE,
       sizeof(double *),                   &C      ,   INOUT,
       sizeof(int),                        &ldc    ,   VALUE,   0 );
}
```

```
#include <omp.h>

int main(int argc , char∗∗ argv) {

    #pragma omp parallel
    #pragma omp master
    {
        ...
        for (int m = 1; m <= 8; m++)
            for (int n = 1; n <= 7; n++) {
                #pragma omp task depend(     in:A(m,0)[0:nb*nb] )  \
                                 depend(     in:A(0, n)[0:nb*nb])   \
                                 depend( inout:A(m,n)[0:nb*nb])
                cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                                 nb, nb, nb, −1.0,
                                 A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
            }
        ...
    }
}
```

```
void dgemm_tile_task( Quark* quark ) {

    enum CBLAS_ORDER order;
    enum CBLAS_TRANSPOSE transa, transb, transc;
    int m, n, k;
    double alpha, beta, *A, *B, *C;

   quark_unpack_args_15(quark, order, transa, transb, transc,
                            m, n, k,
                            alpha, A, lda,
                                   B, ldb,
                            beta  , C, ldc );

    cblas_dgemm(order, transa, transb, transc,
                     m, n, k,
                     alpha, A, lda,
                            B, ldb,
                     beta,  C, ldc );
}
```

Figure 3. A rank-k update example of using tasking with QUARK (Left) and OpenMP4 (Right)

might be very large even for a small program the runtime usually applies a window to process and "walk over" a limited amount of nodes. Representing application data as graph edges and computational operations (tasks) as nodes allows for achieving high levels of asynchronicity and parallelism.

QUARK vs OpenMP QUARK (QUeing And Runtime for Kernels) is a runtime engine for dynamic scheduling and execution of applications aimed at multicore/multisocket shared memory systems. This runtime implements a data flow model, where scheduling decisions are made, resolving data dependencies in a task graph. QUARK analyzes and resolves data dependencies at runtime by unfolding the DAG of tasks. QUARK is capable of DAG merging and loop reordering. At scheduling time, the runtime tries to maximize data reuse based on the data locality information. The tasks to be executed are realized by means of a FIFO queue and idling threads are allowed to perform LIFO work stealing. Apart from basic flags presented in Figure 3, QUARK features special flags to control task priorities and set data locality, accumulation, and gathering properties. With QUARK it is also possible to assign multiple tasks to a single thread, as well as switch to manual scheduling of certain tasks.

Compared to QUARK, OpenMP provides a simpler standard interface widely adopted on most modern hardware platforms. Since version 4.0, OpenMP features a `task` pragma with the `depend` keyword making specification of input/output dependencies straightforward.

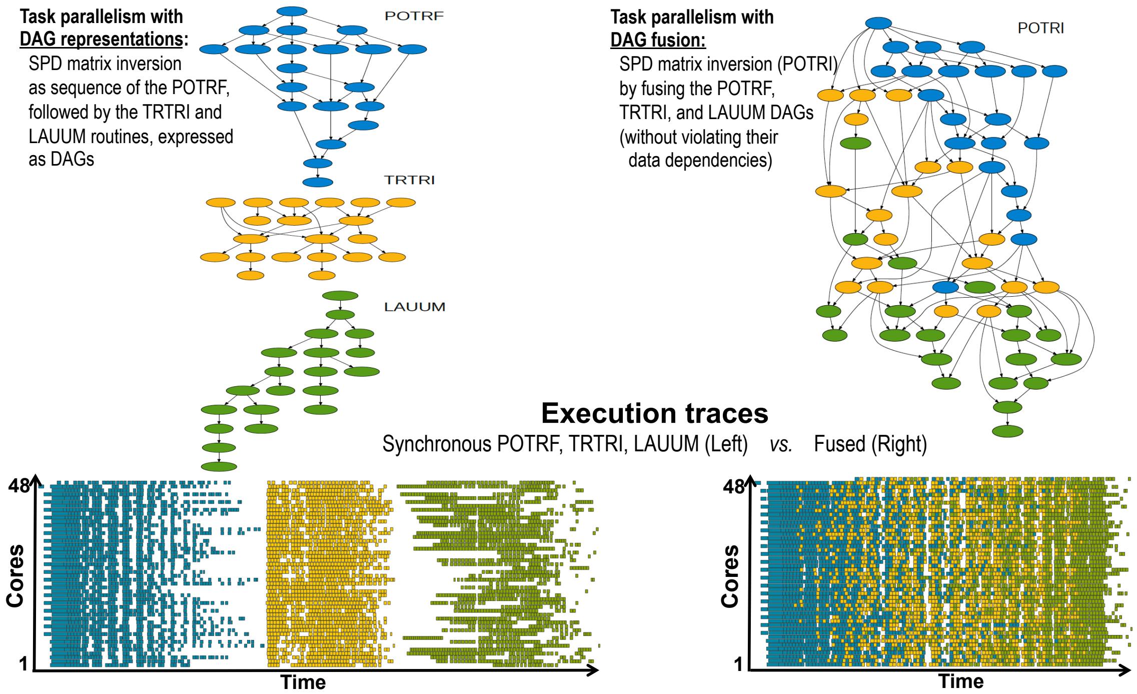


Figure 4. The DAG approach for inverting a symmetric and positive definite (SPD) matrix as a sequence of the POTRF, TRTRI, and LAUUM routines, expressed as DAGs, along with its execution trace on a 48 core CPU system (Left). The same routine, but with fused DAGs, so that data dependencies are not violated, along with its execution trace (Right)

