POMPEI: Programming with OpenMP4 for Exascale Investigations[*]

Jack Dongarra[1,2,3], Azzam Haidar[1], Oscar Hernandez[2], Stanimire Tomov[1], and Manjunath Gorentla Venkata[2]

[1] University of Tennessee,
Knoxville, TN, USA
dongarra,haidar,tomov@eecs.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, TN, USA.
oscar,manjugv@ornl.gov
[3] University of Manchester, Manchester, U.K.

Abstract

The objective of the **P**rogramming with **O**pen**MP**4 for **E**xascale **I**nvestigations (POMPEI) project is to explore new task-based programming techniques together with data structure centric programming for scientific applications to harness the potential of extreme-scale systems. Tasking is a well established by now approach on such systems as it has been used successfully to handle their large-scale parallelism and heterogeneity, which are leading challenges on the way to exascale computing. The approach is to harness the latest features of OpenMP4.5 and OpenACC2.5 to design abstractions shared among tasks and mapped efficiently to data-structure driven programming paradigms. This technical report describes the approach, along with its reference implementation and results for dense linear algebra algorithms.

1 Introduction

To meet the challenges of billion-way parallelism at exascale, it is widely agreed that math libraries should, wherever possible, embrace event-driven and message-driven execution models, in which work is abstracted in the form of asynchronous tasks, whose completion can trigger additional computation tasks and data movements. Although this model does not work for all application areas in HPC, it is very suitable for implementing dense linear algebra (DLA) workloads, which have historically been on the leading edge of the communitys effort to address major changes in the hardware ecosystem. Therefore, we propose to build on tools established in the area of DLA, by enhancing them with a number of new techniques and abstractions to meet the HPC needs of other scientific applications of interest. In particular, we propose to concentrate on the adoption of the new OpenMP4 standard in DLA libraries, while exploring portable high-level data structure programming APIs in conjunction with the SharP LDRD project [12, 13]. OpenMP4 offers task scheduling constructs. Their implementation in the GNU compiler suite has illustrated advantages against other, custom build schedulers (like QUARK and PaRSEC from UTK, SMPSs from the Barcelona Supercomputer Center, and StarPU from INRIA), further raising the interest in the use and adoption of task scheduling approaches in numerical libraries.

Handling heterogeneity is the other leading challenge for numerical libraries. Libraries need to run on a wide range of systems, supporting multiple processor architectures and multiple memory systems. Vendors provide their own systems for offloading work to accelerators, e.g., CUDA (NVIDIA), OpenCL (AMD), and Intel compiler offload features (Intel), which makes the simultaneous support of various accelerators challenging. To evolve toward exascale-friendly programming approaches we will explore

[*]POMPEI Technical Progress Report for the time period of 07/01/2016 – 09/30/2016.

the use of the OpenACC standard in numerical libraries. The Oak Ridge Leadership Computing Facility (OLCF) has made a strategic investment in OpenACC for the Titan system and applications are starting to use it. Therefore, we propose to develop key extensions to OpenACC/OpenMP to access different types of memory hierarchies and support selected applications and demonstrate their efficiency with prototype implementations.

This technical report presents our POMPEI proposal on how to address the above two challanges, taking case studies and examples of fundamental dense linear algebra algorithms. To this end, we report the technical details on POMPEI's:

Study, analysis, and benchmarks of using OpenMP4.5 and data abstractions for dense linear algebra;

Framework design for POMPEI outlining a dense linear algebra programming approach that uses OpenMP4.5 tasking and SharP data structure programing paradigms that enable ease of development and seamless portability across multicore CPUs, Nvidia GPUs, and Intel Xeon Phi;

Prototype Cholesky factorization for a single node heterogeneous system using the POMPEI framework with OpenMP4.5 tasking and prototype data abstractions and APIs.

2 Study, analysis, and benchmarks

2.1 Directives-based programming model

Directives-based parallel programming is the de-facto standard for shared memory systems. It has a widespread vendor support and a large user base. Implementations, e.g., of the OpenMP standards, provide a set of directives to manage, synchronize, and assign work to threads that share data. Adding support for tasking and offload of work to accelerators in OpenMP, made directives-based programming models based on OpenMP even more attractive. This also motivates our interest in developing directives-based programming models for dense linear algebra (DLA) on heterogeneous systems.

