Unified Development for Mixed Multi-GPU and Multi-Coprocessor Environments using a Lightweight Runtime Environment

Azzam Haidar[1], Chongxiao Cao[1], Asim YarKhan[1], Piotr Luszczek[1], Stanimire Tomov[1], Khairul Kabir[1], and Jack Dongarra[1,2,3]

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

Abstract

Many of the heterogeneous resources available to modern computers are designed for different workloads. 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. This multitude of applicable workloads will likely lead to mixing multicore-CPUs, GPUs, and Intel coprocessors in multi-user environments that must offer adequate computing facilities for a wide range of workloads. In this work, we are using a lightweight runtime environment to manage the resource-specific workload, and to control the dataflow and parallel execution in two-way hybrid systems. The lightweight runtime environment uses task superscalar concepts to enable the developer to write serial code while providing parallel execution. In addition, our task abstractions enable unified algorithmic development across all the heterogeneous resources. We provide performance results for dense linear algebra applications, demonstrating the effectiveness of our approach and full utilization of a wide variety of accelerator hardware.

1 Introduction

With the release of CUDA, the scientific, HPC, and technical computing communities started to enjoy the performance benefits of GPUs. The ever expanding capabilities of the hardware accelerators allowed GPUs to deal with more demanding kinds of workloads and there was very little need to mix different GPUs in the same machine. Intel offering in the realm of hardware acceleration came in the form of a coprocessor called MIC (Many Integrated Cores) now known as Xeon Phi and available under Knights Corner moniker. In terms of capabilities, on the one hand, they are similar to those of GPUs but, on the other hand, there are some slight differences in workloads that could be handled by Phi. We do not aim this paper as comparison between GPUs and the recently introduced MIC coprocessor. Instead, we take a different stand. We believe that users will combine CPUs, GPUs, and coprocessors to leverage strengths of each of them depending on the workload. In a similar fashion, we are showing here how to combine their strengths to arrive at another level of heterogeneity by utilizing all three: CPUs, GPUs, and coprocessors. We call this a multi-way heterogeneity. We present a unified programming model that alleviates the complexity of dealing with multiple software stacks for computing, communication, and software libraries.

2 Background and Related Work

This paper presents research in designing the algorithms and the programing model for high-performance dense linear algebra (DLA) in heterogeneous environments, consisting of a mix of multicore CPUs, GPUs, and Intel Xeon Phi coprocessors (MICs). The mix can contain resources with varying capabilities, e.g., CUDA GPUs from different device generations. While the main goal is to obtain as high fraction of the peak performance as possible for an entire heterogeneous system, a competing secondary goal is to propose a programming model that would simplify the development. To this end, we propose and develop a new lightweight runtime environment, and a number of dense linear algebra routines based on it. We demonstrate the new algorithms, their performance, and the programming model design using the Cholesky and QR factorizations.

2.1 High Performance on Heterogeneous Systems

Efficient use of current computer architectures, namely, running close to their peak performance, can only be achieved for algorithms of high computational intensity, e.g., in DLA, n^2 data requires $O(n^3)$ floating point operations (flops). In contrast, less compute intensive algorithms like the ones using sparse linear algebra can reach only a fraction of the peak performance, e.g., a few Gflop/s for highly optimized SpMV [5] *vs.* over $1,000$ Gflop/s for DGEMM on a Kepler K20c GPU [9]. This highlights the interest in DLA and the importance of designing computational applications that can make efficient use of their hardware resources.

Early results for dense linear algebra were tied to development of high performance BLAS. Volkov el al. [23] developed fast SGEMM for the NVIDIA Tesla GPUs and highly efficient LU, QR, and Cholesky based on that. The fastest DGEMM and factorizations based on it for the NVIDIA Fermi GPUs was developed by Nath et al. [16]. These early BLAS developments were eventually incorporated into the CUBLAS library [9]. The

main DLA algorithms from LAPACK were developed for a single GPU and released through the MAGMA library [15].

More recent work has concentrated on porting additional algorithms to GPUs, and on various optimizations and algorithmic improvements. The central challenge has been to split the computation among the hardware components to efficiently use them, to maintain balanced load, and to reduce idle times. Although there have been developments for a particular accelerator and its multicore host [2, 4, 7, 10, 19, 20], to the best of our knowledge there has been no efforts to create a DLA library on top of a unified framework for multi-way heterogeneous systems.

2.2 Heterogeneous parallel programming models

Both NVIDIA and Intel provide various programming models for their GPUs and coprocessors. In particular, to program and assign work to NVIDIA GPUs and Intel Xeon Phi coprocessors, we use CUDA APIs and heterogeneous offload pragmas/directives, respectively. To unify the parallel scheduling of work between these different architectures we designed a new API, based on a lightweight task superscalar runtime environment, that automates the task scheduling across the devices. In this way, the API can be easily extended to handle other devices that use their own programming models, e.g., OpenCL [7].

Task superscalar runtime environments have become a common approach for effective and efficient execution in multicore environments. This execution mechanism has its roots in dataflow execution, which has a long history dating to the nineteen sixties. It has seen a reemergence in popularity with the advent of multicore processors in order to use the available resources dynamically and efficiently. Many other programming methodologies need to be carefully managed in order to avoid fork-join style parallelism, also called Bulk Synchronous Processing [22], which is increasingly wasteful in a face of ever larger numbers of cores.

In our current research, we build on the QUARK [24, 25] runtime environment to demonstrate the effectiveness of dynamic task superscalar execution in the presence of heterogeneous architectures. QUARK is a superscalar execution environment that has been used with great success for linear algebra software on multicore platforms. The PLASMA linear algebra library has been implemented using QUARK and has demonstrated excellent scalability and high performance [1, 14].

There is a rich area of work on execution environments that begin with serial code and result in parallel execution, often using task superscalar techniques, for example Jade [18], Cilk [6], Sequoia [11], SuperMatrix [8], OmpSS [17], Habanero [4], StarPU [3], or the DepSpawn [12] project.