Asynchronous DAG fusion In most execution environments the end of algorithmic step causes a synchronization before the computation can move to the next step of the algorithm. This is very evident in the fork-join implementations of linear algebra algorithms, where synchronization occurs between successive calls to the high-level (BLAS) algorithms. An asynchronous, data-driven runtime environment can dynamically merge multiple algorithms to achieve higher performances. Figure 4 shows the structure of the symmetric positive definite (SPD) inversion algorithm (aka Cholesky inversion) which is implemented using calls to three high-level routines. 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 $A^{-1} = {L^{-1}}^T L^{-1}$. Using LAPACK, the three steps would be performed by the functions: POTRF, TRTRI, and LAUUM, and would result in a synchronization between each stage. Figure 4 shows the effect of using a dynamic, asynchronous, data-driven execution environment, which can automatically and transparently extract a higher degree of parallelism by fusing the three stages. The data-dependencies between the stages are inferred and tasks from all the stages can be interspersed if the dependencies are fulfilled.

4.2. Static scheduling

For a given DAG, a *static scheduling scheme* passes all the computational tasks to the parallel processing units in a specific execution order before the execution of the DAG. The data movement between the processing units is also statically scheduled. Such a scheme can obtain excellent performance when the programmer knows the properties of the computational tasks and of the target hardware architecture. There have also been extensive studies to automatically tune some of the static parameters in order to improve the performance of the static scheduling.

For example, many of the routines in MAGMA follow a *hybrid programming* model where the computational tasks are statically scheduled on the CPU and GPUs (including multiple GPU and non-GPU-resident factorization algorithms [32, 35, 56, 57]). In this model, BLAS-1 or BLAS-2 based tasks, that tend to be on the critical path, are typically scheduled on the CPU, while the BLAS-3 based tasks are scheduled on the GPU (see Figure 2a). In this fashion, we can take advantage of the data parallelism on the GPU, while the CPU is often efficient at executing the latency-bound tasks. In addition, the scheduling can be designed to respect the data locality while maximizing the overlap of executing the independent tasks on the CPU and GPUs (e.g., look-ahead update of the panel).

One drawback to the static scheduling is that since the tasks are scheduled before the execution starts, the scheduling cannot adapt to changes in the execution environments that may occur during the execution of the program (e.g., the clock frequency and network congestion, or the load imbalance due to the fill-ins during a sparse factorization). Also, in order to obtain the high performance, significant knowledge may be required about the characteristics of the computational tasks and underlying hardware (both of which are becoming extremely complex), and the scheduling needs to be tuned for the specific hardware and potentially for the input data.