DLA programming in libraries like in LAPACK [1], MAGMA [2], and others is based on expressing algorithms in terms of BLAS [3] calls. Subsequently, implementation can achieve high efficiency, provided that highly efficient machine-specific BLAS implementations are provided, e.g., by the manufacturer. These current best practices drive our ideas towards the development for the next generation of parallel programming model – one that will be based on BLAS-based tasking, directives, and dynamic scheduling of work to heterogeneous processing elements.

2.2 OpenMP and OpenACC

The OpenMP community has been swiftly moving forward with standardization of new scheduling techniques for multicores. First, the OpenMP 3.0 standard adopted the Cilk scheduling model, then the OpenMP 4.0 standard adopted the superscalar scheduling model. Not without significance is the fact that the GNU compiler suite was also quick to follow with high quality implementations of the new extensions. We find that these new features are essential for DLA libraries. Since DLA is fundamental to a wide range of scientific and engineering applications, by concentrating on developing best practices in parallel programming models for DLA on heterogeneous architectures, we address indirectly other areas as well.

OpenACC is another specification focused on directive-based ways to program accelerators.

2.3 PAPI-based analysis

We use the Performance Application Programming Interface (PAPI) [4] to get a consistent interface access to the performance counter hardware found in most major microprocessors. Counter data is needed for performance monitoring, application analysis, and optimizations.

2.4 BLAS benchmarks

To study the performance portability of accelerator directives provided by OpenMP 4 and OpenACC, we developed benchmarks with various computational intensities. In particular, we studied DLA as it is well represented on most architectures and many applications use it. We chose kernels of increasing computational intensity – daxpy, dgemv, and dgemm – which are representative of the Level-1, Level-2, and Level-3 BLAS, respectively [5].

2.4.1 daxpy

The daxpy kernel computes $y \leftarrow y + \alpha x$ for vectors x and y and scalar α.

Listing 1 shows the code that we developed for the daxpy benchmark using OpenMP 4 directives. The OpenACC version is similarly based on the equivalent OpenACC syntax. The code consists of a data region specifying arrays to transfer to and from the accelerator and a parallel region directing execution of the daxpy loop to the device. For the OpenMP 4.5 version we did not specify an OpenMP SIMD directive in the inner loop since this vectorization pattern was recognized by all the tested compilers (Intel, PGI and Cray).

```
double alpha, *x, *y;
int n;
#pragma omp target data map(to:x[0:n]) &
  map(tofrom:y[0:n])
{
  int i;
  #pragma omp target teams
  {
    #pragma omp distribute parallel for
    for (i=0; i<n; i++)
      y[i] += alpha * x[i];
  }  // teams
} // data
```

Listing 1: OpenMP 4/OpenACC version of daxpy

2.4.2 dgemv

The dgemv kernel computes $y \leftarrow \beta y + \alpha A x$ or alternatively, $y \leftarrow \beta y + \alpha A^T x$ for matrix A, vectors x and y, and scalars α and β. These are referred to as the non-transpose ("N") and transpose ("T") forms, respectively.

Listing 2 shows the code for the dgemv operation, N case. The code has a similar structure including a data region but also several loops including a doubly nested loop and if statement. Additionally, the non-stride-1 access poses a challenge for compilation to efficient code.

```
double alpha, beta, *x, *y, *A;
int m, n;
#pragma omp target data map(to:A[0:m*n]) &
  map(to:x[0:n]) map(tofrom:y[0:m]) &
  map(alloc:tmp[0:m])
{
  int i, j;
  double prod, xval;
  #pragma omp target teams
  {
    #pragma omp distribute parallel for &
      private(prod, xval, j)
    for (i=0; i< m; i++) {
      prod = 0.0;
      for (j=0; j< n; j++)
        prod += A[i+m*j]*x[j];
      tmp[i] = alpha * prod;
    }
    if (beta == 0.0) {
      #pragma omp distribute parallel for
      for (i=0; i<m; i++)
         y[i] = tmp[i];
    } else {
      #pragma omp distribute parallel for
      for (i=0; i<m; i++)
         y[i] = beta * y[i] + tmp[i];
    } // if
  } // teams
} // data
```

Listing 2: OpenMP4 version of dgemv/N

2.4.3 dgemm

The dgemm kernel computes $C \leftarrow \alpha A^{[T]} B^{[T]} + \beta C$ for matrices A, B, and C, and scalars α and β. Based on the transpose ("T") or not ("N") options for A and B, we recognize four versions – "NN", "TN", "NT", and "TT" – where the first character refers to the T/N option for A, and the second is for B. We developed prototypes and autotuning strategies for this kernel, following the approach above and GEMM designs for GPUs [6]. The developments are described in detail in [7].

