Model-Driven One-Sided Factorizations on Multicore Accelerated Systems

Jack Dongarra [1] [2] [3]***, Azzam Haidar*** [1]***, Jakub Kurzak*** [1]***, Piotr Luszczek*** [1]***, Stanimire Tomov*** [1]***, Asim YarKhan*** [1]

Hardware heterogeneity of the HPC platforms is no longer considered unusual but instead have become the most viable way forward towards Exascale. In fact, the multitude of the heterogeneous resources available to modern computers are designed for different workloads and their efficient use is closely aligned with the specialized role envisaged by their design. Commonly in order to efficiently use such GPU resources, the workload in question must have a much greater degree of parallelism than workloads often associated with multicore processors (CPUs). Available GPU variants differ in their internal architecture and, as a result, are capable of handling workloads of varying degrees of complexity and a range of computational patterns. This vast array of applicable workloads will likely lead to an ever accelerated mixing of multicore-CPUs and GPUs in multi-user environments with the ultimate goal of offering adequate computing facilities for a wide range of scientific and technical workloads. In the following paper, we present a research prototype that uses a lightweight runtime environment to manage the resource-specific workloads, and to control the dataflow and parallel execution in hybrid systems. Our lightweight runtime environment uses task superscalar concepts to enable the developer to write serial code while providing parallel execution. This concept is reminiscent of dataflow and systolic architectures in its conceptualization of a workload as a set of side-effect-free tasks that pass data items whenever the associated work assignment have been completed. Additionally, our task abstractions and their parametrization enable uniformity in the algorithmic development across all the heterogeneous resources without sacrificing precious compute cycles. We include performance results for dense linear algebra functions which demonstrate the practicality and effectiveness of our approach that is aptly capable of full utilization of a wide range of accelerator hardware.

Keywords: hardware accelerators, dense linear algebra, task superscalar scheduling.

1. Introduction

With the release of CUDA in 2007, the communities of scientific, HPC, and technical computing 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 intend in this paper to compare between GPUs and the recently introduced MIC coprocessor. Instead, we take a different stand because we believe that the 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 simultaneously utilizing all three: CPUs, GPUs, and coprocessors. We call this a multi-way heterogeneity for which we present a unified programming model that alleviates the complexity of dealing with multiple software stacks for computing, communication, and software libraries.

[1]University of Tennessee, Knoxville, USA
[2]Oak Ridge National Laboratory, Oak Ridge, USA
[3]University of Manchester, Manchester M13 9PL, UK

2. Background and Related Work

This paper presents research in algorithms and a programing model for high-performance dense linear algebra (DLA) factorizations for heterogeneous multiple multicore CPUs, GPUs, and Intel Xeon Phi coprocessors (MICs). The mix can contain resources with varying capabilities, e.g., CUDA GPUs same architecture but 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, an often 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. In the specific context of DLA, $O(n^2)$ data requires $O(n^3)$ floating point operations (flop's). In contrast, less compute intensive algorithms that use sparse linear algebra can reach only a fraction of the peak performance, e.g., a few Gflop/s for highly optimized sparse matrix-vector multiply (SpMV) [6] versus over 1000 Gflop/s for double-precision matrix-matrix multiply (DGEMM) on a Kepler K20c GPU [11]. This highlights the interest in DLA and the importance of designing computational applications that can make efficient use of the available hardware resources.

Early results for DLA were tied to development of high performance Basic Linear Algebra Subroutines (BLAS). Volkov el al. [30] developed fast single-precision matrix-matrix multiply (SGEMM) for the NVIDIA Tesla GPUs and highly efficient LU, QR, and Cholesky factorizations based on that. The fastest DGEMM and factorizations based on it for the NVIDIA Fermi GPUs were developed by Nath et al. [20]. These early BLAS developments were eventually incorporated into the CUBLAS library [11]. The main DLA algorithms from LAPACK were developed for a single GPU and released through the MAGMA library [19].

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 [3, 5, 9, 12, 25, 26], 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 with OpenCL [9] being a natural target for supporting AMD Radeon compute cards.

Task superscalar runtime environments have become a common approach for effective and efficient execution in multicore environments. This task scheduling mechanism has its roots in dataflow execution, which has a long history dating to the nineteen sixties [7, 16, 24]. 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 [29], which is increasingly wasteful in face of ever larger numbers of cores in a single CPU socket.

In our current research, we build on the QUARK [31, 32] 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 [2, 17].

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 [23], Cilk [8], Sequoia [13], SuperMatrix [10], OmpSS [22], Habanero [5], StarPU [4], or the DepSpawn [14] 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 representative of a lightweight, task-superscalar runtime environment.

Both GPUs and Intel Xeon Phi coprocessors provide high degree of parallelism that can deliver excellent application performance for DLA. However, it is important to understand the distinctions in programming models between GPUs and Intel Xeon Phi coprocessors.

2.2.1. CUDA

CUDA utilizes a large number of concurrent threads of execution to exploit hardware parallelism and achieve high performance. CUDA threads are grouped together into blocks that execute concurrently on the GPU Streaming Multiprocessors (SMs) following the SIMD execution model. NVIDIA Kepler K20 GPU has 2496 CUDA cores and delivers 1.17 Tflop/s in peak double precision floating point performance.

2.2.2. MIC Architecture and Programming Models

The Intel Xeon Phi coprocessor comprises of 61 Intel Architecture (IA) cores. Additionally, there are 8 memory controllers supporting up to 16 GDDR5 channels delivering a theoretical bandwidth of 352 GB/s [1]. Its peak double precision floating point performance is 1200 Gflop/s. Unlike a GPU, the Xeon Phi coprocessor runs a Linux operating system and provides x86 compatibility. It supports popular programming models used on multi-core architectures including MPI, OpenMP and Threading Building Blocks. Xeon Phi also offers the following flexible programming models depending on applications:

- **Native model**: It is used to run applications entirely on a Xeon Phi coprocessor like any other multi-core system.

- **Offload model**: It is used to utilize a Xeon Phi coprocessor as an accelerator to execute computation intensive regions of an application offloaded from host.

Offloading incurs overhead costs for initialization, marshaling and transferring data, and code invocation. Applications must have a high ratio of computation to communication for the offloaded portion. Offload communication overheads can be hidden if code is structured to overlap them with computation, or if data is reused across offload regions [21]. To effectively make use of this model, asynchronous offload features must be used, which include: asynchronous data transfer, asynchronous compute and memory management without data transfer. Signal and wait clauses need to be used in offload pragmas and directives to enable asynchronous data transfer and computation. For example:

offload_transfer pragma/directive called with a signal clause begins an asynchronous data transfer.

offload pragma/directive called with a signal clause begins an asynchronous computation.

offload_wait pragma/directive called with a wait clause blocks execution until an asynchronous data transfer or computation is complete.

At this point, this programming model has been codified by the OpenMP 4.0 standard and may be considered analogous to the OpenACC standard initially implemented on GPUs only.

Beside the above programming models, Intel provides two software libraris – SCIF (Symmetric Communication Interface) and COI (Coprocessor Offload Infrastructure) – to ease the development of software tools and applications that run on the Xeon Phi Coprocessor.

- SCIF (Symmetric Communication Interface) is an API for communication between processes on MIC and the host CPU within the same system. It resembles POSIX socket interface whereby connections are established using *scif_listen*, *scif_connect*, and *scif_accept* routines. For efficient communication between the processes SCIF API provides send-receive semantics where one serves as the source and the other as the destination of the communication channel. Efficient data transfer is ensured by Remote Memory Access (RMA) semantics where a region of memory needs to be registered (using *scif_register* first and *scif_unregister* afterwards) for remote access. Upon registration, further data transfers (using *scif_readfrom* and *scif_writeto* can occur in a one-sided manner with just one process either reading from or writing to a memory window. These reads and writes can make both synchronous and asynchronous progress and the completion of asynchronous RMA operations is checked using *scif_fence_mark* and *scif_fence_wait* routines.
- COI (Coprocessor Offload Infrastructure) model exposes a pipelined programming model to the user. This model allows workloads to run and data to be moved asynchronously, allowing the host processor, device processor, and DMA engines to remain busy simultaneously and independently. This library provides services to create coprocessor-side processes, create FIFO pipelines between the host and coprocessor, move code and data, invoke code (functions) on the coprocessor, manage memory buffers that span the host and coprocessor, enumerate available resources, and so on [21].

3. BLAS

The *Basic Linear Algebra Subroutines* (BLAS) are the main building blocks for dense matrix software packages. Furthermore, the matrix-matrix multiplication routine is the most common

```
for (m = 0; m < M; m++)
  for (n = 0; n < N; n++)
    for (k = 0; k < K; k++)
      C[n][m] += A[k][m]*B[n][k];
```

Figure 1. Canonical form of matrix-matrix multiplication.

```
for (m_ = 0; m_ < M; m_+=tileM)
 for (n_ = 0; n_ < N; n_+=tileN)
  for (k_ = 0; k_ < K; k_+=tileK)
   for (m = 0; m < tileM; m++)
    for (n = 0; n < tileN; n++)
     for (k = 0; k < tileK; k++)
      C[n_+n][m_+m] +=
       A[k_+k][m_+m]*
       B[n_+n][k_+k];
```

Figure 2. Matrix-matrix multiplication with loop tiling.

```
for (m_ = 0; m_ < M; m_+=tileM)
 for (n_ = 0; n_ < N; n_+=tileN)
  for (k_ = 0; k_ < K; k_+=tileK)
  {
   instruction
   instruction
   instruction
    ...
  }
```

Figure 3. Matrix-matrix multiplication with complete unrolling of tile operations.

and most performance-critical BLAS routine, more so in the context of the factorizations that we consider in this paper. This section presents the process of building a fast matrix-matrix multiplication kernel for the GPU in double precision and in real arithmetic (DGEMM) by using the process of autotuning. The target is the NVIDA K40c card.