4.3. Dynamic scheduling

Some of the shortcomings mentioned for the static scheduling can be overcome by dynamic scheduling heuristics. The general mechanisms of using dynamic scheduling are as given in Section 4.1. The benefit is that a lightweight runtime environment with task superscalar concepts can be used to enable the developer to write serial code while providing parallel execution. In this model, the computation must be split into tasks with specified input dependencies and outputs (as in Section 4.1). Knowledge of the algorithm in terms of the best ways to schedule the execution (e.g., for best fit to the underlying hardware components or to minimize communication, etc.) is not needed in runtimes like StarPU, but if available can be passed to the runtime through task priorities or other mechanisms to help the scheduler to improve performance [14, 25, 32, 52].

Thus, dynamic scheduling can ease the parallel programming efforts and sometimes improve performance (*vs.* static scheduling). Moreover, it can be incorporated in a parallel programming model to unify DLA software for heterogeneous systems with components of various strengths. The challenge here is that in order to efficiently use GPU resources, the workload must have a greater degree of parallelism than a workload designed for multicore CPUs, and conceptually, the Intel Xeon Phi coprocessors are capable of handling workloads somewhere in between the two. To address challenges like these, dynamic scheduling plus task abstractions can be used to develop a unified algorithmic parallel programming model for DLA algorithms capable of fully utilizing a wide variety of heterogeneous resources [26], where it is possible to combine NVIDIA and AMD GPUs, multicore CPUs, and Xeon Phi coprocessors in the same system.

Finally, we highlight that while programming models and scheduling are important, it is the algorithmic design that is of highest significance for performance. In particular, the design should account for reducing communication, as time per flop, network bandwidth (between parallel processors), and network latency are all improving, but at exponentially different rates. So an algorithm that is computation-bound and running close to peak today may be communication-bound in the near future. The same holds for communication between levels of the memory hierarchy. To reiterate what we already stated and stress throughout the paper, related to the latter, performance is indeed harder to get on new architectures unless hierarchical communications are applied. Hierarchical communications to get top speed now are needed not only for compute bound operations, e.g., BLAS-3, but also for BLAS-2 [36]. Thus, a programming model should provide abstractions for hierarchical tasks that can block computations over the memory hierarchies in order to reduce data movement. This has been demonstrated in DMAGMA [30] for both shared and large scale distributed memory heterogeneous systems. Figure 2b illustrates the approach: a 2D-block cyclic matrix layout is used; a static scheduling between nodes as in ScaLAPACK; DAG algorithm representation; dynamic scheduling within the node; and hierarchical tasks that can be subdivided or grouped when needed.

4.4. Batched scheduling

The purpose of batched scheduling is to deal with workloads that need to solve a large number of relatively small problems that are independent from each other. The size of each individual problem can be so small (e.g., sub-vector/warp in size) that it often fails to provide sufficient parallelism for modern hardware architectures, especially throughput-oriented accelerators. Such workloads arise in many real applications, including astrophysics [41], quantum chemistry [6], metabolic networks [37], CFD and resulting PDEs through direct and multifrontal solvers [60], high-order FEM schemes for hydrodynamics [15], direct-iterative preconditioned solvers [33], and image [43] and signal processing [5].

The development of a multicore CPU-based solution to such workloads can be implemented using the existing software stack. Each individual problem can fit in cache, where existing optimized routines (e.g., as in MKL [34] or ACML [3]) can run efficiently and achieve high performance. Based on our experience [27], a better approach is to assign one core per problem at a time, rather than launching multithreaded kernels on each problem. The former approach achieves better performance if all cores work concurrently on different independent problems. The scheduling of problems among cores can be static or dynamic (e.g., through an OpenMP `parallel for` clause). The latter scheduling achieves better performance in the generic case of having different problem sizes, as problems are assigned dynamically to cores that become idle during computation, in a fashion similar to Single Queue Multiple Servers configuration.

