Weighted Dynamic Scheduling with Many Parallelism Grains for Offloading of Numerical Workloads to Multiple Varied Accelerators

Azzam Haidar
Yulu Jia
Piotr Luszczek
Stanimire Tomov
Asim YarKhan
University of Tennessee
haidar@eecs.utk.edu
yjia@vols.utk.edu
tomov@eecs.utk.edu
yarkhan@eecs.utk.edu

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
University of Manchester
dongarra@eecs.utk.edu

ABSTRACT

A wide variety of heterogeneous compute resources are available to modern computers, including multiple sockets containing multicore CPUs, one-or-more GPUs of varying power, and coprocessors such as the Intel Xeon Phi. The challenge faced by domain scientists is how to efficiently and productively use these varied resources. For example, in order to use GPUs effectively, the workload must have a greater degree of parallelism than a workload designed for a multicore-CPU. The domain scientist would have to design and schedule an application in multiple degrees of parallelism and task grain sizes in order to obtain efficient performance from the resources. We propose a productive programming model starting from serial code, which achieves parallelism and scalability by using a task-superscalar runtime environment to adapt the computation to the available resources. The adaptation is done at multiple points, including multi-level data partitioning, adaptive task grain sizes, and dynamic task scheduling. The effectiveness of this approach for utilizing multi-way heterogeneous hardware resources is demonstrated by implementing dense linear algebra applications.

CCS Concepts

•General and reference → Performance; •Computer systems organization → Heterogeneous (hybrid) systems; •Software and its engineering → Data flow architectures;

Keywords

Dataflow scheduling; Multi-grain parallelism; Hardware accelerators

1. INTRODUCTION

ScalA15 Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4011-3.
DOI: 10.1145/2832080.2832085

Hardware accelerator systems have been gaining ground not just in terms of computational performance but also in the ease of use with the introduction of software technologies such as CUDA, OpenCL, and, recently, OpenACC. A similar technology offered by Intel are coprocessors, currently known as the Xeon Phi. The computational edge of the latter has also been established and their dominant software technology, OpenMP 4, has served the community well in terms of productivity to render auxiliary the burden of using lower level of the Intel coprocessor software stack: SCI and COI. We refer to this match of hardware with its software stack as the vertical integration of a platform.

Interestingly, horizontal integration, or combining different platforms and their software stacks is much less supported or researched. Yet, modern computing systems are likely to feature multiple accelerators, often not the same kind, in order to accommodate the whole variety of scientific workloads.

Finally, the constant technological progress of hardware accelerators, now featuring nearly 10 billion transistors, foreshadowed the advancement in traditional multicore CPU technology. A CPU can now produce half a Tera-flop of computations per second in double precision on a single motherboard socket. This CPU performance was made possible by the fast paced improvement in CPU vectorization standards with the flagship AVX and multi-argument fused multiply-add instructions coupled with multiple floating units per core that (with proper use of Level 1 cache) can sustain very high performance without being burdened by the vagaries of modern memory systems and NUMA overheads.

We are presenting a programming model for the rapid development of linear algebra applications designed to be efficient on complex heterogeneous hardware. Our programming model derives its productivity from a design that is based on the task-superscalar paradigm, allowing us to write serial-like code. Then, a task-superscalar runtime executes the code adaptively, scalably, and efficiently on the available heterogeneous resources. In this section, we give background for the linear algebra problem, describe approaches to heterogeneous programming, and discuss alternatives for the runtime environment.

2. BACKGROUND AND MOTIVATION

The desire for faster computers and greater immersive experiences has led hardware designers to push beyond the constraints of Moore's Law. Machines are designed with increasing numbers of cores and specialized accelerators that provide enormous performance boosts for the right kind of computation. In response to this, computer scientists are designing algorithms, software and frameworks that can take advantage of these highly parallel resources. Algorithms need to be restructured to allow greater levels of asynchronous parallelism and workloads need to be tailored to match the available hardware, where the hardware can include multicore CPUs, GPUs, and other accelerators or coprocessors.

Dense linear algebra libraries are at the core of a substantial number of scientific codes across science domains. The traditional approach to extracting high linear algebra performance from the hardware resources is by relying on highly tuned, platform-specific optimized libraries, such as the Basic Linear Algebra Subroutines (BLAS). The scientific codes are then constructed as a sequence of calls to an optimized library. However, this can result in a BSP [20, 19] (also called fork-join) style execution where highly tuned parallel code is interleaved with code that achieves very low parallelism. Newer alternatives to extracting parallelism from available hardware have been advocating a higher level approach [6]. This approach is a task-based dataflow approach where precedence-constrained tasks are executed asynchronously and in parallel. This approach can provide a much higher occupancy of the computational resources.