In its canonical form, matrix-matrix multiplication is represented by three nested loops as is shown in Figure 1. The primary tool in optimizing matrix-matrix multiplication is the technique of *loop tiling*. Tiling replaces a single loop with two loops: the inner loop that increments the iteration counter by one, and the outer loop that increments the iteration counter by the tiling factor. In the case of matrix-matrix multiplication, tiling replaces the three loops of Figure 1 with the six loops of Figure 2. Tiling of the matrix-matrix multiplication exploits the *surface to volume effect* mentioned before: execution of $O(n^3)$ floating-point operations over $O(n^2)$ data.

Next, the technique of loop unrolling is applied, which replaces the three innermost loops with a single block of straight-line of code (a single *basic block*), as shown schematically in Figure 3. The purpose of unrolling is twofold: to reduce the penalty of looping (the overhead of incrementing loop counters, advancing data pointers and conditional branching), and to increase instruction-level parallelism by creating long sequences of independent instructions, which can fill out the processor's pipeline and thus increase the instruction issue rate.

This optimization sequence is universal for almost any computer architecture, including "standard" superscalar processors with cache memories, as well as GPU accelerators and other less conventional architectures. Tiling, also referred to as blocking, is often applied multiple times to accommodate successive levels of cache, the registers file, TLB, etc.

In the case of a GPU, the C matrix (in $C \leftarrow C - A \times B$) is overlaid with a 2D grid of thread blocks, each one responsible for computing a single tile of C. Since the code of a GPU kernel spells out the operation of a single thread block, the two outer loops disappear, and only one loop remains – the loop advancing along the k dimension tile by tile. Unrolling the outer loop, and then fusing together copies of the inner loop, sometimes called unroll and jam, is usually

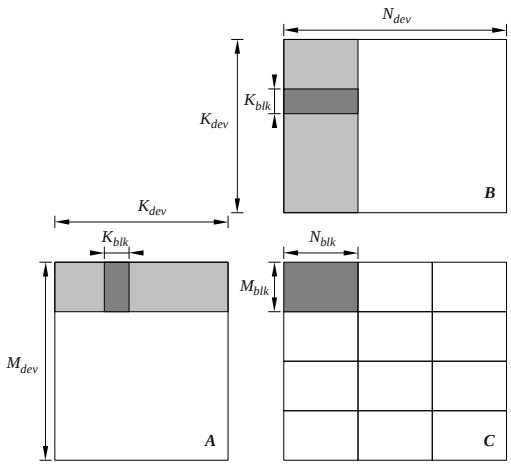

Figure 4. GEMM at the device level.

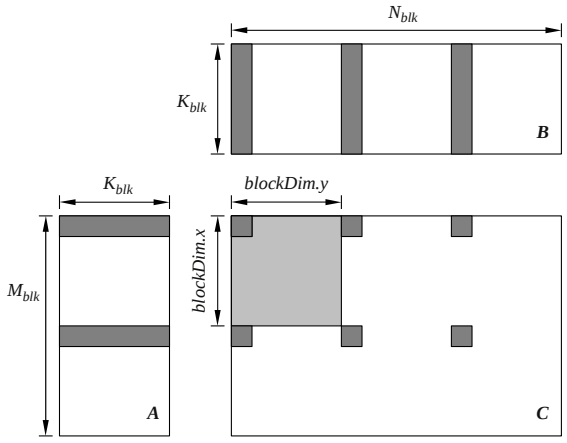

Figure 5. GEMM at the block level.

applied until the point where the register file is exhausted. Unrolling the inner loop beyond that point, can be used to further reduce the overhead of looping.

Figure 4 shows the GPU implementation of matrix-matrix multiplication at the device level. Each thread block computes a tile of C (dark gray) by passing through a stripe of A and a stripe of B (light gray). The code iterates over A and B in chunks of K_{blk} (dark gray). The thread block follows the cycle of:

- making texture reads of the small, dark gray, stripes of A and B and storing them in shared memory,
- synchronizing threads with the `__syncthreads()` call,
- loading A and B from shared memory to registers and computing the product,
- synchronizing threads with the `__syncthreads()` call.

After the complete sweep of the light gray stripes of A and B completes, the tile of C is read, updated and stored back to the device memory. Figure 5 shows closer what happens in the inner loop. The light gray area shows the shape of the thread block. The dark gray regions show how a single thread iterates over the tile.

Figure 6 shows the complete kernel implementation in CUDA. Tiling is defined by `BLK_M`, `BLK_N`, and `BLK_K`. `DIM_X` and `DIM_Y` specify how the thread block covers the tile of C, `DIM_XA` and

DIM_YA define how the thread block covers a stripe of A, and finally DIM_XB and DIM_YB define how the thread block covers a stripe of B.

In lines 24–28 the values of C are set to zero. In lines 32–38 a stripe of A is read (texture reads) and stored in shared memory. In lines 40–46 a stripe of B is read (texture reads) and stored in shared memory. The __syncthreads() call in line 48 ensures that the process of reading of A and B, and storing in shared memory, is finished before operation continues. In lines 50–56 the product is computed, using the values from shared memory. The __syncthreads() call in line 58 ensures that computing the product is finished and the shared memory can be overwritten with new stripes of A and B. In lines 60 and 61 the pointers are advanced to the location of new stripes. When the main loop completes, C is read from device memory, modified with the accumulated product, and written back, in lines 64–77. The use of texture reads with clamping eliminates the need for *cleanup code* to handle matrix sizes not exactly divisible by the tiling factors.

With the parametrized code in place, what remains is the actual autotuning process, i.e., finding good values for the nine tuning parameters. Here the process used in the *Bench-testing Environment for Automated Software Tuning* (BEAST) project is described. It relies on three components:

1. defining the search space,
2. pruning the search space by applying filtering constraints, and
3. benchmarking the remaining configurations and selecting the best performer.

The important point in the BEAST project is to not introduce artificial and/or arbitrary limitations on the search process.

The loops in Figure 7 define the search space for the autotuning of the matrix-matrix multiplication of Figure 6. The two outer loops sweep through all possible 2D shapes of the thread block – up to the device limit in each dimension. The three inner loops sweep through all possible tiling sizes – up to arbitrarily high values, represented by the INF symbol. In practice, the actual values to substitute for the INF symbols can be found by choosing a small starting point, e.g., (64, 64, 8), and going up until further increase has no effect on the number of kernels that pass the selection process based on pruning constraints.

The list of pruning constraints consists of nine simple checks that eliminate kernels deemed inadequate for one of several reasons:

- The kernel would not compile because it exceeded a hardware limit.
- The kernel would compile but failed to launch because it exceeded a hardware limit.
- The kernel would compile and launch, but produced invalid results due to the limitations of the implementation, e.g., unimplemented corner case.
- The kernel would compile, launch, and produce correct results, but have no chance of running fast, due to an obvious performance shortcoming, such as very low occupancy.

The nine checks rely on basic hardware parameters, which can be obtained by querying the card with the CUDA API. The parameters include:

1. The number of threads in the block is not divisible by the warp size.
2. The number of threads in the block exceeds the hardware maximum.
3. The number of registers per thread, to store C, exceeds the hardware maximum.
4. The number of registers per block, to store C, exceeds the hardware maximum.
5. The shared memory per block, to store A and B, exceeds the hardware maximum.
6. The thread block cannot be shaped to read A and B without cleanup code.

```
extern "C" __global__
void beast_gemm_kernel(
    int M, int N, int K,
    double alpha, double *A, int lda,
                  double *B, int ldb,
    double  beta, double *C, int ldc )
{
    int blx = blockIdx.x;        // block's m position
    int bly = blockIdx.y;        // block's n position
    int idx = threadIdx.x;       // thread's m position in C
    int idy = threadIdx.y;       // thread's n position in C
    int idt = DIM_X*idy+idx;     // thread's number

    int idxA = idt % DIM_XA;     // thread's m position for loading A
    int idyA = idt / DIM_XA;     // thread's n position for loading A
    int idxB = idt % DIM_XB;     // thread's m position for loding B
    int idyB = idt / DIM_XB;     // thread's n position for loading B

    __shared__ double sA[BLK_K][BLK_M+1];   // shared memory buffer for A
    __shared__ double sB[BLK_N][BLK_K+1];   // shared memory buffer for B
    double rC[BLK_N/DIM_Y][BLK_M/DIM_X];    // registers for C

    int coord_A = blx*BLK_M     + idyA*lda+idxA;    // A stripe's initial location
    int coord_B = bly*BLK_N*ldb + idyB*ldb+idxB;    // B stripe's initial location
    int m, n, k, kk;                                // loop counters

    #pragma unroll
    for (n = 0; n < BLK_N/DIM_Y; n++)
      #pragma unroll
      for (m = 0; m < BLK_M/DIM_X; m++)
        rC[n][m] = 0.0;

    for (kk = 0; kk < K; kk += BLK_K)
    {
        #pragma unroll
        for (n = 0; n < BLK_K; n += DIM_YA)
            #pragma unroll
            for (m = 0; m < BLK_M; m += DIM_XA) {
                int2 v = tex1Dfetch(tex_ref_A, coord_A + n*lda+m);
                sA[n+idyA][m+idxA] = __hiloint2double(v.y, v.x);
            }

        #pragma unroll
        for (n = 0; n < BLK_N; n += DIM_YB)
            #pragma unroll
            for (m = 0; m < BLK_K; m += DIM_XB) {
                int2 v = tex1Dfetch(tex_ref_B, coord_B + n*ldb+m);
                sB[n+idyB][m+idxB] = __hiloint2double(v.y, v.x);
            }

        __syncthreads();

        #pragma unroll
        for (k = 0; k < BLK_K; k++)
            #pragma unroll
            for (n = 0; n < BLK_N/DIM_Y; n++)
                #pragma unroll
                for (m = 0; m < BLK_M/DIM_X; m++)
                    rC[n][m] += sA[k][m*DIM_X+idx] * sB[n*DIM_Y+idy][k];

        __syncthreads();

        coord_A += BLK_K*lda;
        coord_B += BLK_K;
    }

    #pragma unroll
    for (n = 0; n < BLK_N/DIM_Y; n++) {
        int coord_dCn = bly*BLK_N + n*DIM_Y+idy;
        #pragma unroll
        for (m = 0; m < BLK_M/DIM_X; m++) {
            int coord_dCm = blx*BLK_M + m*DIM_X+idx;
            if (coord_dCm < M && coord_dCn < N) {
                int offsC = coord_dCn*ldc + coord_dCm;
                double &regC = rC[n][m];
                double &memC = C[offsC];
                memC = alpha*regC + beta*memC;
            }
        }
    }
}
```