However, the same approach cannot be used for accelerators like GPUs. The caches in such a category of hardware are too small to host even relatively small problems. For example, the L1/shared memory module in a modern NVIDIA GPU can be configured up to 48KB per Streaming Multiprocessor (SM), which cannot host a square matrix of size larger than 78 in double precision. In addition, existing numerical software for accelerators usually focuses on problems with relatively large sizes. For example, matrix factorization algorithms for GPUs [55] are designed in a hybrid fashion to use CPUs for panel factorizations, which lack enough parallelism, and GPUs for trailing matrix updates, which are embarrassingly parallel (mostly `gemms`). The CPU-GPU communication can be hidden by factorizing a new panel while the update takes place [55]. Unfortunately, such a hybrid approach cannot be used when the matrix is small, since the trailing matrix updates are not big enough to hide the communication of the panel. This is why there is a need to have a new approach for batched workloads.

One of the available solutions can be found in the MAGMA library [2], which uses a fully GPU-based approach. As an example, batched one-sided factorizations are developed using batched BLAS routines. GPU kernels use a 3D grid configuration, where each 2D subgrid is responsible for one problem. The GPU runtime is able to efficiently schedule thread blocks (TBs) across SMs with relatively large batches. In order to achieve high performance on such small sizes, many kernel optimizations are conducted, such as nested recursive blocking, parallel swapping for batched LU factorization, and using streamed `gemm`s for trailing matrix updates [27]. A common target in designing batched kernels is to maximize the number of TBs that can execute concurrently on the same SM, which eventually increases the throughput of solved problems. In order to achieve this target, the design of batched BLAS kernels tends to reduce the resources required per TB, in terms of shared memory, registers, and even number of threads. This is unlike the common approach used in classical BLAS kernels, where TBs often consume a lot of resources to guarantee high performance.

5. Future directions

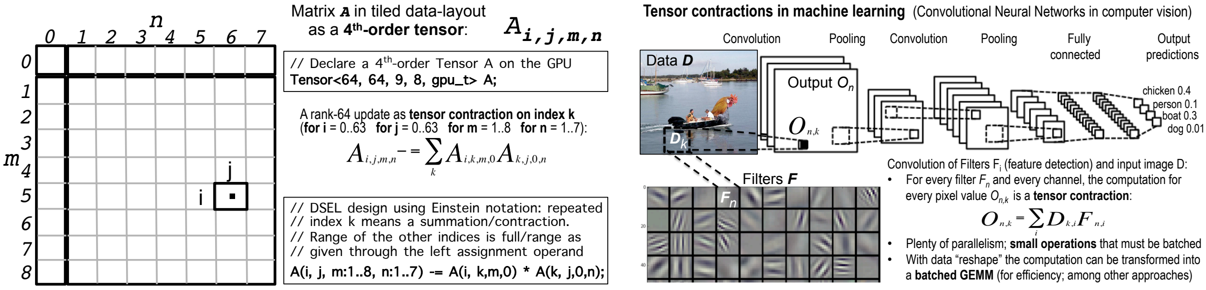
5.1. Latest requirements for LA functionalities in applications

The acceleration of contemporary DLA on heterogeneous architectures, and in particular the acceleration of numerical solvers for large linear systems and eigenvalue problems, is still the main concern in many applications. Examples, according to our recent survey among the Sca/LAPACK, PLASMA, and MAGMA users, are: electronic structure calculations, quantum mechanics and chemistry, continuum mechanics, fluid dynamics (spectral methods), aerodynamics and structures, lattice QCD, weather prediction, computational geophysics for electromagnetic data, convex and non-convex continuous optimization, principal component analysis, PCA calculations for data reduction, linear regression, FEM, hierarchical matrix compression, and various sparse solvers and preconditioners. However, besides these application, around 40% of the responders required LA on many independent problems that are of size O(100) and smaller (and sizes up to O(10) for 20% of the cases). These are applications in machine learning, data analysis, signal processing, batched operations for sparse preconditioners, algebraic multigrid, sparse direct multifrontal solvers, QR types of factorizations on small problems, astrophysics, high-order FEM, and others (see Section 4.4).