We chose QUARK as our lightweight runtime environment for this research because it provides flexibility in low level control of task location and binding that would be harder to obtain using other superscalar runtime environments. But, the conceptual work done in this research could be replicated within other projects, so we view QUARK simply as a convenient exemplar of a lightweight, task-superscalar runtime environment.

3 Algorithmic Advancements

In this section, we present the linear algebra aspects of our generic solution for development of either Cholesky, Gauss, and Householder factorizations based on block outer-product updates of the trailing matrix.

Conceptually, one-sided factorization maps a matrix A into a product of matrices X and Y:

$$\mathcal{F} : \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mapsto \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$

Algorithmically, this corresponds to a sequence of in-place transformations of A, whose storage is overwritten with the entries of matrices X and Y (P_{ij} indicates currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow$$
$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & A_{22}^{(1)} & A_{23}^{(1)} \\ X_{31} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & P_{22} & A_{23}^{(1)} \\ X_{31} & P_{32} & A_{33}^{(1)} \end{bmatrix} \rightarrow$$
$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & A_{33}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & X_{22} & Y_{23} \\ X_{31} & X_{32} & P_{33} \end{bmatrix} \rightarrow$$
$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \rightarrow [XY],$$

where XY_{ij} is a compact representation of both X_{ij} and Y_{ij} in the space originally occupied by A_{ij}.

	Cholesky	Householder	Gauss
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK2 xGEMM	xLARFB	xLASWP xTRSM xGEMM

Table 1: Routines for panel factorization and the trailing matrix update.

Observe two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update: $A^{(i)} \rightarrow A^{(i+1)}$. Implementation of these two phases leads to a straightforward iterative scheme shown in Algorithm 1. Table 1 shows BLAS and LAPACK routines that should be substituted for the generic routines named in the algorithm.

Algorithm 1: Two-phase implementation of a one-sided factorization.

```
for P_i ∈ {P_1, P_2, ..., P_n} do
    PanelFactorize(P_i)
    TrailingMatrixUpdate(A^(i))
```

The use of multiple accelerators for the computations complicates the simple loop from Algorithm 1: we have to split the update operation into multiple instances for each of the accelerators. This was done in Algorithm 2. Notice that PanelFactorize() is not split for execution on accelerators because it

Algorithm 2: Two-phase implementation with a split update.

for $P_i \in \{P_1, P_2, \ldots\}$ **do**
  PanelFactorize(P_i)
  TrailingMatrixUpdate$_{\text{Kepler}}$($A^{(i)}$)
  TrailingMatrixUpdate$_{\text{Phi}}$($A^{(i)}$)

is considered a latency-bound workload which faces a number of inefficiencies on throughput-oriented devices. Due to their high performance rate exhibited on the update operation, and the fact that the update requires the majority of floating-point operations, it is the trailing matrix update that is be off-loaded. The problem of keeping track of the computational activities is exacerbated by the separation between the address spaces of main memory of the CPU and the devices. This requires synchronization between memory buffers and is included in the implementation shown in Algorithm 3.

Algorithm 3: Two-phase implementation with a split update and explicit communication.

for $P_i \in \{P_1, P_2, \ldots\}$ **do**
  PanelFactorize(P_i)
  PanelSend$_{\text{Kepler}}$(P_i)
  TrailingMatrixUpdate$_{\text{Kepler}}$($A^{(i)}$)
  PanelSend$_{\text{Phi}}$(P_i)
  TrailingMatrixUpdate$_{\text{Phi}}$($A^{(i)}$)

The code has to be modified further to achieve closer to optimal performance. In fact, the bandwidth between the CPU and the devices is orders of magnitude too slow to sustain computational rates of accelerators[1] The common technique to alleviate this imbalance is to use *lookahead* [21].

Algorithm 4: Lookahead of depth 1 for the two-phase factorization.

PanelFactorize(P_1)
PanelSend(P_1)
TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$(P_1)
PanelStartReceiving(P_2)
TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($R^{(1)}$)
for $P_i \in \{P_2, P_3, \ldots\}$ **do**
  PanelReceive(P_i)
  PanelFactorize(P_i)
  PanelSend(P_i)
  TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$(P_i)
  PanelStartReceiving(P_i)
  TrailingMatrixUpdate$_{\{\text{Kepler,Phi}\}}$($R^{(i)}$)
PanelReceive(P_n)
PanelFactor(P_n)

Algorithm 4 shows a very simple case of lookahead of depth 1. The update operation is split into an update of the next panel, the start of the receiving of the next panel that just got updated, and an update of the rest of the trailing matrix R. The splitting is done to overlap the communication of the panel and the update operation. The complication of this approach comes from the fact that depending on the communication bandwidth and the accelerator speed, a different lookahead depth might be required for optimal overlap. In fact, the adjustment of the depth is often required throughout the factorization's runtime to yield good performance: the updates consume progressively less time when compared to the time spent in the panel factorization.

Since the management of adaptive lookahead is tedious, it is desirable to use a dynamic DAG scheduler to keep track of data dependences and communication events. The only issue is the homogeneity inherent in most of the schedulers which is violated here due to the use of three different computing devices that we used. Also, common scheduling techniques, such as task stealing, are not applicable here due to the disjoint address spaces and the associated large overheads. These caveats are dealt with comprehensively in the remainder of the paper.

4 Lightweight Runtime for Heterogeneous Hybrid Architectures

In this section, we discuss the techniques that we developed in order to achieve an effective and efficient use of heterogeneous hybrid architectures. Our proposed techniques consider both, the higher ratio of execution and the hierarchical memory model of the new and emerging accelerators and coprocessor hardware in order to create a heterogeneous programming model. We also redesign an existing superscalar task execution environment to handle to schedule and execute tasks on multi-way heterogeneous devices. For our experiments, we consider shared-memory multicore machines with some collection of GPUs and MIC devices. Below, we present QUARK and the specific changes incorporated in it to facilitate execution on heterogeneous devices.