Figure 6. Complete dgemm (C = alpha A B + beta C) implementation in CUDA.

```
// Sweep thread block dimensions.
for (dim_m = 1; dim_m <= MAX_THREADS_DIM_X; dim_m++)
 for (dim_n = 1; dim_n <= MAX_THREADS_DIM_Y; dim_n++)
  // Sweep tiling sizes.
  for (blk_m = dim_m; blk_m < INF; blk_m += dim_m)
   for (blk_n = dim_n; blk_n < INF; blk_n += dim_n)
    for (blk_k = 1; blk_k < INF; blk_k++)
    {
     // Apply pruning constraints.
    }
```

Figure 7. The parameter search space for the autotuning of matrix multiplication.

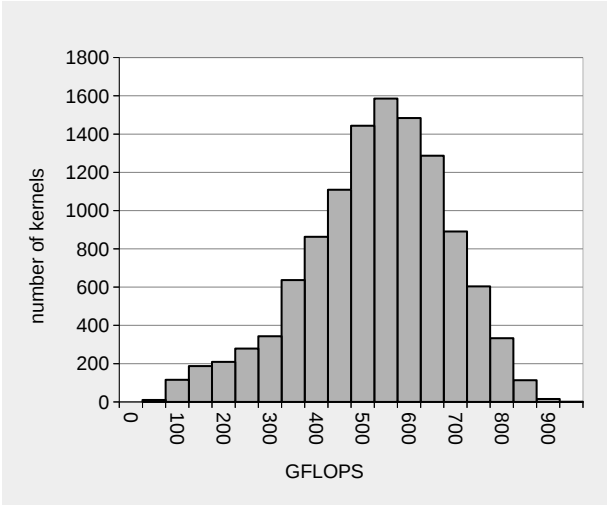

Figure 8. Distribution of the dgemm kernels.

7. The number of load instructions, from shared memory to registers, in the innermost loop, in the PTX code, exceeds the number of *Fused Multiply-Adds* (FMAs).
8. Low occupancy due to high number of registers per block to store C.
9. Low occupancy due to the amount of shared memory per block to read A and B.

In order to check the last two conditions, the number of registers per block, and the amount of shared memory per block are computed. Then the maximum number of possible blocks per multiprocessor is found, which gives the maximum possible number of threads per multiprocessor. If that number is lower than the minimum occupancy requirement, the kernel is discarded. Here the threshold is set to a fairly low number of 256 threads, which translates to minimum occupancy of 0.125 on the Nvidia K40 card, with the maximum number of 2,048 threads per multiprocessor.

This process produces 14,767 kernels, which can be benchmarked in roughly one day. 3,256 kernels fail to launch due to excessive number of registers per block. The reason is that the pruning process uses a lower estimate on the number of registers, and the compiler actually produces code requiring more registers. We could detect it in compilation and skip benchmarking of such kernels or we can run them and let then fail. For simplicity we chose the latter. We could also cap the register usage to prevent the failure to launch. However, capping register usage usually produces code of inferior performance.

Eventually, 11,511 kernels ran successfully and pass correctness checks. Figure 8 shows the performance distribution of these kernels. The fastest kernel achieves 900 Gflop/s with tiling of $96 \times 64 \times 12$, with 128 threads (16×8 to compute C, 32×4 to read A, and 4×32 to read B). The achieved occupancy number of 0.1875 indicates that, most of the time, each multiprocessor executes 384 threads (three blocks).

In comparison, CUBLAS achieves the performance of 1,225 Gflop/s using 256 threads per multiprocessor. Although CUBLAS achieves a higher number, this example shows the effectiveness of the autotuning process in quickly creating well performing kernels from high level language source codes. This technique can be used to build kernels for routines not provided in vendor libraries, such as extended precision BLAS (double-double and triple-float), BLAS for oddly shaped matrices (tall and skinny), etc. Even more importantly, this technique can be used to build domain specific kernels for much broader spectrum of application areas.

As the last interesting observation, we offer a look at the PTX code produced by NVIDIA's `nvcc` compiler (Figure 9). We can see that the compiler does exactly what is expected: it completely unrolls the loops in lines 50–56 of the C code in Figure 6 into a stream of loads from shared memory to registers followed by FMA instructions, with substantially larger number of FMAs than loads.

4. Algorithmic Advancements

In this section, we present the linear algebra aspects of our generic solution for development of either Cholesky, LU (based on Gaussian elimination), or QR (based on applying Householder reflectors) factorizations which use block outer-product updates of the trailing matrix. These three factorizations are commonly known as the *one-sided* factorizations because they apply a sequence of transformations on only one side of the original matrix and thus they do not preserve the matrix spectrum as the *two-sided* transformations do.

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 the currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow$$

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

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

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \rightarrow \begin{bmatrix} XY \end{bmatrix},$$

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

Observe two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update: $A^{(i)} \rightarrow A^{(i+1)}$. The implementation of these two

```
1   ld.shared.f64   %fd258, [%rd3];
2   ld.shared.f64   %fd259, [%rd4];
3   fma.rn.f64  %fd260, %fd258, %fd259, %fd1145;
4   ld.shared.f64   %fd261, [%rd3+128];
5   fma.rn.f64  %fd262, %fd261, %fd259, %fd1144;
6   ld.shared.f64   %fd263, [%rd3+256];
7   fma.rn.f64  %fd264, %fd263, %fd259, %fd1143;
8   ld.shared.f64   %fd265, [%rd3+384];
9   fma.rn.f64  %fd266, %fd265, %fd259, %fd1142;
10  ld.shared.f64   %fd267, [%rd3+512];
11  fma.rn.f64  %fd268, %fd267, %fd259, %fd1141;
12  ld.shared.f64   %fd269, [%rd3+640];
13  fma.rn.f64  %fd270, %fd269, %fd259, %fd1140;
14  ld.shared.f64   %fd271, [%rd4+832];
15  fma.rn.f64  %fd272, %fd258, %fd271, %fd1139;
16  fma.rn.f64  %fd273, %fd261, %fd271, %fd1138;
17  fma.rn.f64  %fd274, %fd263, %fd271, %fd1137;
18  fma.rn.f64  %fd275, %fd265, %fd271, %fd1136;
19  fma.rn.f64  %fd276, %fd267, %fd271, %fd1135;
20  fma.rn.f64  %fd277, %fd269, %fd271, %fd1134;
21  ld.shared.f64   %fd278, [%rd4+1664];
22  fma.rn.f64  %fd279, %fd258, %fd278, %fd1133;
23  fma.rn.f64  %fd280, %fd261, %fd278, %fd1132;
24  fma.rn.f64  %fd281, %fd263, %fd278, %fd1131;
25  fma.rn.f64  %fd282, %fd265, %fd278, %fd1130;
26  fma.rn.f64  %fd283, %fd267, %fd278, %fd1129;
27  fma.rn.f64  %fd284, %fd269, %fd278, %fd1128;
28  ld.shared.f64   %fd285, [%rd4+2496];
29  fma.rn.f64  %fd286, %fd258, %fd285, %fd1127;
30  fma.rn.f64  %fd287, %fd261, %fd285, %fd1126;
31  fma.rn.f64  %fd288, %fd263, %fd285, %fd1125;
32  fma.rn.f64  %fd289, %fd265, %fd285, %fd1124;
33  fma.rn.f64  %fd290, %fd267, %fd285, %fd1123;
34  fma.rn.f64  %fd291, %fd269, %fd285, %fd1122;
35  ld.shared.f64   %fd292, [%rd4+3328];
36  fma.rn.f64  %fd293, %fd258, %fd292, %fd1121;
37  fma.rn.f64  %fd294, %fd261, %fd292, %fd1120;
38  fma.rn.f64  %fd295, %fd263, %fd292, %fd1119;
39  fma.rn.f64  %fd296, %fd265, %fd292, %fd1118;
40  fma.rn.f64  %fd297, %fd267, %fd292, %fd1117;
41  fma.rn.f64  %fd298, %fd269, %fd292, %fd1116;
42  ld.shared.f64   %fd299, [%rd4+4160];
43  fma.rn.f64  %fd300, %fd258, %fd299, %fd1115;
44  fma.rn.f64  %fd301, %fd261, %fd299, %fd1114;
45  fma.rn.f64  %fd302, %fd263, %fd299, %fd1113;
46  fma.rn.f64  %fd303, %fd265, %fd299, %fd1112;
47  fma.rn.f64  %fd304, %fd267, %fd299, %fd1111;
48  fma.rn.f64  %fd305, %fd269, %fd299, %fd1110;
49  ld.shared.f64   %fd306, [%rd4+4992];
50  fma.rn.f64  %fd307, %fd258, %fd306, %fd1109;
51  fma.rn.f64  %fd308, %fd261, %fd306, %fd1108;
52  fma.rn.f64  %fd309, %fd263, %fd306, %fd1107;
53  fma.rn.f64  %fd310, %fd265, %fd306, %fd1106;
54  fma.rn.f64  %fd311, %fd267, %fd306, %fd1105;
55  fma.rn.f64  %fd312, %fd269, %fd306, %fd1104;
56  ld.shared.f64   %fd313, [%rd4+5824];
57  fma.rn.f64  %fd314, %fd258, %fd313, %fd1103;
58  fma.rn.f64  %fd315, %fd261, %fd313, %fd1102;
59  fma.rn.f64  %fd316, %fd263, %fd313, %fd1101;
60  fma.rn.f64  %fd317, %fd265, %fd313, %fd1100;
61  fma.rn.f64  %fd318, %fd267, %fd313, %fd1099;
62  fma.rn.f64  %fd319, %fd269, %fd313, %fd1098;
63  ld.shared.f64   %fd320, [%rd3+776];
64  ld.shared.f64   %fd321, [%rd4+8];
65  fma.rn.f64  %fd322, %fd320, %fd321, %fd260;
66  ld.shared.f64   %fd323, [%rd3+904];
67  fma.rn.f64  %fd324, %fd323, %fd321, %fd262;
68  ld.shared.f64   %fd325, [%rd3+1032];
69  fma.rn.f64  %fd326, %fd325, %fd321, %fd264;
70  ld.shared.f64   %fd327, [%rd3+1160];
71  fma.rn.f64  %fd328, %fd327, %fd321, %fd266;
72  ld.shared.f64   %fd329, [%rd3+1288];
73  fma.rn.f64  %fd330, %fd329, %fd321, %fd268;
74  ld.shared.f64   %fd331, [%rd3+1416];
75  fma.rn.f64  %fd332, %fd331, %fd321, %fd270;
76  ld.shared.f64   %fd333, [%rd4+840];
77  fma.rn.f64  %fd334, %fd320, %fd333, %fd272;
78  fma.rn.f64  %fd335, %fd323, %fd333, %fd273;
79  fma.rn.f64  %fd336, %fd325, %fd333, %fd274;
80  fma.rn.f64  %fd337, %fd327, %fd333, %fd275;
81  fma.rn.f64  %fd338, %fd329, %fd333, %fd276;
82  fma.rn.f64  %fd339, %fd331, %fd333, %fd277;
83  ld.shared.f64   %fd340, [%rd4+1672];
84  fma.rn.f64  %fd341, %fd320, %fd340, %fd279;
85  fma.rn.f64  %fd342, %fd323, %fd340, %fd280;
86  fma.rn.f64  %fd343, %fd325, %fd340, %fd281;
87  fma.rn.f64  %fd344, %fd327, %fd340, %fd282;
88  fma.rn.f64  %fd345, %fd329, %fd340, %fd283;
89  fma.rn.f64  %fd346, %fd331, %fd340, %fd284;
```