Another requirement for DLA numerical libraries is to support more general data formats - not just the standard column/row major dense matrices, or the recently introduced tiled matrices, but also user defined, e.g., suitable to describe physical properties featuring multilinear relations, like tensors [1, 8]. Figure 5, Left illustrates the notion of tensor and tensor contraction in DLA, as well as a tensor contraction design using a Domain Specific Embedded Language (or DSEL). Figure 5, Right illustrates the need for tensor contractions in machine learning. The computational characteristics in this case are common to many applications: the operations of interest are in general small, they must be batched for efficiency, and various transformations may have to be explored to transform the batched small computations into regular – and therefore efficient to implement – operations, e.g., `gemm`s [1]. The efforts to provide solutions for these problems must often be multidisciplinary, incorporating LA, languages, code generations and optimizations, domain science, and application-specific numerical algorithms.

5.2. Weaknesses and strengths of current approaches

The tasking approach and the use of BLAS are here to stay. The current state-of-the-art illustrates that these approaches are prevailing, showing advantages in both performance and ease of development. In spite of their strengths, though, they have weakness too. In particular, a



```
// Declare a 4th-order Tensor A on the GPU
Tensor<64, 64, 9, 8, gpu_t> A;
```

A rank-64 update as **tensor contraction on index k** (**for i** = 0..63 **for j** = 0..63 **for m** = 1..8 **for n** = 1..7):

$$A_{i,j,m,n} -= \sum_k A_{i,k,m,0} A_{k,j,0,n}$$

```
// DSEL design using Einstein notation: repeated
// index k means a summation/contraction.
// Range of the other indices is full/range as
// given through the left assignment operand
A(i, j, m:1..8, n:1..7) -= A(i, k,m,0) * A(k, j,0,n);
```

Convolution of Filters F_i (feature detection) and input image D:

- For every filter F_n and every channel, the computation for every pixel value $O_{n,k}$ is a **tensor contraction**:

$$O_{n,k} = \sum_i D_{k,i} F_{n,i}$$

- Plenty of parallelism; **small operations** that must be batched
- With data "reshape" the computation can be transformed into a **batched GEMM** (for efficiency; among other approaches)

Figure 5. Left: Examples of a 4^{th}-order tensor for tile matrix layout used in DLA, a tensor contraction, and a tensor contraction design using Einstein summation notation through a DSEL. **Right**: Illustration of tensor contractions needed in machine learning

critical weakness is the level of interoperability with other tools and libraries. For example, fusion of DAGs in the tasking model, as in Figure 4, is achievable only within the programing model used for the different DAG components; only at that level does the user/developer of a fused POTRI know the subtasks and their dependencies. If a developer uses third party tools, i.e., does not have access to the subtasks and their dependencies, the fork-join bottleneck between the routines of the libraries used will be encountered. Other weaknesses include: lack of software for small batched problems, and thus lack of support from run-time systems for efficient scheduling of these types of problems; lack of enough flexibility in data structures for data-driven parallel models; deficiencies in mechanisms to deal with heterogeneity in terms of natural enforcement of load balance, especially for distributed systems with different nodes.

5.3. The next generation of parallel programming models for DLA

BLAS, tasking, and scheduling will be indispensable in the next generation of parallel programming models for DLA. Thus, "adapting what we have can (and must) work" [31]. Next generation programming models must address the weaknesses mentioned above. The interoperability issue can be addressed by building models on open standards, e.g., designing runtime systems on top of open standards that support tasking and are competitive with current runtimes. Thus, we advocate the adoption of OpenMP4.5 in the design of parallel programming models for DLA libraries. Load balancing issues can be addressed by better support for hierarchical tasks and data abstractions. Therefore, future directions in MAGMA, for example, include the development of key extensions to OpenACC/OpenMP to access different types of memory hierarchies, the design of data-structure APIs for efficient mapping to extreme-scale heterogeneous systems, and interoperability with other programming models and libraries. Related to new data structures, analysis of current applications of interest – ranging from machine learning to big data analytics – has demonstrated the need to handle higher dimensional data; these are cases where flattening applications to LA on two-dimensional data (matrices) may not be enough. To address these needs, support of tensor data abstractions and batched scheduling in the parallel programming models can be highly beneficial [1].