4.1 Task Superscalar Scheduling

Task-superscalar execution takes a serial sequence of tasks as input and schedules them for execution in parallel, inferring the data dependences between the tasks at runtime. The dependences between the tasks are inferred through the resolution of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). The dependences are extracted from the serial code by having the user annotate the data when defining the tasks, noting whether the data is to be read and/or written.

The RaW hazard, often referred to as the *true dependency*, is the most common one. It defines the relation between a task writing the data and another task reading that data. The reading task has to wait until the writing task completes. In contrast, if multiple tasks all wish to read the same data, they need not wait on each other but can execute in parallel. Task-superscalar execution results in an asynchronous, data-driven execution that may be represented by a *Direct Acyclic Graph* (DAG), where the tasks are the nodes in the graph and the edges correspond to data movement between the tasks. Task-superscalar execution is a powerful tool for productivity. Since serial code is the input

[1] The bandwidth for current generation PCI Express is at most 16 GB/s and the devices achieve over 1000 Gflop/s performance.

for the runtime system, and the parallel execution avoids all the data hazards, the correctness of the serial code guarantees parallel correctness.

Using task superscalar execution, the runtime can achieve parallelism by executing tasks with non-conflicting data dependences (e.g., simultaneous reads of data by multiple tasks). Superscalar execution also enables lookahead in the serial code, since tasks from further ahead in the serial presentation of tasks can be executed as soon as their dependences are satisfied.

Lightweight Runtime Environment QUARK (QUeuing and Runtime for Kernels) is the lightweight runtime environment chosen for this research, since it provides an API that allows low level task placement. QUARK is a data-driven dynamic superscalar runtime environment with a simple API for serial task insertion. It is the dynamic runtime engine used within the PLASMA linear algebra library and has been shown to provide high productivity and performance benefits [14, 24, 25].

5 Efficient and Scalable Programming Model Across Multiple Devices

In this section, we discuss the programming model that raises the level of abstraction above the hardware and its accompanying software stack to offer a uniform approach for algorithmic development. We describe the techniques that we developed in order to achieve an effective use of multi-way heterogeneous devices. Our proposed techniques consider both, the higher ratio of execution and the hierarchical memory model of the new emerging accelerators and coprocessors.

5.1 Supporting Heterogeneous Platforms

GPU accelerators and coprocessors have a very high computational peak compared to CPUs. For simplicity, we refer to both GPUs and coprocessors as accelerators. Also, different types of accelerators have different capabilities, which makes it challenging to develop an algorithm that can achieve high performance and reach good scalability. From the hardware point of view, an accelerator communicates with the CPU using I/O commands and DMA memory transfers, whereas from the software standpoint, the accelerator is a platform presented through a programming interface. The key features of our model are the processing unit capability (CPUs, GPUs, Xeon Phi), the memory access, and the communication cost. As with CPUs, the access time to the device memory for accelerators is slow compared to peak performance. CPUs try to improve the effect of the long memory latency and bandwidth by using hierarchical caches. This does not solve the slow memory problem completely but is often effective. On the other hand, accelerators use multithreading operations that access large data sets that would overflow the size of most caches. The idea is that when the accelerator's thread unit issues an access to the device memory, that thread unit stalls until the memory returns a value. In the meantime, the accelerator's scheduler switches to another hardware thread, and continues executing that thread. In this way, the accelerator exploits program parallelism to keep functional units busy while the memory fulfills past requests. By comparison with CPUs, the device memory delivers higher absolute bandwidth (around 180 GB/s for Xeon Phi and 160 GB/s for Kepler K20c). To side-step memory issues, we develop a strategy that prioritizes the data-intensive operations to be executed by the accelerator and keep the memory-bound ones for the CPUs since the hierarchical caches with out-of-order superscalar scheduling are more appropriate to handle it. Moreover, in order to keep the accelerator busy, we redesign the kernels and propose dynamically guided data distribution to exploit enough parallelism to keep the accelerators and processors busy.

Algorithm 5: Cholesky implementation for multiple devices.

```
Task_Flags panel_flags = Task_Flags_Initializer
Task_Flag_Set(&panel_flags, PRIORITY, 10000)
memory-bound → locked to CPU
Task_Flag_Set(&panel_flags, BLAS2, 0)
for k ∈ {0, nb, 2 × nb, ..., n} do
    Factorization of the panel dA(k:n,k)
    Cholesky on the tile dA(k,k)
    TRSM on the remaining of the panel dA(k+nb:n,k)
    DO THE UPDATE: SYRK task has been split into a set of parallel compute intensive GEMM to increase parallelism and enhance the performance. Note that the first GEMM consists of the update of the next panel, thus the scheduler check the dependency and once finished it can start the panel factorisation of the next loop on the CPU.
    if panel_m > panel_n then
        SYRK with trailing matrix
        for j ∈ {k + nb, k + 2nb, ..., n} do
            GEMM dA(j:n,k) × dA(j,k)^T = dA(j:n,j)
```

From a programming model point of view, we probably cannot hide the distinction between the two levels of parallelism. For that, we convert each algorithm into a host part and an accelerator part. Each routine to run on the accelerator must be extracted into a separate hardware specific kernel function. The kernel itself may have to be carefully optimized for the accelerator, including unrolling loops, replacing some memory-bound operations by compute-intensive ones even if it has a marginal extra cost, and also arranging its tasks to use the device memory efficiently. The host code must manage the device memory allocation, the CPU-device data movement, and the kernel invocation. We redesigned our QUARK runtime engine in order to present a much easier programming environment and to simplify scheduling. This often allows us to maintain a single source version that handles different types of accelerators either independently or when mixed together. Our intention is that our model simplifies most of the hardware details, but gives us finer levels of control. Algorithm 5 shows the pseudocode for the Cholesky factorization as an algorithm designer views it. It consists of a sequential code that is simple to comprehend and independent of the architecture. Each of these calls represents a task that is inserted into the scheduler, which stores it to be executed when all of its dependencies are satisfied. Each task

by itself consists of a call to a kernel function that could either be a CPU or an accelerator function. We tried to hide the differences between hardware and letting the QUARK engine handle the transfer of data automatically. In addition, we developed low-level optimizations for the accelerators, in order to accommodate hardware- and library-specific tuning and requirements. Moreover, we implemented a set of directives that are evaluated at runtime in order to fully map the algorithm to the hardware and run close to the peak performance of the system. Using these strategies, we can more easily develop simple and portable code, that can run on different heterogeneous architecture letting the scheduling and execution engine do much of the tedious bookkeeping.