Figure 9. A portion of the PTX code for the innermost loop of the fastest DGEMM kernel.

	Cholesky	QR (Householder reflectors)	LU (Gaussian Elimination)
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK2 xGEMM	xLARFB	xLASWP xTRSM xGEMM

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

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 routine names in the algorithm.

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

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

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$_{\text{Phi}}$($A^{(i)}$)

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 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 should 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 modern accelerators.[4] The common technique to alleviate this imbalance is to use *lookahead* [27, 28].

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

```
PanelFactorize(P_1)
PanelSend(P_1)
TrailingMatrixUpdate_{Kepler,Phi}(P_1)
PanelStartReceiving(P_2)
TrailingMatrixUpdate_{Kepler,Phi}(R^(1))
for P_i ∈ {P_2, P_3, ...} do
    PanelReceive(P_i)
    PanelFactorize(P_i)
    PanelSend(P_i)
    TrailingMatrixUpdate_{Kepler,Phi}(P_i)
    PanelStartReceiving(P_i)
    TrailingMatrixUpdate_{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 Direct Acyclic Graph (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 of communicating the stolen task's data across the PCI bus. These caveats are dealt with comprehensively in the remainder of the paper.

5. 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. What we propose, considers both the

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

higher ratio of execution and the hierarchical memory model of the modern accelerators and coprocessors. Taken together, it emerges as a multi-way 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 its modifications that facilitate execution on heterogeneous devices.

5.1. Task Superscalar Scheduling

Task-superscalar execution takes as an input a sequence of tasks in a form of a DAG and schedules them for execution in parallel while honoring the data dependences between the tasks at runtime. The dependences between the tasks are honored through the resolution of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). The dependences that form the DAG's edges are extracted from the serial formulation of the code by having the user include annotations on the data items when defining the tasks. The annotations indicate 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 wish to read the same data, they need not wait on each other but can execute in parallel once the data is available. This is the classical case of Lamport's *concurrent* events [18]. 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 the original code looks as if it was serial it can be written and debugged in a serial environment. After the code becomes the input for the runtime system, the parallel execution avoids all the data hazards and thus preserves the correctness of the serial code, which in turn implies the parallel correctness of execution (granted, of course, that the runtime system does not introduce conditions that violate user's annotations).

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 naturally enables lookahead into the serial code, since the tasks from the DAG representation can be executed as soon as their dependences are satisfied and not necessarily in the order they were inserted into the DAG from the the serial code.

5.2. 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 us to enforce at a low level a precise task placement. QUARK is a data-driven dynamic superscalar runtime environment with a simple API for serial task insertion and parallel execution on multicore processors. It is the dynamic runtime engine used within the PLASMA linear algebra library and has been shown to provide high productivity for code development and good performance on modern multicore processor systems [17, 31, 32].

6. 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.

6.1. 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 exhibit good scalability. From the hardware perspective, 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 (each of the CPUs, GPUs, Xeon Phi is assigned one such capability), the memory access overhead, and the communication cost. As with CPUs, the access time to the device memory for accelerators is slow compared to peak performance (still the accelerator memory tends to be many times faster than the CPU memory). CPUs try to improve the effect of the long memory latency and bandwidth constraint by using hierarchical caches. This does not solve the slow memory problem completely but is often effective in our target factorizations. 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 the past requests. By comparison with CPUs, the device memory delivers higher absolute bandwidth (effectively around 180 GB/s for Xeon Phi and 160 GB/s for Kepler K20c). To side-step the issues related to the memory constraints, we developed 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 accelerators busy, we redesign the kernels and propose dynamically guided data distribution to exploit enough parallelism to keep the accelerators and processors occupied at the same time.

From the programming model point of view, it is hard to hide the distinction between the two kinds of parallel devices. For that, we convert each algorithm into a host part and an accelerator part. Each routine to be 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 the QUARK runtime engine in order to alleviate the programming burden and to simplify scheduling. This often allows us to maintain

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)
```

a single source version that handles different types of accelerators either independently or when mixed together. Our intention is for our model to simplify most of the hardware details, but, at the same time, give us finer levels of control.

Algorithm 5 shows the pseudocode for the Cholesky factorization as an algorithm designer would see it. It consists of a sequential code that is simple to comprehend and independent of the architecture. Each of the 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 results in execution on either a CPU or an accelerator – commonly it would called a *kernel*. We tried to hide the differences between hardware and let 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 prerequisites. Moreover, we implemented a set of 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 more easily develop simple and portable code, that can run on different heterogeneous architecture and lets 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.

6.2. Resource Capability Weights (CW)

There is no simple way to express the difference in the workload-capabilities between the CPUs and accelerators in a generic fashion. Clearly, we cannot balance the load, if we treat them

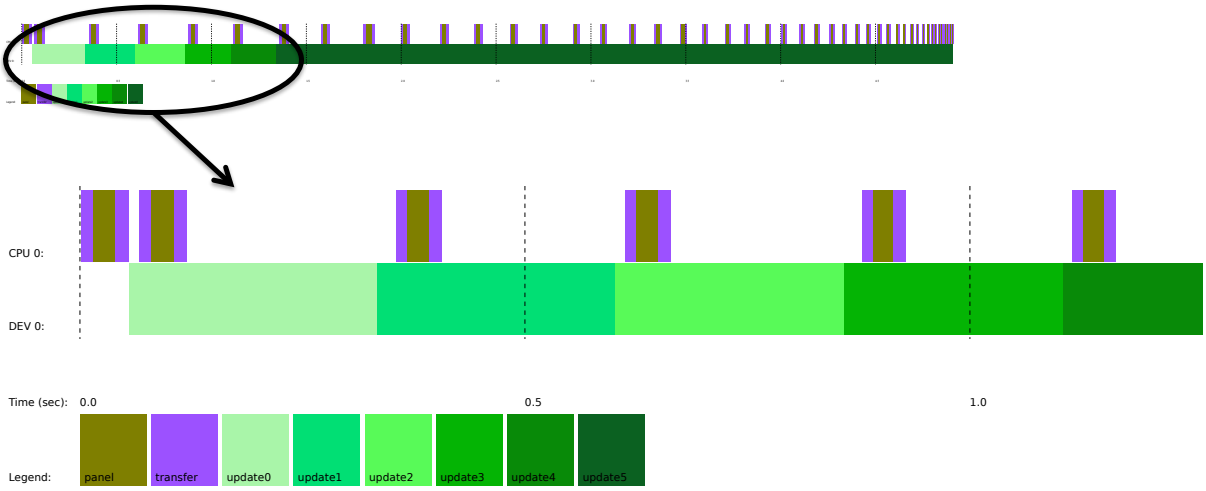

Figure 10. A trace of the Cholesky factorization on a multicore CPU with a single GPU K20c, assigning the panel factorization task to the CPU (brown task) and the update to the GPU (green task) for a matrix of size 20,000.

as similarly equipped peers and assign them equivalent amounts of work. This naïve strategy would cause the accelerator to be idle most of the time. 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, extended version of 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 which indicates that the kernel is either bandwidth-bound or compute-bound function (as shown in Algorithm 5 with a call to Task_Flag_Set with a BLAS2 argument) or estimated according to the volume of data and the elapsed time of a kernel by the QUARK engine at runtime.