2.5 Cholesky factorization

The LAPACK routine for the Cholesky factorization of a real (double precision) symmetric positive definite matrix A is dpotrf. Following LAPACK's convention, the Cholesky factorization of A has the form $A = U^T U$, if the symmetric matrix is specified in the upper triangular part of A, or alternatively $A = LL^T$, if A is given in the lower-triangular part of A. Below are three implementations that we consider; all implementations are for the $A = LL^T$ case.

2.5.1 Reference implementation

Listing 3 shows a reference pseudo-code implementation for the Cholesky factorization of an m-by-m matrix A. Using LAPACK's convention, we refer to this code as potf2.

```
for (j=0; j< m; j++){
    // Update the current diagonal element (Level-1 BLAS ddot)
    ajj = a(j,j);
    for (i=0; i<j; i++)
       ajj -= a(j,i)^2;

    // Factor the current diagonal element
    a(j,j) = sqrt( ajj )

    // Update current column (Level-2 BLAS dgemv followed by scaling)
    a(j+1:m , j) -= a(j+1:m, 0:j) a(j, 0:j)^T;
    a(j+1:m , j) *= 1.0/a(j,j);
}
```

Listing 3: Reference implementation of $A = LL^T$ factorization (potf2)

2.5.2 Blocked implementation

Listing 4 shows the LAPACK's blocked implementation for the potrf of an m-by-m matrix A.

```
for (j=0; j< m; j+=nb){
    // Adjust the block size for the end block
    jb = min(nb, m-j);

    // Update the current diagonal block (Level-3 BLAS syrk)
    a(j:j+jb,j:j+jb) -= a(j:j+jb,0:j) * a(j:j+jb,0:j)^T;

    // Factor the current diagonal block (e.g., using Listing 1)
    potf2( 'Lower', jb, a(j,j), lda, info);

    // Update current block column (Level-3 BLAS gemm followed by trsm)
    a(j+jb:m , j:j+jb) = a(j+jb:m,0:j    ) * a(j:j+jb,0:j    )^T;
    a(j+jb:m , j:j+jb) = a(j+jb:m,j:j+jb) * a(j:j+jb,j:j+jb)^{-T};
}
```

Listing 4: Blocked implementation of $A = LL^T$ factorization (potrf)

Here a(j:j+jb , j:j+jb)^ {-T} is not computed explicitly. Instead, we use LAPACK's trsm routine to solve for X the system $X\ a(j : j+jb, j : j+jb)^T = B$, where X is $a(j+1 : m, j : j+jb)$, B is $a(j+1 : m, j : j+jb)$, and $a(j : j+jb, j : j+jb)$ is the lower triangular matrix resulting from the potf2 Cholesky factorization of the current diagonal block.

2.5.3 Blocked implementation with tasks

The main building blocks of the implementation in Listing 4 – the diagonal block update, diagonal block factorization, update of the column block, and its scaling – can be defined as dsyrk, potf2, dgemm, and dtrsm tasks, respectively. Using the OpenMP tasking mechanism, we describe the tasks along with their inputs and outputs, as given in Listing 5, and leave it to OpenMP to schedule the execution without violating the data dependencies. The sequential description of the algorithm is known as left-looking, as going and factorizing the matrix from left to right, only the current panel is updated using information on its left. Studying the data dependencies, one can see that the computation can be organized in other ways, e.g., right-looking where the transformations needed for the current panel factorization are applied immediately to the right of the matrix, or top-looking where only the top left sub-matrix is used for the update of next block row and corresponding diagonal block factorization. Note that using a dynamic scheduling mechanism, e.g., based on the OpenMP tasking, one can describe the algorithm in sequential fashion, e.g., right-looking, but the execution be data driven and in general not follow the order of the sequential description. Listing 5 shows an implementation that features large tasks and a lookahead strategy, as in MAGMA, and is a simple preview to the approach that we propose in Section 3.

```
// Pointers used for dependencies
double *pnew, *pold = A(0,0), *p;

#pragma omp parallel
#pragma omp master
for (j=0; j< m; j+=nb){
    // Adjust the block size for the end block
    jb = min(nb, m-j);

    // Update the current diagonal block (Level-3 BLAS syrk)
    pnew  = A(j,j);
    #pragma omp task firstprivate(jb, j, pold, pnew)     \
                     depend(   in:pold )                 \
                     depend(   out:pnew )
    dsyrk('Lower','No trans', jb, j, -1, a(j,0),lda, 1, a(j,j),lda);
```

```
      // Factor the current diagonal block (e.g., using Listing 1)
      #pragma omp task firstprivate(jb,j, pnew)              \
                       depend( inout:pnew )
      potf2('Lower', jb, a(j,j), lda, info);

      // Update current block column (Level-3 BLAS gemm followed by trsm)
      p = A(j+jb,j);
      #pragma omp task firstprivate(jb,j, p, pold)           \
                       depend(    in:pold)                   \
                       depend(   out:p )
      dgemm('No trans', 'Trans', m-j-jb, jb, j,
            -1, a(j+jb,0),lda, a(j,0),lda, 1, a(j+jb,j), lda);

      #pragma omp task firstprivate(jb,j,p,pnew)             \
                       depend(    in:p)                      \
                       depend( inout:pnew )
      dtrsm('Right', 'Lower', 'Transpose', 'Non-unit',
            m-j-jb, jb, 1, a(j,j),lda, a(j+jb,j),lda);

      pold = pnew;
  }
```