In the discussion below, we will describe in detail the optimization techniques we propose, and explore some of the features of our model and also describe some directives that help with tuning performance in an easy fashion. The study here is described in the context of the Cholesky factorization but it can be easily applied to other algorithms such as the QR decomposition and LU factorization.

5.2 Resource Capability Weighing (CW)

Since there is no simple way to express the difference in the workload-capabilities between the CPUs and accelerators. Clearly, we cannot balance the load, if we treat them as peers and assign them equivalent amount of work this naïve strategy would cause the accelerator to be substantially idle. As described above, in our model we propose to assign the latency-bound operations to the CPUs and the compute-intensive ones to accelerators. In order to support multi-way heterogeneous hardware, QUARK was extended with a mechanism for distributing tasks based on the individual capabilities of each device. For each device i and each kernel type k, QUARK maintains an α_{ik} parameter which corresponds to the effective performance rate that can be achieved on that device. In the context of linear algebra algorithms, this means that we need an estimation of performance for Level 1, 2, and 3 BLAS operations. This can be done either by the developer during the implementation where the user gives a directive to QUARK that this kernel is either bandwidth-bound or compute-bound function (as shown in Algorithm 5 with a call to Task_Flag_Set with BLAS2 argument) or estimated according to the volume of data and the elapsed time of a kernel by the QUARK engine at runtime.

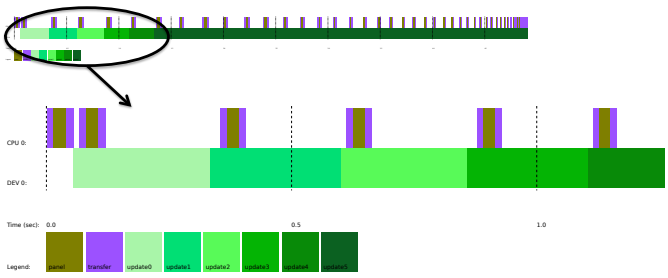

Figure 1: A trace of the Cholesky factorization on a multicore CPU with a single GPU K20c, assigning the panel factorization task to the CPU (brown task) and the update to the GPU (green task) for a matrix of size 20,000.

Figure 1 shows the execution trace of the Cholesky factorization on a single multicore CPU and a K20c GPU of system A. We see that the memory-bound kernel (e.g., the panel factorization for the Cholesky algorithm) has been allocated to the CPU while the compute-bound kernel (e.g., the update performed by DSYRK) has been allocated to the accelerator. The initial data is assumed to be on the device, and when the CPU is executing a task, they need to be copied from the device and sent back to be used for updating the trailing matrix. The data transfer is represented by the *purple* color in the trace. The CPU panel computation is represented by the *gold* color. The trailing matrix update are depicted in *green*. For clarity, we varied the intensity of the green color representing the update from light to dark for the first 5 steps of the algorithm. From this trace, we can see that the GPU is kept busy all the way until the end of execution. The use of the lookahead technique described in Algorithm 4, does not require any extra effort since it is handled by the QUARK engine through the dependencies analysis. The engine will ensure that the next panel (panel of step $k+1$) is updated as soon as possible by the GPU in order to be sent to the CPU to be factorized while the GPU is continuing the update of the trailing matrix of step k. Also, the QUARK engine manages the data transfer to and from the CPU automatically. The advantage of such strategy is not only to hide the data transfer cost between the CPU and GPU (since it is overlapped with the GPU computation), but also to keep the GPU's CUDA streams busy by providing enough tasks to execute. This is highlighted in Figure 1, where we can see that the panel of step 1 is quickly updated by the GPU and sent to the CPU to be factorized and sent back to the GPU, which is a perquisite to perform the trailing matrix update of step 1, before the GPU has already finished the update of trailing matrix of step 0, and so on.

However, it is clear that we can improve this further by fully utilizing all the available resources, particularly exploiting the idle time of the CPUs (white space in the trace). Based on the parameters defined above, we can compute a resource capability-weights for each task that reflects the cost of executing it on a specific device. This cost is based on the communication cost (if the data has to be moved) and on the type of computation (memory-bound or compute-bound) performed by the task. For a task that requires an $n \times n$ data, we define its computation type to be from one of the Levels of BLAS (either 1, 2, or 3). Thus the two factors are fairly simply defined as:

$$\text{communication} = \frac{n \times n}{\text{bandwidth}}$$

$$\text{computation} = n^k \times \alpha_{ik} \qquad \text{where } k \text{ is Level } k \text{ BLAS}$$

The capability-weights for a task is then the ratio of the total cost of the task on one resource versus another resource. For example, the capability-weights for the update operation (a Level 3 BLAS) from the execution shown in Figure 1 is around $1:10$ which means that the GPU can execute 10 times as many update tasks as CPU.

Greedy Scheduling using Capability-Weights: As each task is inserted into the runtime, it is assigned to the resource with the largest remaining capability-weights. This greedy heuristic takes into account the capability-weights of the re-

source as well as the current number of waiting tasks *preferred* to be executed by this resource. For example, for the CPU, the panel tasks are memory-bound and thus are preferred to be executed always on the CPU side. The heuristic tries to maintain the ratios of the capability-weights across all the resources.