Figure 10 shows the execution trace of the Cholesky factorization on a single multicore CPU and a K20c GPU of System A (hardware labels and description is given in Section 7.1). 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, the data 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 dependence 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 a 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 10, 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 simply defined as:

$$\text{communication} = \frac{n \times n}{\text{bandwidth}}$$
$$\text{computation} = n^k \times \alpha_{ik}$$
$$\text{where } k \text{ is the Level } k \text{ BLAS}$$

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

6.3. 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 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 preferred to be executed always on the CPU side. The heuristic tries to maintain the ratios of the capability-weights across all the resources.

In Figure 11, 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 12 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 §7.1).

6.4. Improved Task Priorities

In order to highlight the importance of task priorities, we recall, that the panel factorization task of most of the one-sided factorizations (the Cholesky, QR, and LU decompositions) is on

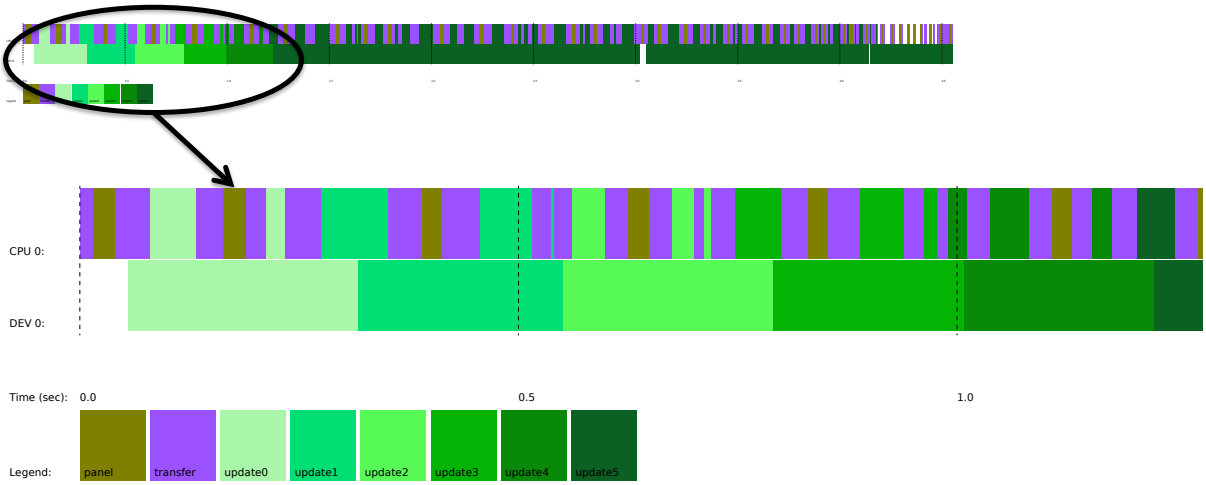

Figure 11. A trace of the Cholesky factorization on 16-core, 2-socket Sandy Bridge CPU and a K20c GPU, using capability-weights to distribute tasks.

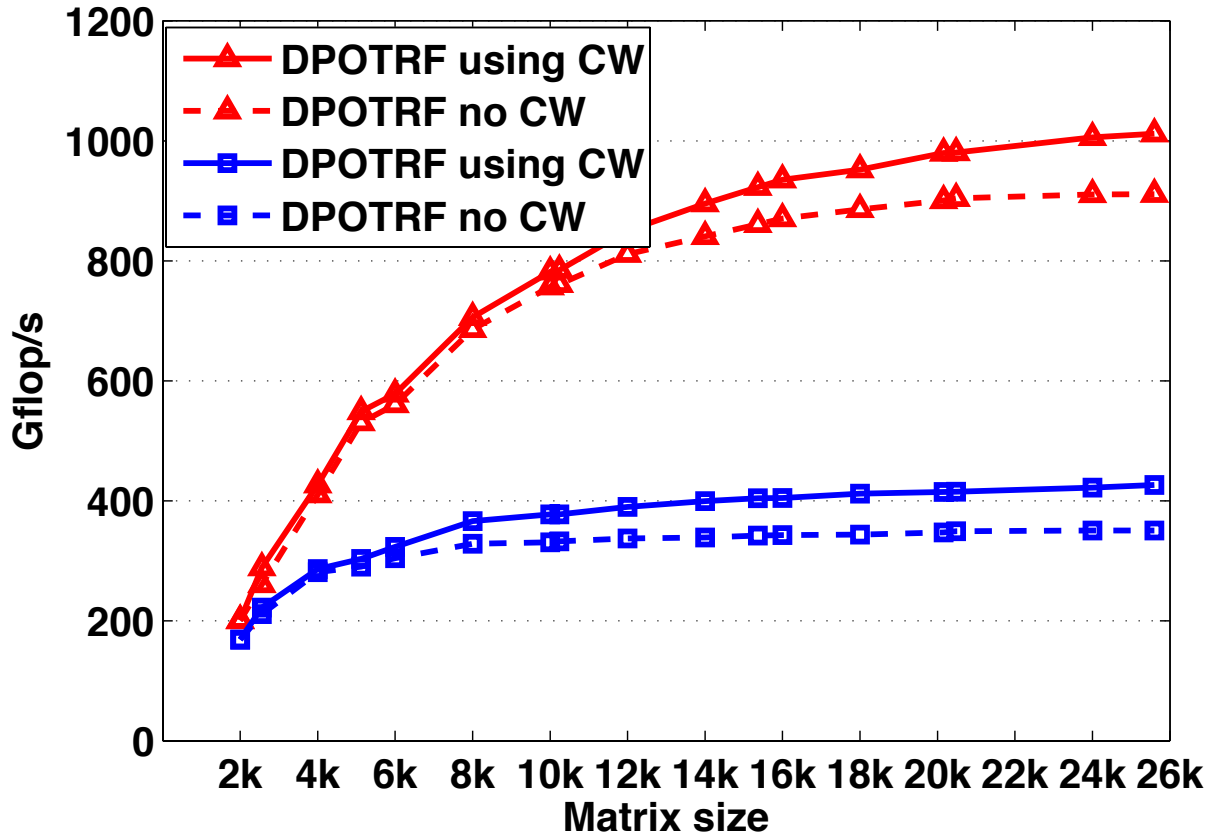

Figure 12. Performance comparison of the Cholesky factorization when using the capability-weights (CW) to distribute tasks among the heterogeneous hardware, on a node of 16 Sandy-Bridge CPU and either a Kepler (K20c) of system A "red curve" or a Fermi(M2090) of system B "blue curve".

the critical path of the execution. In other words, only if a panel computation is done in its entirety, its corresponding update computation (compute-bound operation) may proceed. In the traces in Figures 10 and 11, 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 increases 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 13 shows the update tasks being executed earlier in the trace.

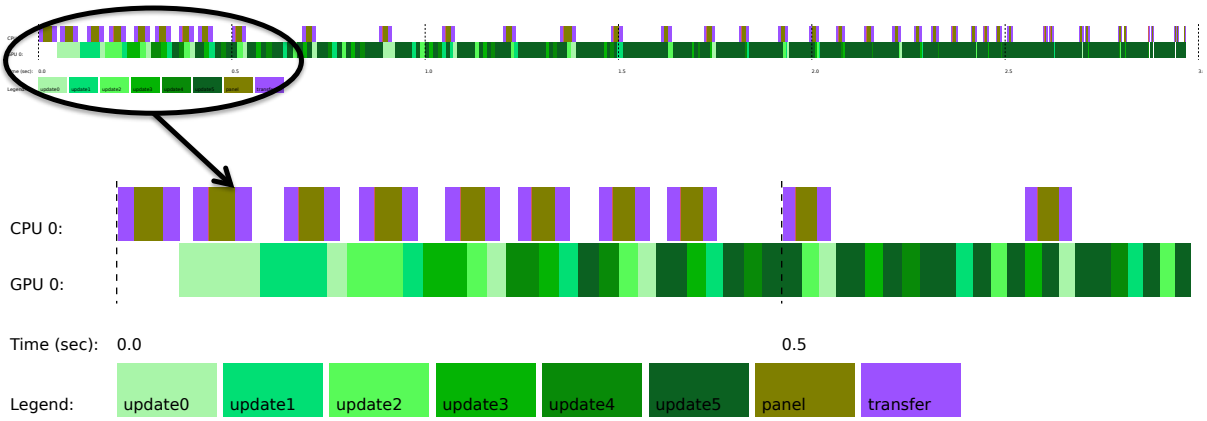

Figure 13. Trace of the Cholesky factorization on multicore 16 Sandy Bridge CPU and a K20c GPU, using priorities to improve lookahead.

6.5. 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.