Listing 5: Blocked implementation with OpenMP 4 tasks of the $A = LL^T$ factorization (dpotrf).

Also note that we give the dependencies as discrete values without range. Currently, a range can be specified as a contiguous space, i.e., a dependency on a (non-contiguous) submatrix can not be represented using a leading dimension (lda), as typically done in DLA. This is not a problem for our model, as we will discuss further below, because we can still use OpenMP to handle the dependencies, while ranges and data movements (e.g., between CPUs and GPUs) will be handled by other abstractions.

3 Framework design

Making an algorithm work with heterogeneous hardware components can be challenging. We designed a framework along with a programming model that provide high-level abstractions for programming multi-way heterogeneous resources. The goal is to have a unified approach and a seamless porting of algorithms across a wide range of hardware. We use: 1) algorithmic blocking techniques, 2) $1-D$ *block-column* and $2-D$ data distributions guided by hardware-capabilities-weight, and 3) optimized kernels that enable ease of programming as well as efficiency on a wide range of architectures through use of building blocks. To be fast, reliable, and efficient, we take advantage of OpenMP 4 that allows us to write serial code while extracting parallelism and enabling adaptive execution on the available resources.

This work builds on our earlier work on programming models for DLA [8] on multi-way node-heterogeneous [9] and distributed [10] architectures. We extend the ideas into a prototype framework for DLA that can handle new architectures through a single code.

3.1 OpenMP 4.5 tasking

Here we present the linear algebra aspects of our generic solution for development of the one-sided factorizations: Cholesky (Chol), Gaussian (LU), and the Householder QR. Algorithmically, as presented in Algorithm 1 and illustrated in Figure 1, these factorizations can be viewed as a sequence of steps with two distinct phases per step: 1) a panel factorization that affects the data depicted by the blue portion of Figure 1, and, 2) a trailing matrix update that updates data represented by the magenta and green color in Figure 1. Table 1 shows the BLAS and LAPACK routines that must be substituted for the generic routines named in the algorithm. From a hardware standpoint, the increased computational capability

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

Algorithm 1: Two-phase matrix factorization.

	Cholesky	Householder	Gaussian
`PanelFactorize`	xPOTF2 xTRSM	xGEQF2 xLARFT	xGETF2 xLASWP
`TrailingMatrixUpdate`	xSYRK xGEMM	xLARFB	xTRSM xGEMM

Table 1: Panel factorization and trailing matrix update routines.

requires an incredible increase in the amount of concurrency that a software must be able to utilize. Our design achieves this through a *Master-Devices* approach that we describe in the next section. From a software point of view, we know that from the Table 1 routines, the `PanelFactorize` is memory-bound, while the `TrailingMatrixUpdate` is compute-bound. Thus, one can expect inefficiency of the simple loop of Algorithm 1 due to the nature of the `PanelFactorize` phase. As a consequence, the algorithm must be modified further in order to overcome this issue and to achieve closer to optimal performance. The first optimization is to hide the inefficiency of the memory-bound task (e.g., the `PanelFactorize` phase). A common technique to achieve this is *lookahead*, where the update phase of step i is split into an update of the next panel $UnextP$ (the first green block from the left of Figure 1), and an update of the rest of the trailing matrix $Urest$ (the green blocks to the right). Thus, once the update of the next panel is done, its `PanelFactorize` phase of step $i+1$ can start in parallel with the update of the rest of the trailing matrix $Urest$, hiding its memory-bound behavior. The Multi-way Heterogeneous Programing model described in the next subsection provides an easy way to perform these two operations in parallel.