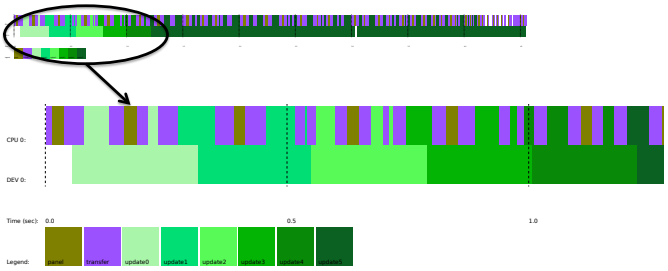

Figure 2: A trace of the Cholesky factorization on 16-core, 2-socket Sandy Bridge CPU and a K20c GPU, using capability-weights to distribute tasks.

In Figure 2, we can see the effect of using capability-weights to assign a subset of the update tasks to the CPU. The GPU remains as busy as before, but now the CPU can contribute to the computation and does not have as much idle time as before. Careful management of the capability-weights ensures that the CPU does not take any work that would cause a delay to the GPU, since that would negatively affect the performance. We also plot in Figure 3 the performance gain obtained when using this technique. The graph shows that on a 16-core Sandy Bridge CPU, we can achieve a gain of around 100 Gflop/s (red curve) and 80 Gflop/s (blue curve) when enabling this technique when using a single Fermi M2090 and Kepler K20c GPU on system A and B, respectively (the hardware is described in §6.1).

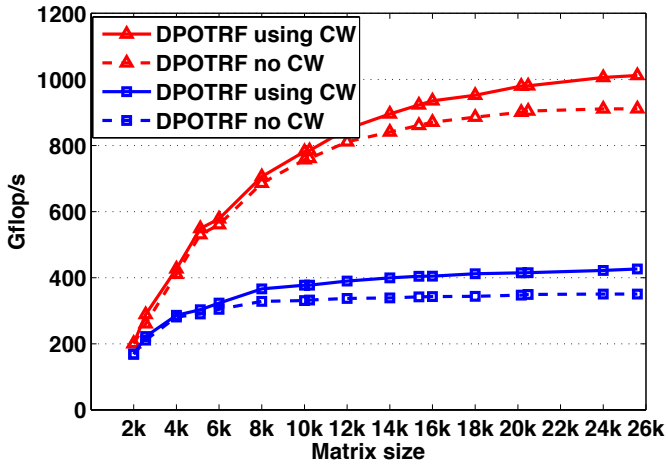

Figure 3: Performance comparison of the Cholesky factorization when using the capability-weights (CW) to distribute tasks among the heterogeneous hardware, on a node of 16 Sandy-Bridge CPU and either a Kepler (K20c) of system A "red curve" or a Fermi(M2090) of system B " blue curve".

Improved Task Priorities: In order to highlight the importance of task priority, we recall, that the panel factorization task of most of the one-sided factorization (e.g., the Cholesky, QR and LU algorithms) is on the critical path of execution. In other words, only if a panel computation is done in its entirety, its corresponding update computation (compute-bound operation) can proceed. In the traces in Figures 1 and 2, it can be observed that the panel factorization on the CPU occurs at regular intervals (e.g., the lookahead depth is one). By changing the priority of the panel factorization tasks (using QUARK's task priority flags as mentioned in Algorithm 5), the panel factorization can be executed earlier. This will increase the lookahead depth that the algorithm exposes, increasing parallelism so that there are more update tasks available to be executed by the device resources. Using priorities to improve lookahead results in approximately 5% improvement in the overall performance of the factorization. Figure 4 shows the update tasks being executed earlier in the trace.

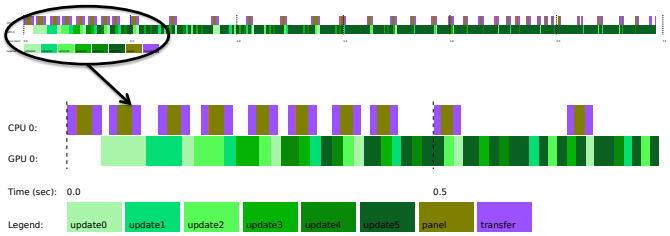

Figure 4: Trace of the Cholesky factorization on multicore 16 Sandy Bridge CPU and a K20c GPU, using priorities to improve lookahead.

Data layout: When we proceed to the multiple accelerator setup, the data is initially distributed over all the accelerators in a 1-D block-column cyclic fashion, with an approximately equal number of columns assigned to each. Note that the data is allocated on each device as one contiguous memory block with the data being distributed as columns within the contiguous memory segment. This contiguous data layout allows large update operations to take place over a number of columns via a single Level 3 BLAS operation, which is far more efficient than having multiple calls with block columns.

Hardware-Guide Data Distribution (HGDD): The experiments showed that the standard 1-D block cyclic data layout was hindering performance in heterogeneous multi-accelerator environments. Figure 5 shows the trace of the Cholesky factorization for a matrix of size 30,000 on System D (consisting of a Kepler K20c, a Xeon Phi (MIC) and Kepler K20-beta that has half the K20c performance). The trace shows that the execution flow is bound by the performance of the slowest machine (the beta K20, second row) and thus we expect lower performance on this machine.

We propose to re-adjust the data layout distribution to be hardware-guided by the use of the capability-weights. Using the QUARK runtime, the data is either distributed or redistributed in an automatic fashion so that each device gets the appropriate volume of data to match its capabilities. So, for example, for System D, using capability weights of K20c:MIC:K20-beta of 10:8:5 would result in a cyclic distribution of 10 columns of data being assigned to the K20c, for each 8 columns assigned to the MIC, and each 5 columns assigned to the K20beta. The superscalar execution environment can do this capability-weighted

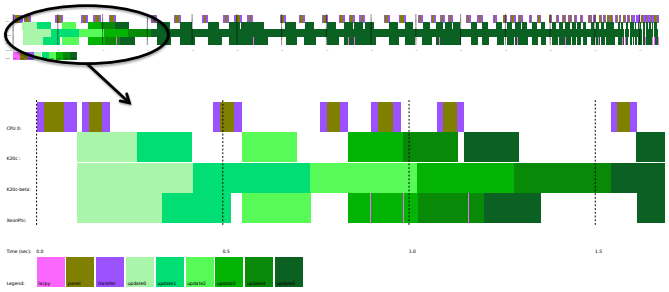

Figure 5: Cholesky factorization trace on multicore CPU and multiple accelerators (a Kepler K20c, a Xeon Phi, and an old K20-beta-release), using 1D block cyclic data distribution without enabling heterogeneous hardware-guided data distribution.

data assignment at runtime.