6.6. Hardware-Guided Data Distribution (HGDD)

The experiments so far showed that the standard 1-D block cyclic data layout was hindering performance in heterogeneous multi-accelerator environments. Figure 14 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 data assignment at runtime.

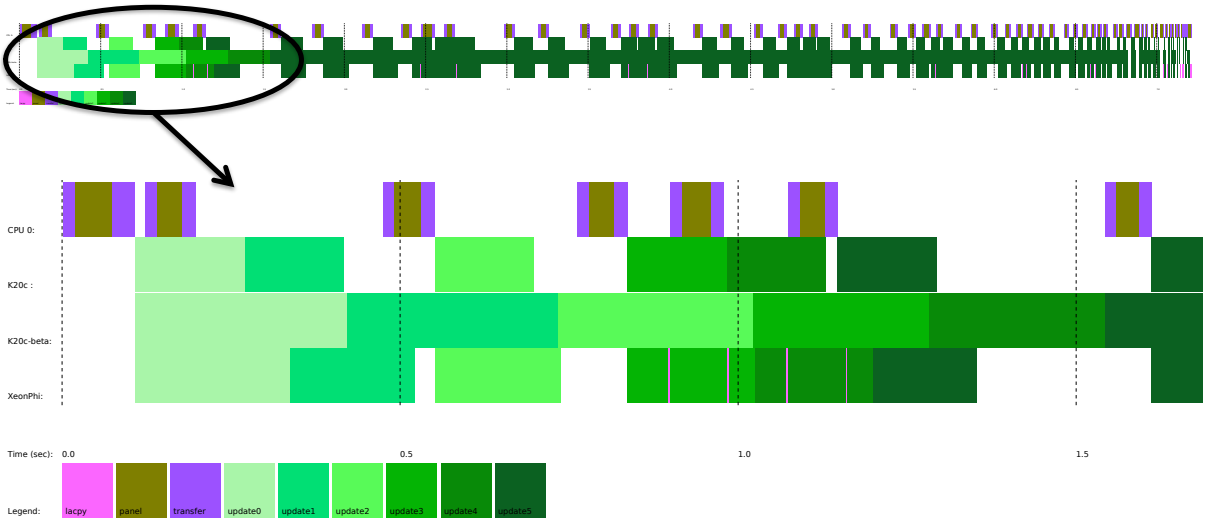

Figure 14. Cholesky factorization trace on multicore CPU and multiple accelerators (a Kepler K20c, a Xeon Phi, and an old K20-beta-release), using 1D block cyclic data distribution without enabling heterogeneous hardware-guided data distribution.

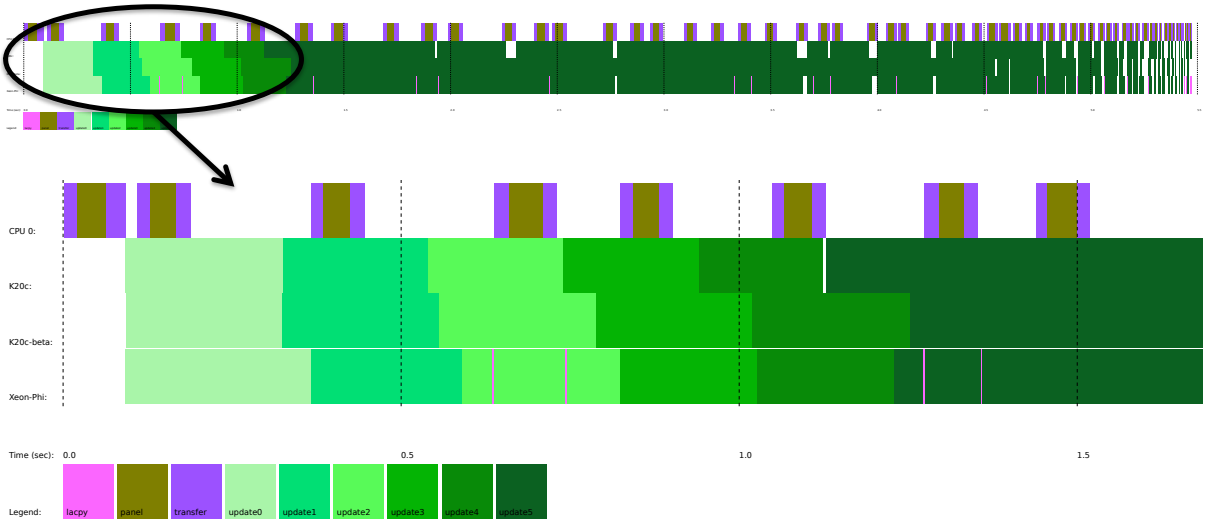

Figure 15. Cholesky factorization trace on multicore CPU and multiple accelerators (a Kepler K20c, a Xeon Phi, and an old K20-beta-release), using the heterogeneous hardware-guided data distribution techniques (HGDD) to achieve higher hardware usage.

Figure 15 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 16 and Figure 17 the performance comparison of the Cholesky factorization and the QR decomposition when using the HGDD strategy. The curves in blue show 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 16 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.

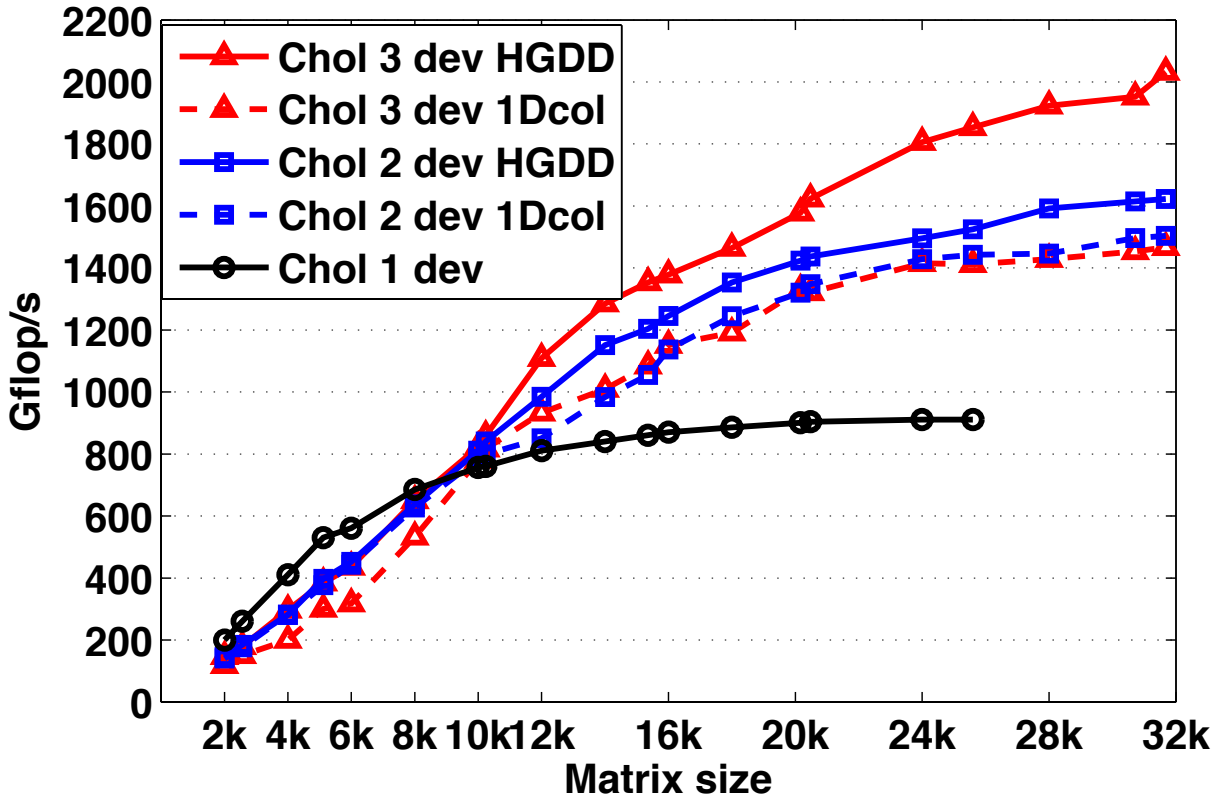

Figure 16. Performance comparison of the Cholesky factorization when using the hardware-guided data distribution techniques versus a 1-D block-column cyclic, on heterogeneous accelerators consisting of a Kepler K20c (1dev), a Xeon Phi (2dev), and an old K20-beta-release (3dev).

6.7. 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 consideration of hardware constraints, the methods of achieving fast communication, and approaches that allow the algorithms to reach good scalability on both CPUs and accelerators. Our target algorithms in this study are the one sided-factorization (Cholesky, QR, and LU).