Developing linear algebra algorithms for today's heterogeneous machines requires a level of complexity that was not required in homogeneous environments. Each different type of hardware is likely to have a different workload-grain size for optimal performance. For example, for matrix-matrix multiplication, a GPU will require large matrix sizes in order to achieve high performance, whereas a CPU can achieve its best performance at much smaller sizes.

Furthermore, in a distributed memory environment, the data needs to be partitioned among the nodes to enable load balance and scalability. If the nodes have varying capabilities, then the algorithm needs to allow different workloads to be allocated to each node.

This paper presents research in designing the algorithms and the programming model for high-performance DLA in distributed-memory heterogeneous environments. The compute nodes in these environments can be composed of a mix of multicore CPUs, GPUs, , and coprocessors such as Xeon Phi (formerly MIC), all of which may have varying capabilities and different optimal workload granularity. While the main goal is to obtain as high fraction of the peak performance as possible for the entire 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 distributed-memory lightweight runtime environment, and describe the construction of DLA routines based on it. We demonstrate the new heterogeneous runtime environment and its programming model using the LU factorization.

The design of this new environment considers our experience [14], as well as other state-of-the-art developments in the area, summarized as follows, to extract and develop the techniques best suited for DLA on distributed heterogeneous systems.

3. NOVELTY AND CONTRIBUTIONS

We consider the following to be novel contributions of this paper, they either need to be considered independently or as whole because they are implemented in a single working solution.

1. Our scheduling allows a mix of tasks that are primarily either numerical or non-numerical but can also be both. We have investigated primarily numerical tasks in our prior work [14] and our current work extends it to non-numerical memory-bound tasks. The details can be found in this and following sections.

2. We handle in our extended scheduler tasks with a range of ratios of computational load to memory accesses. This means that the scheduler is much more sensitive to data locality since suboptimal scheduling decisions can no longer be attenuated by high computational load. In order to take advantage of the feature and improved runtime schedule, the user labels tasks with categories that range from mostly compute-bound to mostly memory-bound. Presets are available for common library routines such as BLAS Level 1, 2, or 3.

3. We show a unified method that supports for coprocessors, GPUs, and multicore CPUs without prescribed relative computational strength of the devices. The computational capacity is represented as a single number – a weight that correlates well to the peak performance of the device. To account for the delay of data transfers, we use very simple communication model that uses the peak device link bandwidth to estimate communication overhead. Such a model is to simplistic for large scale networks but is adequate within a single node regime.

4. We implement a hardware-oblivious scheduling with virtual computing devices that can be any combination of a single or multiple CPU cores, a single or multiple GPUs, a single or multiple GPU streams (a stream roughly representing a single NVIDIA SMX) or OpenCL queues, a single or multiple coprocessor threads (the Knights Landing edition of Xeon Phi features up to 240 threads on the high end version of the hardware).

5. We allow the user to use the synchronous communication model with asynchronous communication progress. The former simplifies the programming model while the latter maximizes the achieved bandwidth and reduces the overhead associated with latency.

6. We introduced the concept of multi-grain scheduling with practical implementation that efficiently handles fine-, coarse-, and mid-grained tasks to match them against accelerators and CPUs of drastically difference performance.

7. We use weights to characterize the hardware and to distribute the data across the system, which allows the user to capture performance and bandwidth of compute devices and links. This in turn provides efficient means for the scheduler to distribute work, data, and asynchronously balance the load between the hardware components.

4. RELATED WORK

Linear Algebra: Gaussian elimination with pivoting is efficiently realized in software by means of an LU factorization with row interchanges. In particular, the computationally achieved mapping $PA \rightarrow LU$ is the common implementation vehicle to achieve practical numerical stability and high performance. However, when considering the multitude of hardware components we mentioned earlier, it quickly becomes a challenge of how to achieve good use of each of the components: CPUs, GPUs, and/or coprocessors. We have numerical kernels of various computational intensity, e.g., BLAS Level 1, 2, and 3. Each of these have much different memory footprint and tolerance to the memory system overheads. The kernels that select pivots and its application have a very low computational, e.g., comparable to the Level 1 and 2 BLAS, while contributing nothing to the total floating-point operation count.