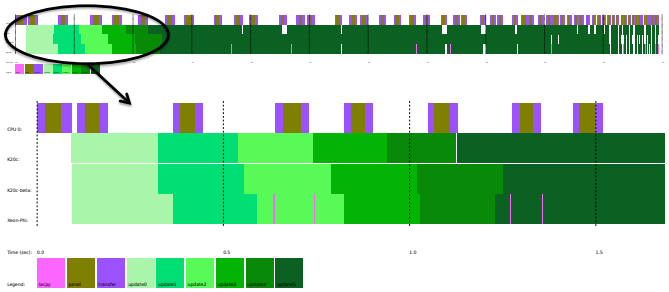

Figure 6: Cholesky factorization trace on multicore CPU and multiple accelerators (a Kepler K20c, a Xeon Phi, and an old K20-beta-release), using the heterogeneous hardware-guided data distribution techniques (HGDD) to achieve higher hardware usage.

Figure 6 shows the trace of the Cholesky factorization for the same example as above (a matrix of size $30K$ a node of the system D) when using the hardware-guided data distribution (HGDD) strategy. It is clear that the execution trace is more compact meaning that all the heterogeneous hardware are fully loaded by work and thus one can expect an increase in the total performance. For that we represent in Figure 7 and Figure 8 the performance comparison of the Cholesky factorization and the QR decomposition when using the HGDD strategy. The curves in blue shows the performance obtained for a one K20c and one XeonPhi experiments. The dashed line correspond to the standard 1-D block-column cyclic distribution while the continuous line illustrate the HGDD strategy. We observe that we can reach an improvement of about 200-300 Gflop/s when using the HGDD technique. Moreover, when we add one more heterogeneous device (the K20beta), here it comes to the complicated hardware situation, we can notice that the standard distribution do not exhibit any speedup. The dashed red curve that represents the performance of the Cholesky factorization using the standard data distribution on the three devices of system D behaves closely and less efficient that the one obtained with the same standard distribution on two devices (dashed blue curve). This was expected, since adding one more device with lower capability may decrease the performance as it may slowdown the fast device. The blue and red curves in Figure 7 illustrate that the HGDD technique exhibits a very good scalability for both algorithms. The graph shows that the performance of the algorithm is not affected by the heterogeneity of the machine, our proposed implementation is appropriate to maintain a high usage of all the available hardware.

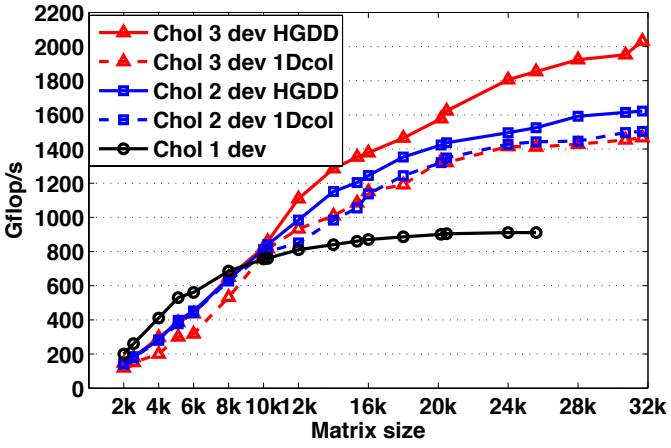

Figure 7: Performance comparison of the Cholesky factorization when using the hardware-guided data distribution techniques versus a 1-D block-column cyclic, on heterogeneous accelerators consisting of a Kepler K20c (1dev), a Xeon Phi (2dev), and an old K20-beta-release (3dev).

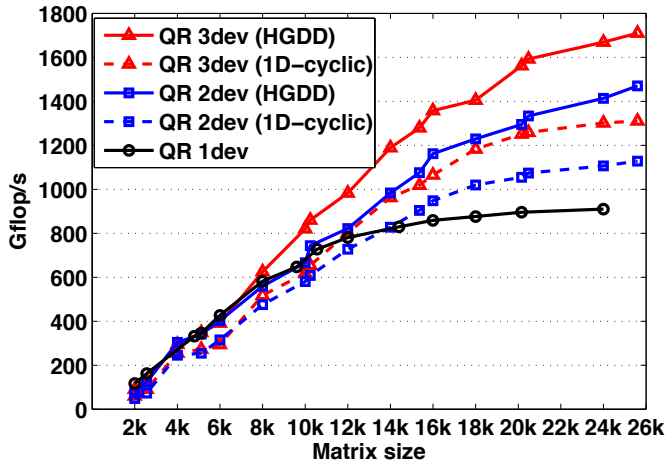

Figure 8: QR performance comparison for hardware-guided data distribution techniques *vs.* 1-D block-column cyclic, on heterogeneous accelerators consisting of a Kepler K20c (1dev), a Xeon Phi (2dev), and an old K20-beta-release (3dev).

5.3 Hardware-Specific Optimizations

One of the main enablers of good performance is optimizing the data communication between the host and accelerator. Another one is to redesign the kernels to exploit the inherent parallelism of the accelerator even by adding extra computational cost.

In this section, we describe the development of our heterogeneous multi-device kernels, which includes the constraints to consider, the methods to deal with communication, and ways of achieving good scalability on both CPUs and accelerators. Our target algorithms, in this study are the one sided-factorization

(Cholesky, QR and LU).

Redesigning BLAS kernels to exploit parallelism and minimize memory movement: The Hermitian rank-k update (SYRK) required by the Cholesky factorization implements the operation $A^{(k)} = A^{(k)} - PP^*$, where A is an n by n Hermitian trailing matrix of step k, and P is the result of the panel factorization done by the CPU. After distribution, the portion of A on each accelerator no longer appears as a symmetric matrix, but instead has a ragged structure shown in Figure 9.