The efficient use of multiple computational devices for Algorithm 1 strongly depends on the data distribution of the matrix. The `PanelFactorize` phase operates on a *block-column* of data (cf., the blue blocks of Figure 1), and thus it is preferable for its data to be located in the same memory space in

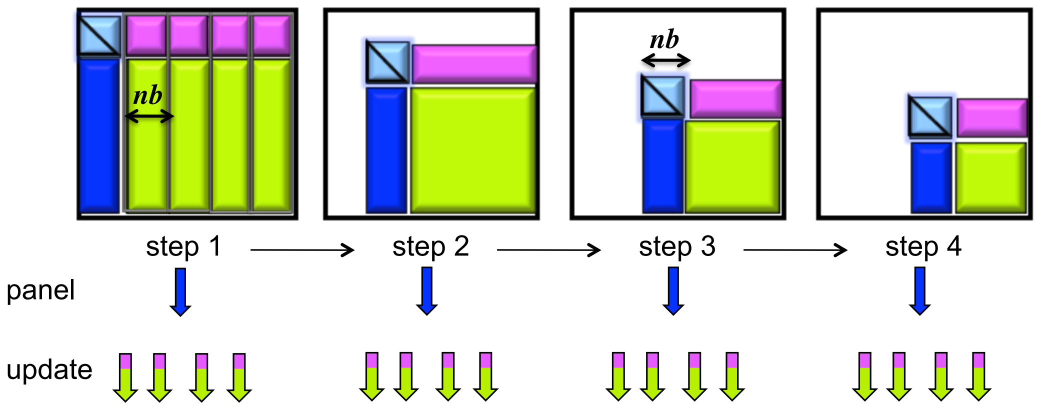


Figure 1: Two-phase implementation of a one-sided factorization.

order to avoid communications, and to increase data locality, and cache reuse. The compute-intensive `TrailingMatrixUpdate` phase requires data from the panel and the top block of *nb* rows, and in the QR case a summation over the columns of the trailing matrix, which makes a *block-row* data distribution a bad candidate. For example, the xGEMM routine of the LU update (green blocks of Figure 1) requires data from the output of the xTRSM magenta blocks of Figure 1. We easily deduce that a *block-column* data distribution is preferable for both phases. Thus, based on the analysis of all the routines described in Table 1, we concluded that an optimal distribution that minimizes the communications is a $1D$ *block-column* data distribution. Note that in contrast to the $2D$ data distribution, well known from the distributed memory ScaLAPACK, here we are in shared memory, and the number of targeted devices is in general less than 10. It turns out that the benefits of a $2D$ distribution (to keep load balance throughout the computation) cannot overcome the overheads of the extra communications and synchronizations associated with it on shared memory systems. This is illustrated by the performance experiments in Figures 4 and 5. We see that two KNCs (purple curves) reach twice the performance of one KNC (cyan curve), i.e., a perfect scalability, and performance is close to the theoretical peak. This shows that the optimization techniques cited above are well implemented, and that the panel computation and the CPU-KNC communications are overlapped with the trailing matrix update phase. More details are provided in Section 5. We propose a $1D$ hardware-capabilities-weight *block-column* distribution, which distributes the data across the devices based on their computing capabilities (in a $1D$ *block-column* fashion). For more details about the hardware-capabilities-weight distribution, we refer to [9].

3.2 Programming Multi-way Heterogeneous Resources

Developing software that properly maps algorithmic requirements to the specific strengths of the hardware components requires the design of heterogeneous algorithms. Xeon Phi and GPUs have high computational peaks compared to multicore CPUs. The difference in capabilities makes it challenging to develop a portable algorithm that can achieve high performance, reach good scalability in a multi-way heterogeneous environment, while also being easy to use, modify, and optimize. For example, computations on the critical path of an algorithm (mainly memory-bound operations like the `PanelFactorize` phase) may be more suitable to run on a small number of cores – the well known way is to run it on multicore CPUs – than on high-throughput computing devices such as GPUs or Xeon Phis, which are more suitable for highly-parallel computations as in the `TrailingMatrixUpdate` phase. This is what we call hybrid mode – where CPUs work together with accelerators. We have demonstrated the methodology of developing these types of hybrid algorithms in the MAGMA Library [9].