Heterogeneous Programming Models: NVIDIA, Intel, and AMD provide programming models that are closely aligned with their hardware offerings – each one offers a vertically integrated software stack. NVIDIA provides the CUDA API to manage their GPUs, Intel offers programming for the Xeon Phi using language annotations such as pragmas and offload directives. AMD programs their GPUs using the OpenCL programming model. Managing a mix of multicore-CPUs, GPUs and Xeon Phi coprocessors within the context of a single program remains a challenge. We are designing a methodology and API that attempts to unify programming in such multi-way heterogeneous hardware environments. The StarPU project [2] has similar goals in providing a unified task based programming environment, however we try to be more adaptive to a wider variety of hardware resources and task grain sizes.

Task-Superscalar Runtime Environments The increasing power and complexity of available hardware has multiplied the difficulty of keeping hardware busy and exacerbated the cost of idle resources. Task-superscalar runtime environments have become a common approach for effective and efficient execution on current hardware. These programming environment present two major advantages. First, programming is done by writing task-structured serial code, which decreases the burden on the programmer. Second, the tasks can be executed in an asynchronous, out-of-order schedule, which can potentially make very efficient use of the hardware resources.

Task-superscalar execution environments take a serial sequence of tasks as input and schedule them for execution in parallel, inferring the data dependencies between the tasks at runtime. The dependencies 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 tasks are then scheduled for execution in an asynchronous, data-driven, task-superscalar runtime environment. This execution can 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. Since serial code is the input to the runtime system, and the parallel execution respects all the data hazards, the correctness of the serial code guarantees parallel correctness.

There has been much research on execution environments that take serial code as input and result in a parallel execution, generally using task superscalar techniques, for example Jade [18], Cilk [5], OpenMP 4.0 [8], Sequoia [12], SuperMatrix [7], OmpSS [17], Habanero [4], StarPU [3], QUARK [22], or the DepSpawn [13] project.

We are developing a programming methodology that could be implemented within the context of several of the available task-superscalar runtime systems. We have chosen to do our development with the QUARK (QUeuing and Runtime for Kernels) because it provides a simple, serial, library based approach to superscalar execution, targeting multicore processors and allowing for low level task placement [22]. It has been used with GPU accelerators [16], and has a prototype distributed memory implementation [21]. QUARK provides the dynamic runtime layer for the PLASMA library [1, 15] and as such has undergone substantial stress testing for performance and scalability.

5. ADAPTIVE MULTIGRAIN SCHEDULER

Figure 1 shows a pseudocode for the main scheduling algorithm – it is sequential in execution but dispatches tasks to run in parallel and on heterogeneous hardware. The else-clause of the outermost if-statement allows the scheduler to execute user tasks on the core assigned to processing user task requests. The switching between scheduling and executing tasks is done based on the computational load and availability of hardware resources. A number of queues hold the tasks ready to be executed – the data they depend on has been made available through either a prior task or an incoming network communication. Invocation of the get_next_task() method on the user_code object switches the context from the scheduler loop to the user-level code that creates tasks and passes them to the scheduler. There are not many restrictions on the user code and a sample is presented in Algorithm 1. The user is responsible for providing computational fragments as tasks, define dataflow dependences between tasks that are free of side-effects, and indicate task priority as well as its kind: memory- or compute-bound. The scheduler will take over the execution of tasks by first inserting them in the appropriate queue (according to tasks type and available code: CPU, GPU, or Phi coprocessor). The selection of the right queue takes into account the length of each queue, which reflects the current and future load of the device, and the computational capacity of the device. This is achieved by assigning tasks to device bins with a greedy heuristic. The queue selection is represented as the find_closest_queue() method of the queues object. Finally, the task is inserted into the appropriate queue according to its priority to allow for progress along the critical path of the task graph. The priorities are user-defined and are optional but in practice, they can give a performance advantage for some workloads. Similarly, appropriate allocation of hardware resources to the scheduler devices can be regarded as a tuning option. For example, the panel factorization in the LU algorithm benefits from the combined cache size of multiple cores [10, 9, 11] and thus should be executed on a virtual device comprised of multiple cores.

```
def main_thread_loop(user_code, queues, threshold):
  # if there are enough tasks for cores
  if queues.total_length() > threshold:
    # resume user's code for task submission
    t = user_code.get_next_task()
    q = queues.find_closest_queue(t.devices())
    q.insert(t, t.priority())
  else:
    # if tasks available for stealing
    if t = queues.steal_task(main_cpu):
      t.execute() # execute a single task
    else:
      queues.wait_for_tasks()
```

Figure 1: Pseudocode for the main loop of the scheduler.