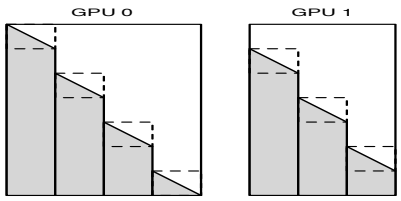

Figure 9: Block-cyclic data distribution. Shaded areas contain valid data. Dashed lines indicate diagonal blocks.

Because of this uneven storage, the multi-device SYRK cannot be assembled purely from regular SYRK calls on each device. Instead, each block column must be processed individually. The diagonal blocks require special attention. In the BLAS standard, elements above the diagonal are not accessed; the user is free to store unrelated data there and the BLAS library will not alter it. To achieve this, one can use a SYRK to update each diagonal block, and a GEMM to update the remainder of each block column below the diagonal block. However, these small SYRK operations have little parallelism and so are inefficient on an accelerator. This can be improved to some degree by using either multiple streams (GPU) or a pragma (MIC) to execute several SYRK updates simultaneously. However, because we have copied the data to the device, we can consider the space above the diagonal to be a scratch workspace. Thus, we update the entire block column, including the diagonal block, writing extra data into the upper triangle of the diagonal block, which is subsequently ignored. We do extra computation for the diagonal block, but gain efficiency overall by launching fewer BLAS kernels on the device and using the more efficient GEMM kernels, instead of small SYRK kernels, resulting in overall 5-10% improvement in performance.

Improving Coalesced Data Access. The LU factorization uses the LASWP routine to swap two rows of the matrix. However, this simple data copy operation might drop the performance of such routines on accelerator architecture since it is recommended that a set of threads read coalescent data from the memory which is not the case for a row of the matrix. Such routine needs to be redesigned in order to overcome this issue. The device data on the GPU is transposed using a specialized GPU kernel, and is always stored in a transposed form, to allow coalesced read/write when the LASWP function is involved. We note that the transpose does not affect any of the other kernel (GEMM, TRSM) required by the LU factorization. The coalesced read/write improves the performance of the LASWP function 1.6 times.

Enabling Specific Architecture Kernels The size of the main Level 3 BLAS kernel that has to be executed on the devices is yet another critical parameter to tune. Every architecture has its own set of input problem sizes that achieve higher than average performance. In the context of one-sided algorithms, all the Level 3 BLAS operations depend on the size of the block panel. On the one hand, a small panel size lets the CPUs finish early but leads to lower Level 3 BLAS performance on the accelerator. On the other hand, a large panel size burdens the CPU and the CPU computation is too slow to be fully overlapped with the accelerator work. The panel size corresponds to a trade-off between the degree of parallelism and the amount of data reuse. In our model, we can easily tune this parameter or allow the runtime to autotune it by varying the panel size throughout the factorization.

Trading Extra Computation for Higher Performance Rate: The implementation that is discussed here is more related to the hardware architecture based on hierarchical memory. The LARFB routine used by the QR decomposition consists of two GEMM and one TRMM operation. Since accelerators are better at handling compute-bound tasks, for computational efficiency, we replace the TRMM by GEMM, thus achieving 5-10% higher performance when executing these kernels on the accelerator.

6 Experimental Setup and Results

6.1 Hardware Description and Setup

Our experiments were performed on a number of shared-memory systems available to us at the time of writing of this paper. They are representative of a vast class of servers and workstations commonly used for computationally intensive workloads. We conducted our experiments on four different systems all of each equipped with an Intel multicore system with dual-socket, 8-core Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket had 24 MiB of shared L3 cache, and each core had a private 256 KiB L2 and 64 KB L1 cache. The system is equipped with 52 GB of memory and the theoretical peak in double precision is 20.8 Gflop/s per core.

- System A is also equipped with six NVIDIA K20c cards with 5.1 GB per card running at 705 MHz, connected to the host via two PCIe I/O hubs at 6 GB/s bandwidth.
- System B is equipped with three NVIDIA M2090 cards with 5.3 GB per card running at 1.3 GHz, connected via two PCIe I/O hubs at 6 GB/s bandwidth.
- System C is also equipped with three Intel Xeon Phi cards with 15.8 GB per card running at 1.23 GHz, and achieving a double precision theoretical peak of 1180 Gflop/s, connected via four PCIe I/O hubs at 6 GB/s bandwidth.
- System D is a heterogeneous system equipped with a K20c, and a Intel Xeon Phi card as the ones described above, and also an old K20c beta release 3.8 GB running at 600 MHz. All are connected via four PCIe I/O hubs at 6 GB/s bandwidth.

A number of software packages were used for the experiments. On the CPU side, we used the MKL (Math Kernel Library) [13]. On the Xeon Phi side, we used the MPSS 2.1.5889-16 as the

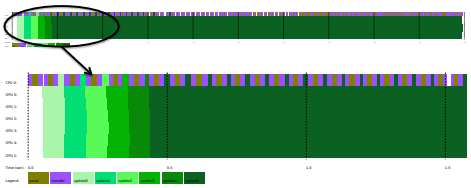

Figure 10: Trace of Cholesky factorization on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

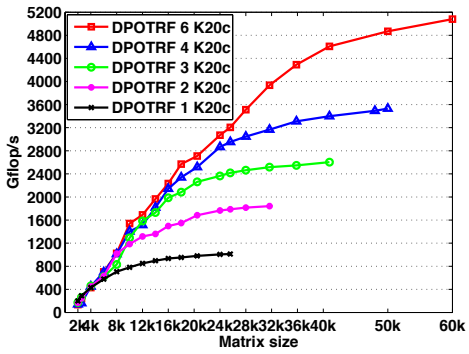

Figure 11: Performance scalability of Cholesky factorization on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

software stack, icc 13.1.1 20130313 which comes with the Composer XE 2013.4.183 suite as the compiler, and finally on the GPU accelerator we used CUDA version 5.0.35.