Our goal is to design a fremework for this approach that is based on OpenMP and supports multiple architectures, such as multicore CPUs, GPUs, and Xeon Phi, including the recent self-hosted KNL, through a single code. One of the abstractions of the framework is of a virtual view of the hardware as a Master(or Host) computing unit with low capability that is responsible for tasks that are memory-bound, and other Device units (or workers) that are for compute-intensive tasks, as illustrated in Figure 2. This way, we can represent any native mode as a virtual hybrid mode even within the same hardware [11]. For example, a KNL with 64 cores can be viewed as a Master using 4 cores, and one Device using 60 cores. Another possible configuration is a Master using 4 cores along with 6 Devices using 10 cores each, etc. Hence, the techniques proposed in Section 3.1 can be easily developed now using this Master-Device design. For the rest of the this report, we consider any hardware as two compute units, a **Master** unit and one, or many, **Device** units, independently from the associated hardware.

The key features taken into account by our model are the capabilities of the computational resources, the memory access, and the communication cost. We have developed a strategy that prioritizes the data-intensive operations to be executed by the Devices and the memory-bound ones by the Master. Moreover, we redesigned the kernels and implemented dynamically guided data distribution to exploit

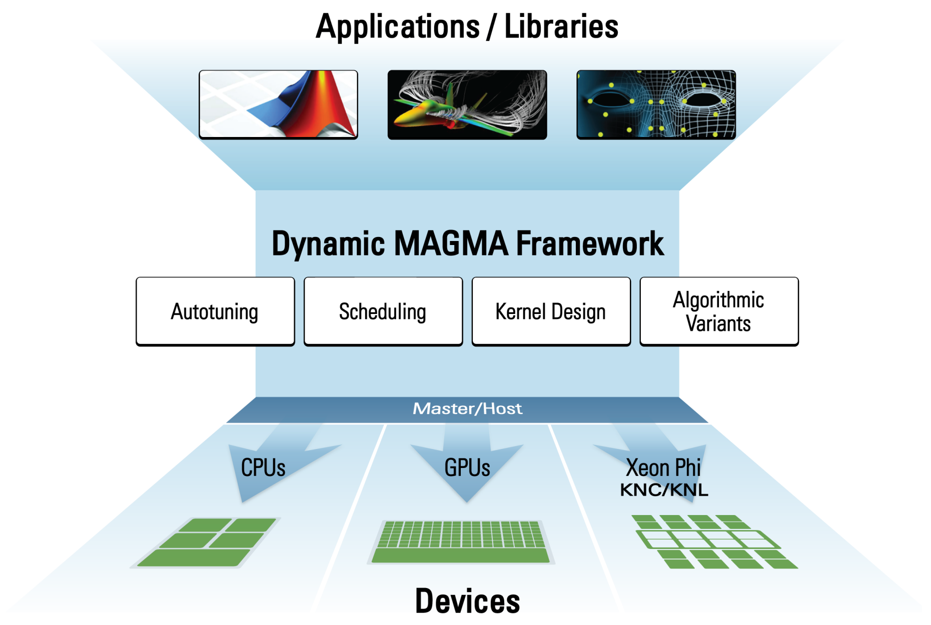


Figure 2: The design of our Unified Programming Model.

parallelism in order to keep the Devices busy. From a programming model point of view, each algorithm is converted into a Master part and a Device part. The routines destined to execute on the Devices must be extracted into a separate hardware-specific kernel function. The kernels may need to be optimized for the Device, e.g., including unrolling loops, replacing some memory-bound operations with compute-intensive ones even if it has a marginal extra cost, and also arranging operations to use the Device memory efficiently. The Master must manage the Device memory allocation, the Master-Device data movement, and the kernel invocation. We used a 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 hardware either independently, or mixed together. Our goal is to abstract hardware details, while still maintaining fine levels of control.