6. HETEROGENEOUS AND ADAPTIVE PROGRAMMING MODEL

In this section we discuss a programming model that produces a higher level of abstraction when programming multi-way heterogeneous resources and allows us to have a unified approach to programming various devices. We describe the techniques that we use in our runtime environment to enable adaptive execution on the available resources. This work builds on our earlier work on programming for multi-way heterogeneous architectures [14].

6.1 A Model for Programming Multi-way Heterogeneous Resources

GPUs and coprocessors have a very high computational peak compared to multicore CPUs. The variety of capabilities that they provide makes it challenging to develop an algorithm that can achieve high performance and reach good scalability in a multi-way heterogeneous environment. 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: be it explicit API calls or implicit function calls inserted by the compiler through programming language directives. The key features taken into account by our model are the capabilities of the computational resources (CPUs, GPUs, Xeon Phi), the memory access, and the communication cost. As with CPUs, the time for an accelerator to access device memory 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 completely the slow memory problem but is often effective.

Accelerators often use multithreading operations that access large data sets that would overflow the size of most caches. When the accelerator's thread unit issues an access to the device memory, that thread unit stalls until the memory returns the requested data. During this time, the accelerator's scheduler switches to another hardware thread and continues executing the other thread. This is how an accelerator exploits program parallelism to keep functional units busy while the memory fulfills past requests. In comparison with CPUs, the accelerator memory access delivers higher absolute bandwidth (around 180 GB/s for K40 and Xeon Phi, and 160 GB/s for Kepler K20c).

To side-step memory issues, we have developed a strategy that prioritizes the data-intensive operations to be executed by the accelerator and to 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, we redesigned the kernels and implemented dynamically guided data distribution to exploit parallelism in order to keep the accelerators and processors busy.

From a programming model point of view, we cannot hide the distinction between the differing levels of parallelism, designed for the CPU host or the accelerator. Each algorithm is converted into a host part and an accelerator part. The routines destined to execute 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 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 mixed together. Our intention is that our model simplifies most of the hardware details, but gives us finer levels of control.

Algorithm 1 shows the pseudocode for the LU factorization from an algorithm designer's point of view. It consists of a sequential code that is simple to comprehend and is independent of the architecture. Each call 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 to allow the QUARK engine to handle the transfer of data automatically. In addition, we developed low-level interfaces and optimizations for both types of accelerators, in order to accommodate hardware-, software- and library-specific tuning and fulfillment of data requirements. We have implemented

Algorithm 1: LU implementation for multiple devices.

```
Initilizer with default values
Task_Flags panel_flags = Task_Flags_Initializer
Set the default priority of tasks
Task_Flag_Set(&task_flags, PRIORITY, 10)
Panel factorization is memory-bound → disable task stealing

Task_Flag_Set(&panel_flags, BLAS2, 0)
memory-bound → locked to CPU
for $k \in \{0, nb, 2 \times nb, \ldots, n\}$ do
    Factorization of the panel A(k:n,k:k+nb)
    TASK: getf2(A(k:n,k:k+nb))
    swap the rows to the left and the right of the panel
    TASK: laswp(A(k:n,1:k))
    TASK: laswp(A(k:n,k+nb:n))
    Schur complement update: The Schur update task,
    $A_{22} \leftarrow A_{22} - A_{21}A_{11}^{-1}A_{12}$, is split into a set of parallel,
    compute intensive tasks that call trsm and gemm. This in-
    creases parallelism and enhances the overall performance.
    Note that the first task (trsm and gemm) consists of the
    update of the next panel. The scheduler inserts it immedi-
    ately in the queue since with a high priority. Right after
    the dependences are satisfied and once the high priority
    tasks finish, the data are sent it to the CPU in order to
    perform the panel factorization of the next step – this
    technique is called lookahead.

    for $j \in \{k + nb, k + 2nb, \ldots, n - nb\}$ do
        trsm(A(k:k+nb,k:k+nb) → A(k:k+nb,j:j+nb)
    if $panel\_m > panel\_n$ then
        gemm with trailing matrix
        for $j \in \{k + nb, k + 2nb, \ldots, n - nb\}$ do
            gemm( A(j:n,k:k+nb) × A(k:k+nb,j,j+nb) →
            A(j:n,j:j+nb) )
```

a set of scheduling directives that are evaluated at runtime in order to fully map the algorithm to the hardware, and to run close to the peak performance of the system. Using these strategies, we can easily develop simple and portable code that can run on different heterogeneous architectures, letting the scheduling and execution engine do the task dependency analysis, resource scheduling, and finally, the task execution. A simple example of this functionality is the lookahead technique. The first task (trsm and gemm) consists of the update of the next panel. The scheduler pushes it first on the queue as a priority task (since it is on the critical path), tracks its dependencies, and once finished, sends it to the CPU in order to perform the panel factorization of the next step. This technique is called lookahead, and is hidden here by the scheduler without any extra lines of code.