6.2 Mixed MIC and GPU Results

Getting good performance across multiple accelerators remains a challenging problem that we address with the algorithmic and programming techniques described in this paper. The efficient strategies used to schedule and exploit parallelism across multi-way heterogeneous platforms will be highlighted in this subsection through the extensive set of experiments that we performed on the four systems that we had access to.

Figure 10 show a snapshot of the execution trace of the Cholesky factorization on System A for a matrix of size 40K using six GPUs K20c. As expected the pattern of the trace looks compressed which means that our implementation is able to schedule and balance the tasks on the whole six GPUs devices.

Figures 11 and 12 show the performance scalability of the

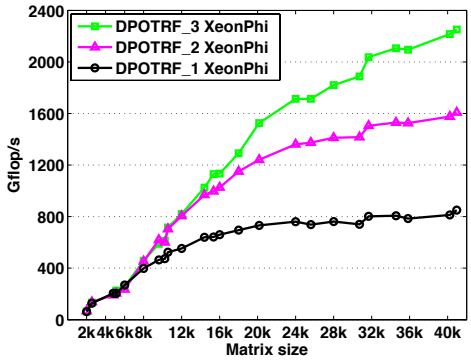

Figure 12: Performance scalability of Cholesky factorization on multicore CPU and multiple accelerators (up to 3 XeonPhi).

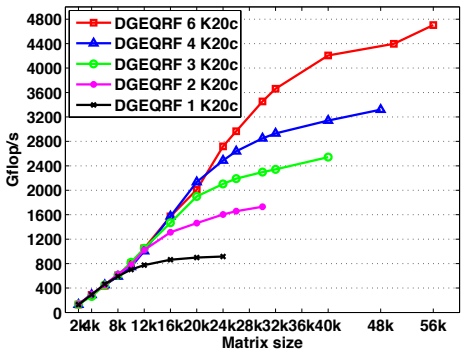

Figure 13: Performance scalability of the QR decomposition on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

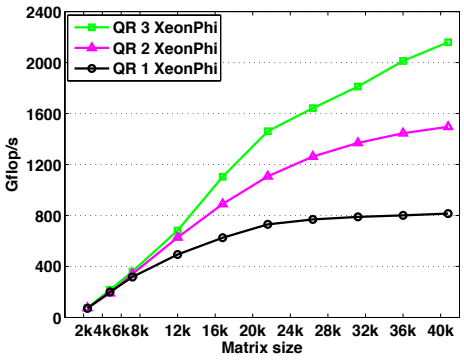

Figure 14: Performance scalability of the QR decomposition on multicore CPU and multiple accelerators (up to 3 Xeon Phi).

Cholesky factorization in double precision on either the 6 GPUs of System A or the 3 Xeon Phi of System C. The curves show performance in terms of Gflop/s. We note that this also reflects the elapsed time, e.g., a performance that is two times higher, corresponds to an elapsed time that is two times shorter. Our heterogeneous multi-device implementation shows very good scalability. On System A, for a 60,000 matrix, the Cholesky factorization achieves 5.1 Tflop/s when using the 6 Kepler K20c GPUs. We observe similar performance trends when using System C. For a matrix of size 40,000, the Cholesky factorization reaches up to 2.3 Tflop/s when using the 3 Intel Xeon Phi coprocessors. Figure 13 depicts the performance scalability of the QR factorization on System A and Figure 14 shows also the obtained results on System C. For a matrix of size 56,000, the QR factorization reaches around 4.7 Tflop/s on the System A using the 6 Kepler K20c GPUs and 2.2 Tflop/s on the System C using the 3 Intel Xeon Phi coprocessors.

7 Conclusions and Future Work

We designed algorithms and a programing model for developing high-performance dense linear algebra in multi-way heterogeneous environments. In particular, we presented best practices and methodologies from the development of high-performance DLA for accelerators. We also showed how judicious modifications to task superscalar scheduling were used to ensure that we meet two competing goals: (1) to obtain high fraction of the peak performance for the entire heterogeneous system, (2) employ a programming model that would simplify the develop-

ment. We presented initial implementations of two algorithms. Future work will include merging MAGMA's CUDA, OpenCL, and Intel Xeon Phi development branches into a single library using the new programming model.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant OCI-1032815 and Subcontract RA241-G1 on NSF Prime Grant OCI- 0910735, DOE under Grants DE-SC0004983 and DE-SC0010042, NVIDIA and Intel.

References

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users Guide. Technical report, ICL, University of Tennessee, 2010.

[2] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 271–276, New York, NY, USA, 2012. ACM.

[3] 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.

[4] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[5] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[7] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clMAGMA: High Performance Dense Linear Algebra with OpenCL. In *International Workshop on OpenCL, IWOCL 2013*, Atlanta, Georgia, USA, May 13-14 2013.

[8] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM.

[9] NVIDIA CUBLAS library. https://developer.nvidia.com/cublas.

[10] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov. Portable HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi. In *10th International Conference on Parallel Processing and Applied Mathematics, PPAM 2013*, Warsaw, Poland, September 8-11 2013.

[11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[12] C. H. González and B. B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475 – 489, 2013. Novel On-Chip Parallel Architectures and Software Support.

[13] Intel. Math Kernel Library. http://software.intel.com/intel-mkl/.

[14] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra. Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*, Computer and Information Science Series. Chapman and Hall/CRC, April 26 2013.

[15] MAGMA library. http://icl.cs.utk.edu/magma/.

[16] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Perf. Comput. Applic.*, 24(4):511–515, 2010. DOI: 10.1177/1094342010385729.

[17] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151. IEEE, 2008.

[18] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993. DOI: 10.1109/2.214440.

[19] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[20] F. Song, S. Tomov, and J. Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and multi-GPU Systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

[21] P. E. Strazdins. Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In *10th International Conference on Parallel and Distributed Computing and Systems, IASTED*, Las Vegas, USA, 1998.

[22] L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.

[23] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC'08*, Austin, TX, November 15-21 2008. IEEE Press. DOI: 10.1145/1413370.1413402.

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

[25] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.