Algorithm 2 shows the pseudocode for the LU factorization. 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 Master or a Device function. We tried to hide the differences between hardware and to allow the scheduler engine to handle the transfer of data asynchronously and automatically, when needed (meaning when Master and Device do not share the same memory). We have proposed a set of scheduling directives (such as DEVTYPE, PRIORITY, and OPTYPE flags) 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. DEVTYPE specify the type of of the device, OPTYPE specify the operation type (e.g., memory-bound BLAS2 or compute-intensive BLAS3) while PRIORITY specify the priority of the task. 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 these functionalities is the implementation of the lookahead technique that does not requires

```
Task_Flags panel_flags = Task_Flags_Initializer ;
Task_Flags update_flags = Task_Flags_Initializer ;
$\boxed{\text{Set the priority of the panel task}}$ ;
TaskFlagSet(&panel_flags, PRIORITY, 10) ;
$\boxed{\text{Panel is memory-bound} \rightarrow \text{locked to Master and disable task stealing}}$ ;
TaskFlagSet(&panel_flags, OPTYPE, BLAS2, DEVTYPE, Master) ;
$\boxed{\text{Update is compute-intensive} \rightarrow \text{preference to Device}}$ ;
TaskFlagSet(&update_flags, OPTYPE, BLAS3, DEVTYPE, Device) ;
for $k \in \{0, nb, 2 \times nb, \dots, n\}$ do
    $\boxed{\text{Factorization of the panel A(k:n,k:k+nb)}}$ ;
    TASK: getf2(A(k:n,k:k+nb)) ;
    $\boxed{\text{Swap the rows to the left and the right of the panel}}$;
    TASK: laswp(A(k:n,1:k)) ;
    TASK: laswp(A(k:n,k+nb:n)) ;
    for $j \in \{k+nb, k+2nb, \dots, n-nb\}$ do
        if $j = k+nb$ then
            TaskFlagSet(&update_flags, PRIORITY, 10) ;
        TASK: trsm(A(k:k+nb,k:k+nb) $\rightarrow$ A(k:k+nb,j:j+nb));
        if $panel\_m > panel\_n$ then
            TASK: gemm( A(j:n,k:k+nb) $\times$ A(k:k+nb,j,j+nb) $\rightarrow$ A(j:n,j:j+nb) ) ;
```

Algorithm 2: LU implementation for multiple devices.

any extra programming effort. The first task of the trailing matrix update phase (trsm and gemm) consists of the update of the next panel. Since it is a priority task, the scheduling engine ensures that the scheduler places it at the top of the queue as a priority task (since it is on the critical path), tracks its dependencies, and once finished, sends it to the Master in order to perform the panel factorization of the next step, while the accelerator Device continues the update of the trailing matrix of step k. This technique is called lookahead, and is hidden here by the scheduler without any extra lines of code. Figure 3 shows the execution trace of the LU factorization on the KNL. We see that the panel factorization task of step $i+1$ runs in parallel with the update of the rest of the trailing matrix of step i, allowing lookahead.

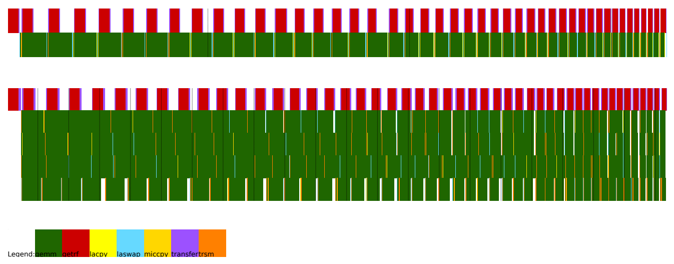
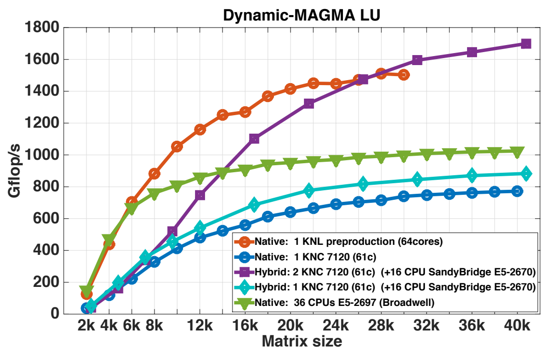


Figure 3: Execution traces for the LU factorization on KNL (64 cores) viewed as a Master using 4 cores and either one Device of 60 cores (top) or four Devices of 15 cores each (bottom) [11]. The panel factorization is represented in red (Master tasks) and the update by the green color (Devices tasks).