6.2 Adaptive Task Scheduling and Data Layout

In this section we overview the techniques that we used to provide an adaptive, scalable, high performance execution in a multi-way heterogeneous environment. Further details and experiments for our choices can be found in our earlier work [14].

The LU pseudocode generates a serial sequence of tasks which are inserted by the algorithm into the QUARK superscalar runtime environment. For an efficient execution, the tasks need to be assigned to the computational resources taking into account the varying computational differences between the resources. In order to keep a measure of the difference between the resources, 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. This α_{ik}, also referred to as a **resource capability weight for the task**, can be provided by the user via a task-flag, or could potentially be estimated by the runtime environment. As an example, the capability-weights for the update operation (a Level 3 BLAS) is around $1:10$ which means that the GPU can execute 10 times as many update tasks as CPU.

The tasks are scheduled using **adaptive scheduling with capability weights**. As a 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 resource 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 preferentially executed on the CPU side. The adaptive heuristic tries to maintain the ratios of the capability-weights across all the resources to maintain a balanced execution time.

When a task is assigned to a computational resource, it may not be executed at once. QUARK schedules tasks for execution after their dependencies are correctly fulfilled and all the data hazards are avoided. A major positive side effect of this superscalar scheduling is that no extra effort is required to enable **dynamic lookahead** in the execution of the algorithm. If tasks from future iterations can be executed after a specific task from the current iteration completes, then the QUARK runtime will expose and schedule these tasks. In order to expose the lookahead tasks as soon as possible, tasks are inserted with **enhanced task priorities**. For example, in the block factorization algorithms, the panel operations are given a higher priority than the update operations, which leads to higher lookahead depth, since the factorization of the next panel will be a higher priority than the current trailing matrix update.

Once the tasks are scheduled for execution, the QUARK runtime provides **transparent data movement**. If QUARK detects the data required for a task is not available at the location that the task is scheduled, it manages the data transfer. 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.

The initial **data layout** is structured so that the data is distributed over all the accelerators in a 1-D block-column cyclic fashion, with an approximately equal number of columns assigned to each. 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.

Given the heterogeneous resources, a standard 1-D block cyclic data layout does not match the work-grain to the resource. We have updated the algorithm and runtime to enable **dynamic data redistribution** which re-adjusts the data layout using the resource capability-weights. Using the QUARK runtime, the data is either distributed or redistributed at runtime in an automatic fashion so that each device gets the appropriate volume of data to match its capabilities.

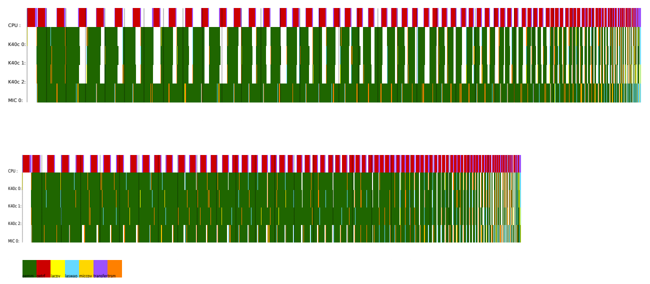


Figure 2: Execution trace of the LU factorization using 3 K40c and 1 Xeon Phi, when adaptive scheduling is disabled (top trace) or enabled (bottom trace).

7. PERFORMANCE RESULTS

Hardware Description and Setup We conducted our experiments on three different systems, denoted A, B, and C, each equipped with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors/sockets, running at 2.6 GHz. In addition,

- Kepler System is equipped with six NVIDIA K20c cards, running at 705 MHz. The practical dgemm peak is about 1000 Gflop/s per GPU.
- Phi System is equipped with three Intel Xeon Phi cards, running at 1.23 GHz, achieving a double precision dgemm peak of about 950 Gflop/s.
- Kepler-Phi System is a heterogeneous system equipped with three NVIDIA K40c cards, running at 825 MHz. The practical dgemm peak is 1200 Gflop/s per GPU. The system is also equipped with an Intel Xeon Phi card, similar to the ones of Phi System.