Conclusions

Support for developing hierarchical algorithms is the main component of current and future parallel programming models for DLA. The evolution in the models follows the expansion of the memory hierarchies in modern architectures. This has been accomplished historically by the use of BLAS. Now BLAS is used to cover the hierarchies on a single core, as well as many cores on shared memory architectures through parallel BLAS implementations. PBLAS has provided an extension of the BLAS model to distributed systems. On complex heterogeneous nodes, BLAS has been combined with runtime systems, where developers split the computation into tasks (e.g., BLAS-type for a single core), and the runtime then maps the tasks for execution over the machine's components. These models lead to fork-join type approaches where the fork is either parallel BLAS, or other parallel LA algorithms provided by a numerical library (e.g., MAGMA). Future directions will improve interoperability between libraries and tools through the use of widely accepted models like CUDA and open standards like the OpenMP4.5, OpenACC, and/or OpenCL. Thus, new developments will be built and will rely on what we have now. Indeed, for distributed memory systems, we have shown that building on traditional approaches, such as the 2D block cyclic distribution from ScaLAPACK and extensions of heterogeneous models for shared memory systems, works. As a word of caution in using static *vs.* dynamic scheduling, we point out that although dynamic scheduling may have certain benefits in terms of ease of implementation and handling heterogeneity, splitting a regular (DLA) computation into many small tasks and trying to handle dependencies among them can introduce overheads. Remarkably, mapping the HPL benchmark on heterogeneous systems is still based on regular, statically scheduled and executed code. Approaches that use the regularity of DLA must be used, where grouping tasks, as well as the ability to hierarchically split them when needed, can be mapped efficiently to the hierarchical memories of current and upcoming heterogeneous systems.

In summary, future directions for the next generation of parallel programming models for high-performance dense linear algebra libraries on heterogeneous systems must include support of hierarchical tasks, runtimes for hierarchical tasks, batched approaches, new data and task abstractions, as well as hybrid BLAS versions based on them.

This material is based upon work supported by the National Science Foundation under Grants No. ACI-1339822, NVIDIA, Intel, AMD, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

1. A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. High-Performance Tensor Contractions for GPUs. Technical Report UT-EECS-16-738, 01-2016 2016.

2. E. Agullo, J. Demmel, 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. *J. Phys.: Conf. Ser.*, 180(1), 2009.

3. ACML - AMD Core Math Library, 2014. Available at `http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml`.

4. E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

5. M. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

6. A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Luc, M. Nooijene, R. Pitzerf, J. Ramanujamg, P. Sadayappanc, and A. Sibiryakovc. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

7. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

8. M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, C. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. Towards a High-Performance Tensor Algebra Package for Accelerators. `http://icl.cs.utk.edu/projectsfiles/magma/pubs/43-smc15_tensor_contractions.pdf`, September 2 2015. Smoky Mountains Computational Sciences and Engineering Conference (SMC'15), Poster, Gatlinburg, TN.

9. R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using smpss. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

10. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

11. J. Choi, J. Demmel, I. S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers Design Issues and Performance. *Computer Physics Communications*, 97, aug 1996.

12. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Second International Workshop, PARA '95, Lyngby, Denmark, August 21-24, 1995, Proc.*, pages 107–114, 1995.

13. M. Corporation. C++ AMP : Language and programming model, 2012. Version 1.0, August.

14. S. Donfack, S. Tomov, and J. Dongarra. Dynamically balanced synchronization-avoiding lu factorization with multicore and gpus. In *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*, 05-2014 2014.

15. T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

16. J. J. Dongara, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, PA, 1979.

17. J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, pages 1–26, 2014.

18. J. Dongarra, J. Kurzak, P. Luszczek, T. Moore, and S. Tomov. Numerical algorithms and libraries at exascale. `http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/`, October 19 2015. HPCwire.

19. J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

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

21. P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, Aug. 2012.

22. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OMPSS: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

23. K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++.* Microsoft Press, 1st edition, 2012. ISBN-13: 978-0735664739 ISBN-10: 0735664730.