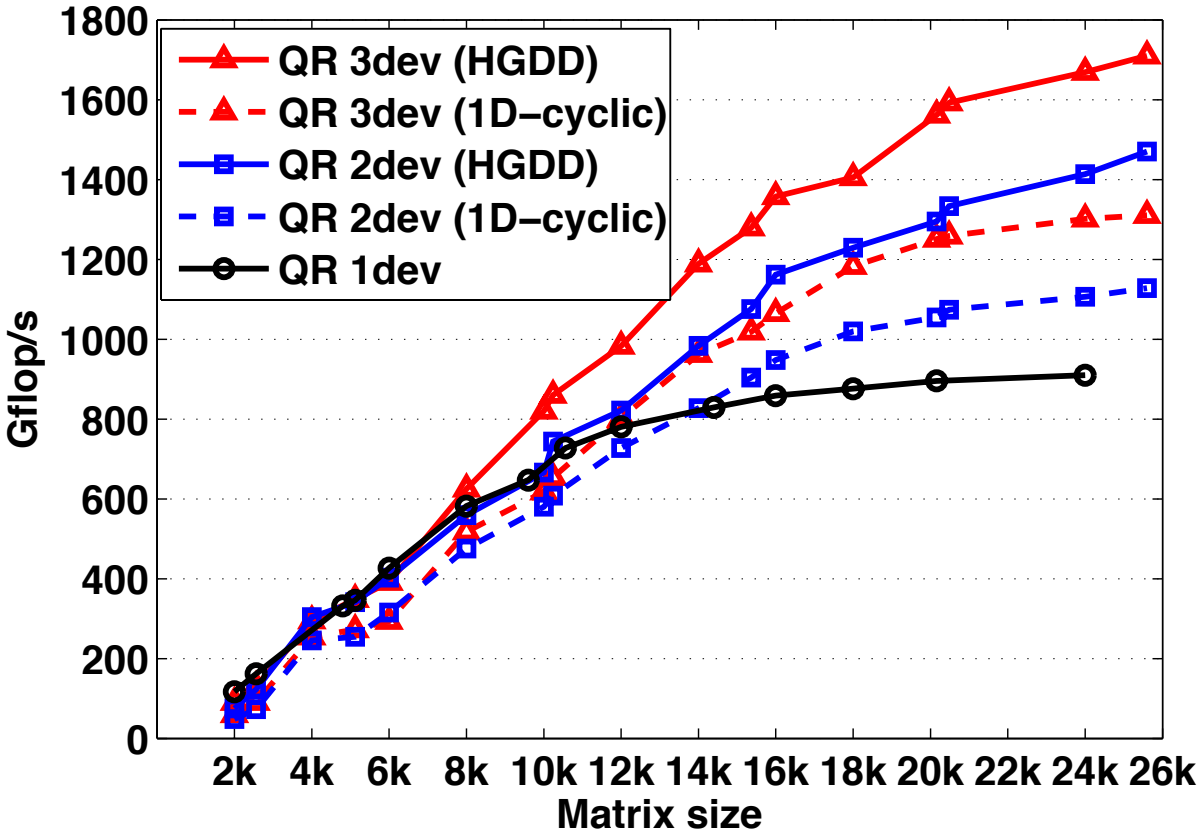

Figure 17. QR performance comparison for hardware-guided data distribution techniques *vs.* 1-D block-column cyclic, on heterogeneous accelerators consisting of a Kepler K20c (1dev), a Xeon Phi (2dev), and an old K20-beta-release (3dev).

6.7.1. Redesigning BLAS Kernels to Exploit Parallelism and Minimize the 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 18.

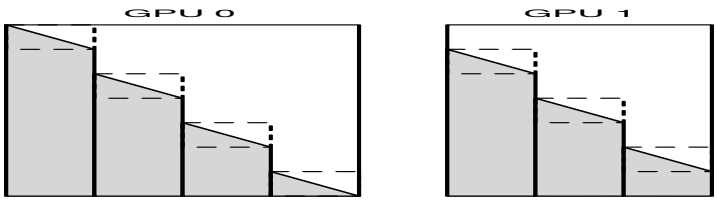

Figure 18. Block-cyclic data distribution. Shaded areas contain valid data. Dashed lines indicate diagonal blocks.

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 up-

date 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.

6.7.2. 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 kernels (GEMM, TRSM) required by the LU factorization. The coalesced reads and writes improve the performance of the LASWP function 1.6 times.

6.7.3. 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.

6.7.4. 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, we replace the TRMM by GEMM for computational efficiency, thus achieving 5-10% higher performance when executing these kernels on the accelerator.

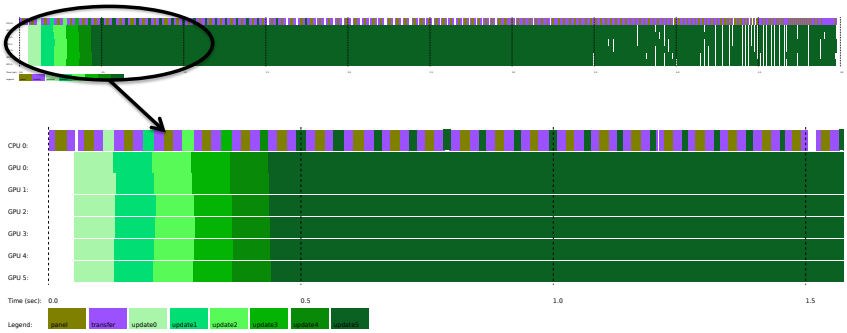

Figure 19. Trace of Cholesky factorization on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

7. Experimental Setup and Results

7.1. 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 processor in a dual-socket configuration with 8-core Intel Xeon E5-2670 (Sandy Bridge) processors in a socket, each running at 2.6 GHz. Each socket had 24 MiB of shared L3 cache, and each core had a private 256 KiB Level 2 and 64 KB Level 1 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 high performance software packages were used for the experiments. On the CPU side, we used the MKL (Math Kernel Library) [15]. On the Xeon Phi side, we used the MPSS 2.1.5889-16 as the 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.

7.2. 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.

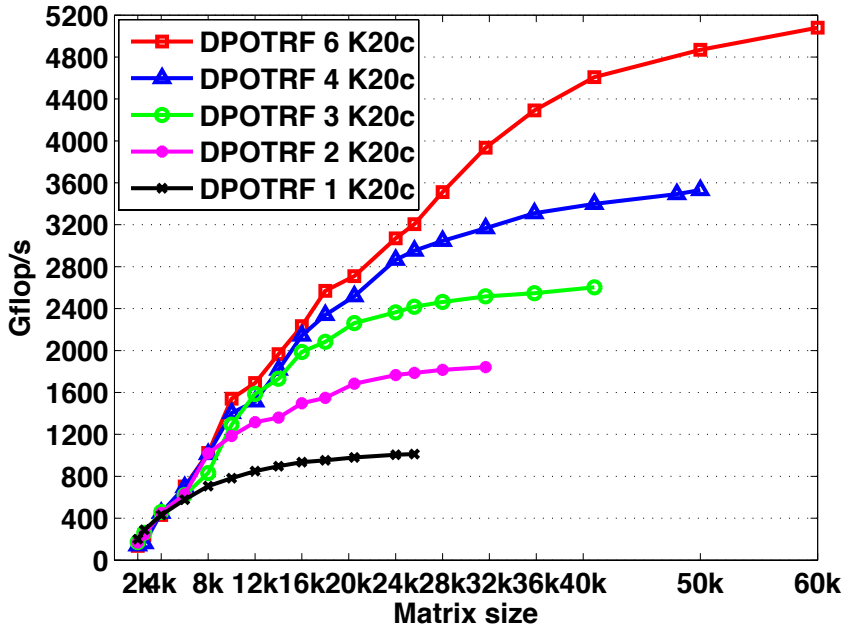

Figure 20. Performance scalability of Cholesky factorization on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

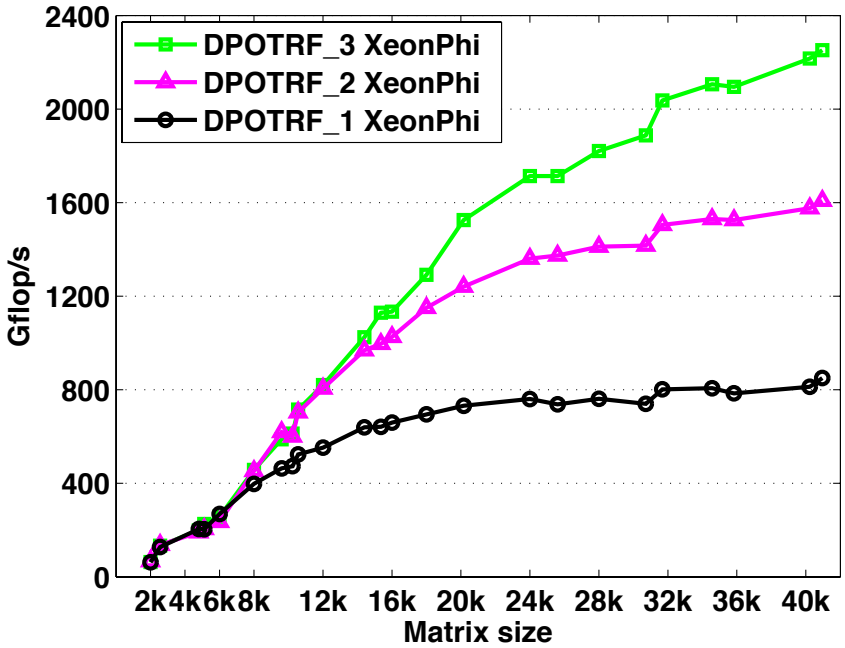

Figure 21. Performance scalability of Cholesky factorization on multicore CPU and multiple accelerators (up to 3 XeonPhi).