All the cards of our three systems are connected to the host via two PCIe I/O hubs with 6 GB/s bandwidth. A number of software packages were used for the experiments. On the CPU, we used Intel MKL (Math Kernel Library). On the Xeon Phi, we used the MPSS 3.5 software stack, icc 15.0.3 20150407 which comes with the Composer XE 2015 suite as the compiler. On the GPU we used CUDA version 7.0.27.

8. DISCUSSION

Figure 2 (top), shows the execution trace of the LU factorization on the Kepler-Phi System without the adaptive data assignment but with dynamic task scheduling technique. The panel factorization kernel is depicted in red and has been allocated to the CPU. The kernel that swaps rows of the matrix is illustrated in cyan. Each device swaps only the rows that were assigned to that device. We note that swapping rows on the GPU significantly decreases the overall performance because rows are not stored consecutively in the memory, and thus the GPU threads cannot read it in a coalesced way – the swapping causes thread divergence due to non-contiguous memory accesses. To alleviate this problem somewhat, we developed hardware-specific kernels and optimization methods and we managed to obtain much better performance from each hardware accelerator. The common technique is for the data on the GPU to be stored in transposed form and to use a specialized and efficient GPU kernel that performs row swapping at nearly the speed of the memory bandwidth. It is due to the fact that the row entries are stored as contiguous locations in memory, and thus can be accessed through coalesced reads and/or writes when the laswp function is invoked.

Note that the transpose does not affect any of the other kernels (gemm and trsm) required by the LU factorization. The coalesced reads/writes improve the performance of the laswp function by 1.6 times in our experiments. The compute-intensive kernels (dtrsm and dgemm) are commonly preferred to be executed on the accelerator and, on the figure, are illustrated by the orange and the green colors, respectively. The data transfer is represented with purple color. The use of the lookahead technique described in Algorithm 1, does not require any extra programming effort since it is handled transparently by the QUARK engine through the dependence analysis and priorities indicated by the programmer. The scheduling engine ensures that the next panel (panel of step $k+1$) is updated as soon as possible by the device (accelerator or coprocessor) in order to be sent to the CPU for factorization, while the accelerator device continues the update of the trailing matrix of step k. Also, the QUARK scheduler manages the data transfers to-and-from the CPU in automated fashion by allocating separate threads of execution for them and including them in its standard dataflow analysis. The advantage of such a strategy is not only to hide the cost of the data transfer between the CPU and device by overlapping it with the computation on the device, but also to keep the device busy by providing enough tasks to execute. Since the performance of K40c is about 20% higher than the Xeon Phi, we observe that the three K40c finish their trailing matrix updates (green) earlier than the Xeon Phi. For example, in Figure 2 (top), the Xeon Phi is performing its second trailing matrix update (second green for Phi), while the GPUs have already finished the update of the third step (third green for K40c). As a result, the panel of the fourth step, which is held by the Xeon Phi, is still not ready. This slows down the execution. At this point, sending it to a K40c will not improve anything, since the K40c must wait anyway. Overall, the dataflow strategy and the dynamic scheduling techniques in use, can overcome the Xeon Phi delay, but only to a certain limit, until the amount of available tasks becomes limited due to the imbalance of our model Xeon Phi when compared to our model of the GPU counterparts. Thus, we note that our optimization techniques work for a few steps before the progress becomes limited by the Xeon Phi coprocessor (please consult the figure where black space indicates wait time). This pattern repeats for a few steps. The overall performance is between 30% to 40% less efficient than it should be from the analysis of the basic kernels without load imbalance. This trace is a clear indication for the need of an adaptive technique that we proposed earlier. The execution trace with the adaptive optimizations in place is highlighted in Figure 2 (bottom). The distribution of load and tasks per device is proportional to its performance power. Thus, the waiting time of faster devices is minimized, keeping all devices equally busy. The obtained performance is about 34% better than what it was without the adaptive strategy.