24. F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):17, 2012.

25. A. Haidar, C. Cao, I. Yamazaki, J. Dongarra, M. Gates, P. Luszczek, and S. Tomov. Performance and Portability with OpenCL for Throughput-Oriented HPC Workloads Across Accelerators, Coprocessors, and Multicore Processors. In *5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 14)*, New Orleans, LA, 11-2014 2014. IEEE.

26. A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.

27. A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on gpus. *International Journal of High Performance Computing Applications*, 2015.

28. A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. Framework for Batched and GPU-resident Factorization Algorithms to Block Householder Transformations. In *ISC High Performance*, Frankfurt, Germany, 07-2015 2015. Springer, Springer.

29. A. Haidar, J. Dongarra, K. Kabir, M. Gates, P. Luszczek, S. Tomov, and Y. Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 01-2015 2015.

30. A. Haidar, A. YarKhan, C. Cao, P. Luszczek, S. Tomov, and J. Dongarra. Flexible linear algebra development and scheduling with cholesky factorization. In *17th IEEE International Conference on High Performance Computing and Communications*, Newark, NJ, 08-2015 2015.

31. M. A. Heroux. Exascale programming: Adapting what we have can (and must) work. http://www.hpcwire.com/2016/01/14/24151/, January 14 2016. HPCwire.

32. M. Horton, S. Tomov, and J. Dongarra. A class of hybrid LAPACK algorithms for multicore and GPU architectures. In *Proceedings of Symposium for Application Accelerators in High Performance Computing (SAAHPC)*, 2011.

33. E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, Feb. 2004.

34. Intel Math Kernel Library, 2014. Available at http://software.intel.com/intel-mkl/.

35. Y. Jia, P. Luszczek, and J. Dongarra. Multi-GPU implementation of LU factorization. In *proceedings of the international conference on computational science (ICCS)*, 2012.

36. K. Kabir, A. Haidar, S. Tomov, and J. Dongarra. On the Design, Development, and Analysis of Optimized Matrix-Vector Multiplication Routines for Coprocessors. In *ISC High Performance 2015*, Frankfurt, Germany, 07-2015 2015.

37. J. L. Khodayari A., A.R. Zomorrodi and C. Maranas. A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metabolic engineering*, 25C:50–62, 2014.

38. Khronos OpenCL Working Group. The opencl specification, version: 1.0 document revision: 48, 2009.

39. J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

40. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979.

41. O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

42. S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

43. J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.

44. R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, Nov. 2010.

45. C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, J. Dongarra, H. Anzt, M. Gates, A. Haidar, Y. Jia, K. Kabir, I. Yamazaki, and J. Labarta. Heterogeneous streaming. In *IPDPSW, AsHES 2016 (accepted)*, Chicago, IL, USA, May 23 2016.

46. OpenACC Non-Profit Corporation. The OpenACC application programming interface version 2.0. `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`, June 2013.

47. OpenMP Architecture Review Board. OpenMP application program interface version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

48. OpenMP Architecture Review Board. OpenMP application program interface version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

49. OpenMP Architecture Review Board. OpenMP application program interface version 4.5, Nov 2015.

50. J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta. Cellss: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

51. A. Pop and A. Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4), 2013.

52. F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *ICS*, pages 365–376, 2012.

53. M. Tillenius. Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015.

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

55. S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

56. I. Yamazaki, S. Tomov, and J. Dongarra. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. In *proceedings of the international conference on computational science (ICCS)*, 2012.

57. I. Yamazaki, S. Tomov, and J. Dongarra. Non-GPU-resident symmetric indefinite factorization. *Submitted to ACM Transactions on Mathematical Software (TOMS)*, 2016.

58. Y. Yan, B. M. Chapman, and M. Wong. A comparison of heterogeneous and manycore programming models. `http://www.hpcwire.com/2015/03/02/a-comparison-of-heterogeneous-and-manycore-programming-models`, March 2 2015. HPCwire.

59. A. YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures.* PhD thesis, University of Tennessee, 2012.

60. S. N. Yeralan, T. A. Davis, and S. Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.