Figure 19 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 20 and 21 show the performance scalability of the 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

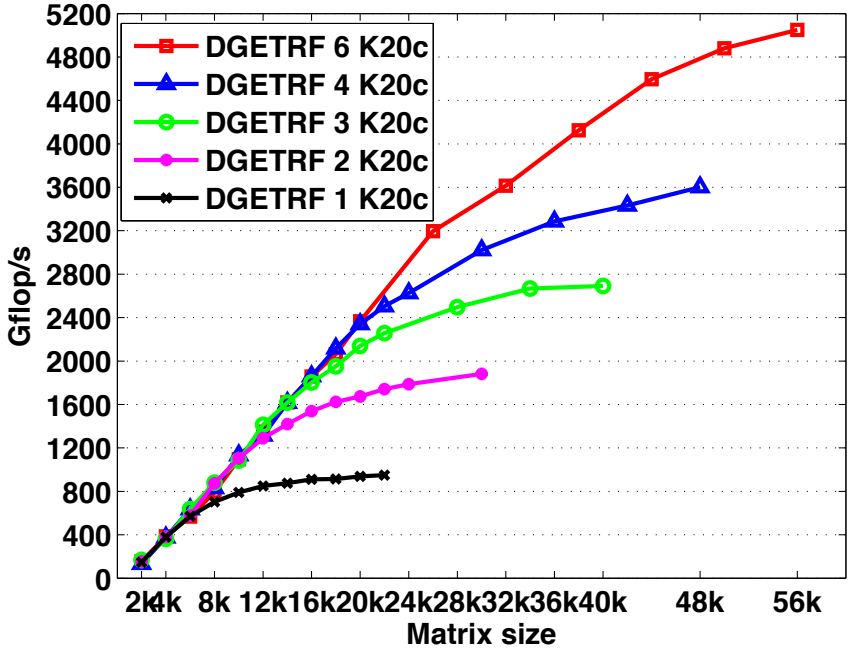

Figure 22. Performance scalability of the LU decomposition on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

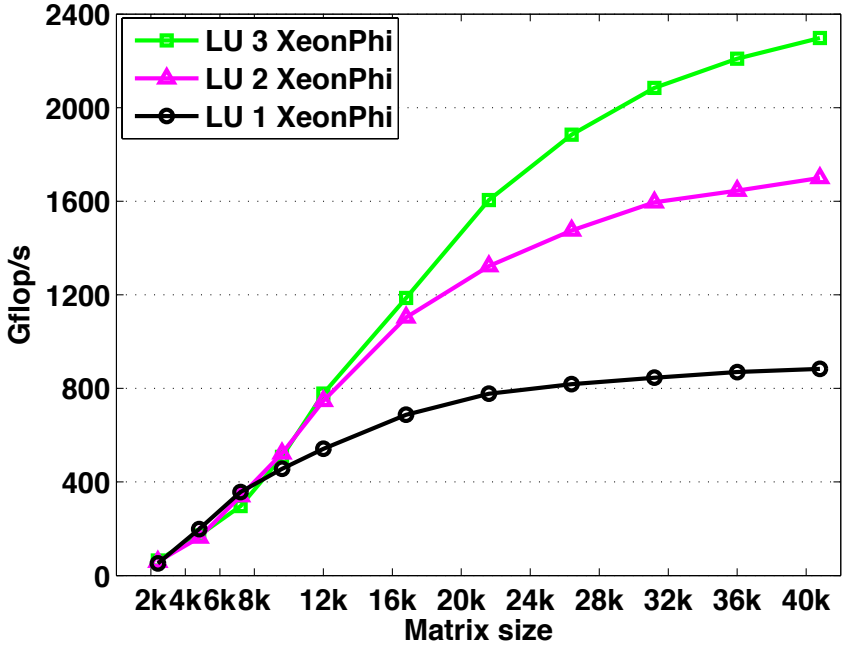

Figure 23. Performance scalability of the LU decomposition on multicore CPU and multiple accelerators (up to 3 Xeon Phi).

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 22 depicts the performance scalability of the LU factorization on System A while Figure 23 shows its performance on System C. Similarly to the Cholesky factorization, the LU factorization achieves around 5.2 Tflop/s on the System A using the 6 Kepler K20c GPUs for a matrix of size 56,000, and it also reaches 2.4 Tflop/s on the

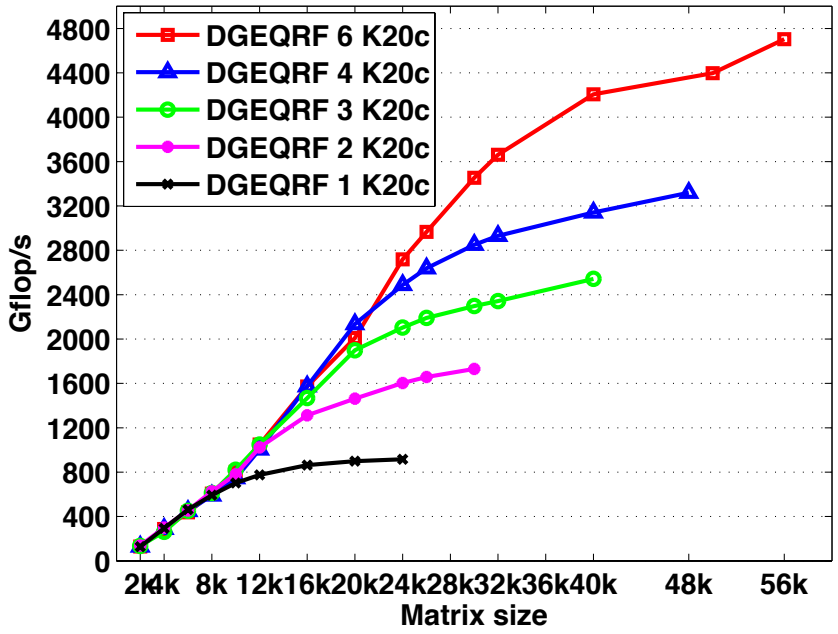

Figure 24. Performance scalability of the QR decomposition on multicore CPU and multiple accelerators (up to 6 Kepler K20c).

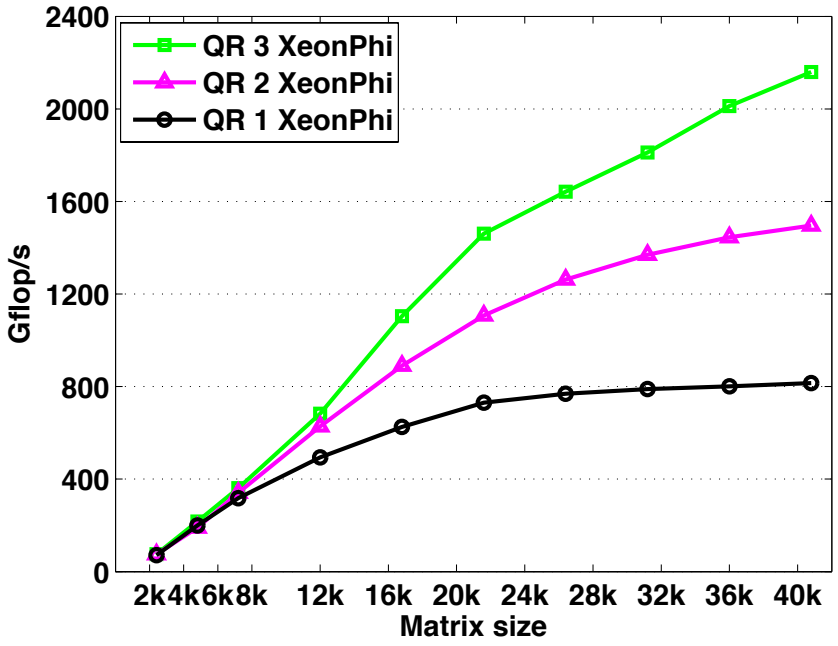

Figure 25. Performance scalability of the QR decomposition on multicore CPU and multiple accelerators (up to 3 Xeon Phi).

System C using the 3 Intel Xeon Phi coprocessors for a matrix of size 40,000. Figure 24 depicts the performance scalability of the QR factorization on System A and Figure 25 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.

8. 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. to employ a programming model that would simplify the development.

We presented initial implementations of two algorithms. Future work will include merging MAGMA's [19] CUDA, OpenCL, and Intel Xeon Phi development branches into a single library using the new programming model.

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. Intel Xeon Phi Coprocessor System Software Developers Guide. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide.

2. 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.

3. Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil 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.

4. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

5. Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek 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.

6. Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

7. A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.

8. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

9. Chongxiao Cao, Jack Dongarra, Peng Du, Mark Gates, Piotr Luszczek, and Stanimire Tomov. clMAGMA: High Performance Dense Linear Algebra with OpenCL. In *International Workshop on OpenCL, IWOCL 2013*, Atlanta, Georgia, USA, May 13-14 2013.

10. Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert 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.

11. NVIDIA CUBLAS library. https://developer.nvidia.com/cublas.

12. Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire 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.

13. Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

14. Carlos H. González and Basilio 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.

15. Intel. Math Kernel Library. http://software.intel.com/intel-mkl/.

16. Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

17. 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.

18. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

19. MAGMA library. http://icl.cs.utk.edu/magma/.

20. 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. http://dx.doi.org/10.1177/1094342010385729 (DOI:~10.1177/1094342010385729).

21. Chris J. Newburn, Rajiv Deodhar, Serguei Dmitriev, Ravi Murty, Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla, and Russell McGuire. Offload compiler runtime for the intel xeon phitm coprocessor. In *ISC*, pages 239–254, 2013.

22. Josep M. Pérez, Rosa M. Badia, and Jesús 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.

23. 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 (DOI:~10.1109/2.214440).

24. J. E. Rodrigues. A graph model for parallel computations. Technical Report MIT/LCS/TR-64, MIT, Cambridge, MA, USA, September 1969.

25. Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis 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.

26. Fengguang Song, Stanimire Tomov, and Jack 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.

27. Peter 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.

28. Peter E. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *Int. J. Parallel Distrib. Systems Networks*, 4(1):26–35, 2001.

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

30. 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. http://dx.doi.org/10.1145/1413370.1413402 (DOI:~10.1145/1413370.1413402).

31. Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, December 2012.

32. Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.

Received June 4, 2014.