Figure 3 shows the performance scalability of our implementation of the LU factorization in double precision on Kepler-Phi System, using 16 CPU cores only (black curve), or a Xeon Phi alone (magenta curve), or a Xeon Phi in combination with 1, 2, and 3 K40c GPUs – the green, blue and red curves, respectively. The curves show the performance in Gflop/s. Note that Gflop/s metric is inversely proportional to the elapsed time for a constant matrix size, i.e., performance that is two times higher, corresponds to an elapsed time that is two times shorter. Our heterogeneous multi-device implementation achieves perfect scalability for large matrix sizes when there is enough computational load to counteract the adverse effects of limited bandwidth and long memory latencies. The peak performance of the cuBLAS dgemm on one K40c is 1,200 Gflop/s. MKL on the Xeon Phi achieves 950 Gflop/s, which translates to the total peak performance of about 4,550 Gflop/s. Note that the shape of the

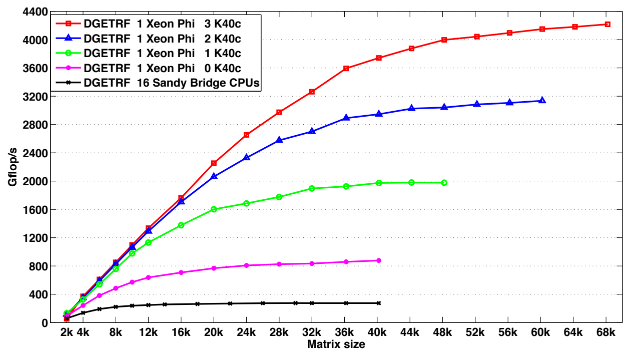


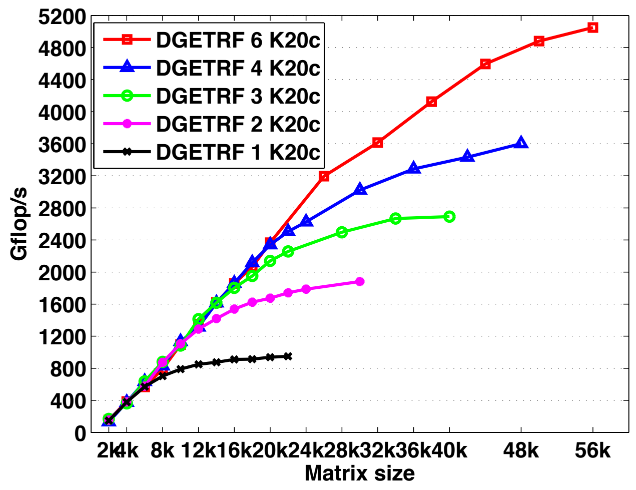
Figure 3: Performance of our dynamic approach on the heterogeneous Kepler-Phi System using either 16 CPU cores Xeon E5-2670, one Intel Xeon Phi and up to 3 NVIDIA K40c and cards.

operands passed to the gemm routine in the LU update is rectangular and reaches about 94% of the square gemm peak mentioned above. Our LU factorization for a matrix of size 60,000 achieves about 4200 Gflop/s on the Kepler-Phi System using one Xeon Phi and 3 Kepler K40c GPUs. This means that our implementation asymptotically comes close to the practical peak performance of the fastest kernel called in the code. It is worth mentioning another beneficial behavior of our implementation that uses the adaptive techniques, namely that for the tested heterogeneous system, the code did not require any further optimizations, and it was able to scale very well without extensive tuning that commonly accompany the effort of porting the software to a new system and which is commonly used for classical software packages that lack adaptive features. The strong scalability of our implementation can also be deduced from the figures by reading the plots in vertical fashion. For a fixed matrix size, slicing the performance plots vertically shows strong scalability of the LU factorization when adding more devices. The ratio between two curves illustrates the speedup that was obtained. For example, using one Xeon Phi for a matrix of size 40,000 is about $3\times$ faster than using CPUs only. Performance reaches 2,000 Gflop/s by adding an extra K40c, which is about 7 times faster than the CPU alone, and 2.2 times faster than using 1 Xeon Phi alone. Similarly, using the 3 K40c within the Xeon Phi make the factorization more than 14 times faster than the CPU only implementation, and about 4.2 times faster than when using a single Xeon Phi.

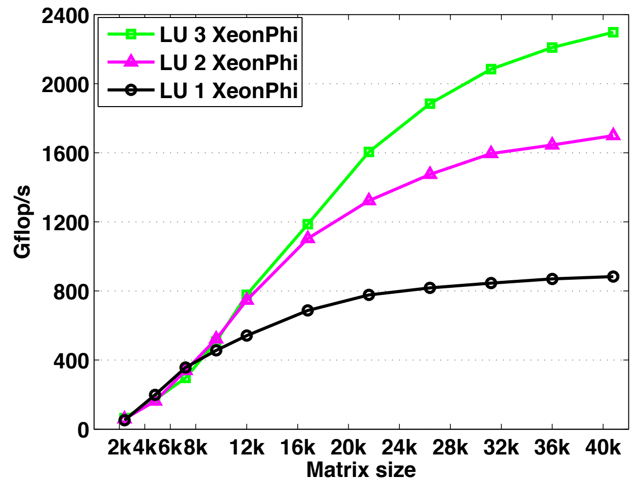
We observed similar performance trends when using either Kepler System or Phi System. In Figures 4a and 4b, we show the performance obtained by our implementation using up to 6 K20c GPUs of Kepler System and up to three Xeon Phi's of Phi System, respectively. The described feature of our programming model and dynamic scheduling techniques allows us to achieve high scalability without much coding or tuning efforts. The same baseline code was compiled and executed on these different machines without any changes.

9. CONCLUSIONS AND FUTURE WORK

While heterogeneous compute nodes have become ubiquitous, the need for an easy programming paradigm capable of providing portability and efficiency across a large range of hybrid environments became ever critical. We designed and implemented a programming model that features a number of optimization techniques for developing high-performance algorithms suitable for multi-way heterogeneous environments with more than one type of a hardware



(a) Performance of our dynamic approach on up to 6 cards, each being NVIDIA K20c.



(b) Performance of our dynamic approach on up to 3 cards, each being Intel Xeon Phi.

Figure 4: Performance of our model using the adaptive dynamic scheduling on hybrid systems.

accelerator. In particular, we presented best practices and methodologies from the development of high-performance LU factorization for accelerators and coprocessors. We also showed how judicious modifications to superscalar task scheduling were used to ensure that we meet two competing goals: (1) to obtain high fraction of the peak performance for the combined resources of the heterogeneous system, and (2) to employ a programming model that would simplify the development. Our performance analysis unequivocally demonstrates that our approach improves the performance of heterogeneous platforms by using the adaptive scheduling techniques and also enhances the scalability of the underlying algorithms by providing a set of features capable of mapping the algorithm and its data to all potential computing resources. This principle can be extended to many other algorithms such as the eigenvalue and singular value methods or even sparse solvers. Future work will include merging CUDA, OpenCL, and Intel Xeon Phi development branches into a single library using our new programming model.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant numbers ACI-1339822 and 1137097, and by the University of Tennessee through the Beacon project, the Department of Energy, and the NVIDIA and Intel Corporations. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

10. REFERENCES

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA users guide, 2010. Innovative Computing Laboratory, University of Tennessee Knoxville.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computat. Pract. Exper.*, 23(2):187–198, 2011. http://dx.doi.org/10.1002/cpe.1631.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Delft, The Netherlands, 2009. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-03868-6, http://dx.doi.org/10.1007/978-3-642-03869-3_80.

[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, Orlando, Florida, USA, 2009. ACM, New York, NY, USA. ISBN 978-1-60558-768-4, http://doi.acm.org/10.1145/1639950.1639989.

[5] 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. ISSN 0362-1340, http://doi.acm.org/10.1145/209937.209958.

[6] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2007.

[7] 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, San Diego, California, USA, 2007. ACM, New York, NY, USA. ISBN 978-1-59593-667-7, http://doi.acm.org/10.1145/1248377.1248397.

[8] L. Dagum and R. Menon. OpenMP: An industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998. ISSN 1070-9924, http://dx.doi.org/10.1109/99.660313.

[9] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *ParCo 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011.

[10] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. High performance matrix inversion based on LU factorization for multicore architectures. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pages 33–42, Seattle, Washington, USA, November 2011. Association for Computing Machinery.

[11] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, May 2014. Article first published online: 18 SEP 2013, DOI: 10.1002/cpe.3110.

[12] 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, Tampa, Florida, 2006. ACM, New York, NY, USA. ISBN 0-7695-2700-0, http://doi.acm.org/10.1145/1188455.1188543.

[13] 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. ISSN 0167-8191, Novel On-Chip Parallel Architectures and Software Support.

[14] 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, Pheonix, AZ, USA, May 2014. IEEE Computer Society, Washington, DC, USA.

[15] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321, Mar. 2011.

[16] J. Kurzak, R. Nath, P. Du, and J. Dongarra. An implementation of the tile QR factorization for a GPU and multiple CPUs. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 248–257, Berlin, Heidelberg, 2012. Springer-Verlag.

[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. http://dx.doi.org/10.1109/2.214440.

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

[20] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990. DOI 10.1145/79173.79181.

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

[22] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.