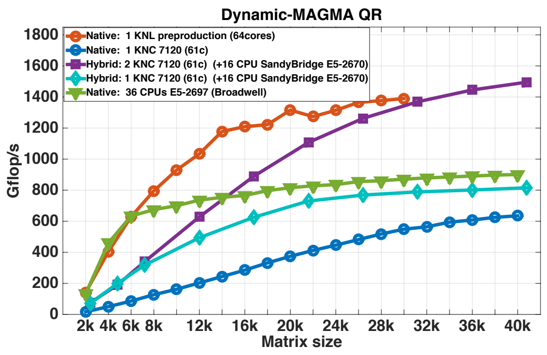
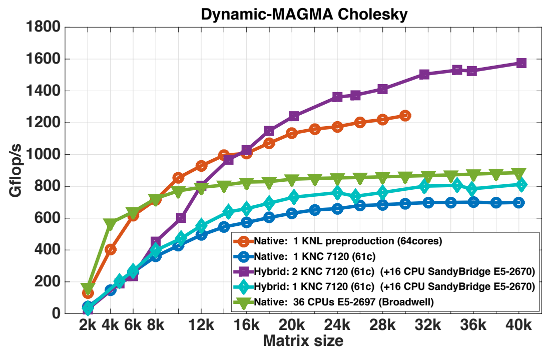


Figure 4: Performance of LU (left) and QR (right) across five hardware configurations.



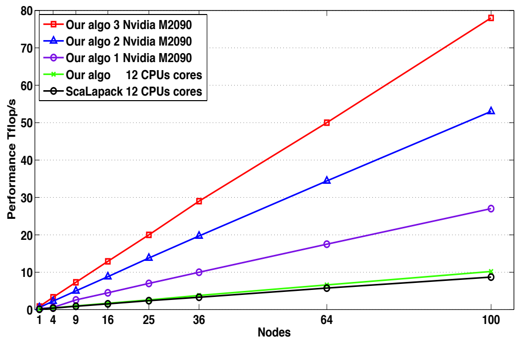


Figure 5: **Left:** Performance of Cholesky across five hardware configurations. **Right:** Weak scalability of the distributed multi-device Cholesky factorization on system with NVIDIA GPUs.

3.3 Framework abstractions

Fig. 6 shows the performance obtained when we used as the target hardware four generations of Intel Xeon multi-socket multicore servers with NUMA shared memory without any accelerators. Each processor generation represents a different combination of floating-point performance and memory bandwidth/latency, which influences the overhead from task switching during the scheduling process. The results presented in the figure show that we obtain good scalability across the processor generations even for relatively small matrix sizes and we are able to extract a sizable portion of peak performance. The presented runs were obtained with each hardware core was used as a single virtual compute device and the Intel HyperThreading was switched off.

The results show that we have the technology needed to target various hardware configurations from a single code that is performance portable across architectures. This opens further opportunities to extend our framework to be data-structure centric, where the codes are expressed with familiar data-structure abstractions that give unified view of data. Thus, data movements for example between CPUs/GPUs will be build in and therefore performed seamlessly to the developer. This can be accomplished through data abstraction APIs that can be integrated in POMPEI and be target of our future plans.

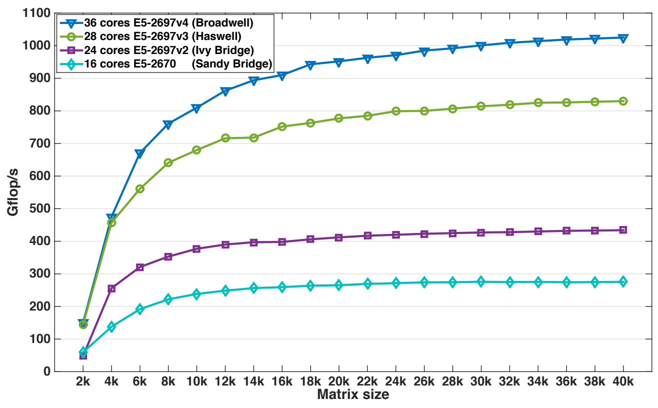


Figure 6: Performance of our approach on native mode for three different type of CPU hardware.

4 Cholesky factorization prototype

The framework presented in Section 3 uses the QUARK run-time. The framework can rely on other run-times and current work is on replacing QUARK by the OpenMP 4 tasking mechanism. We developed prototypes for experimentation and future work for the Cholesky factorization. Our deliverable for this period includes the three versions presented in Section 4. The tasks can use optimized implementations, e.g., BLAS from MKL, or our own implementations based on OpenMP. We developed representative Level-1, Level-2, and Level-3 BLAS kernels as a proof of concept and to demonstrate the portability of single BLAS implementation across architectures. Some performance results are given next, in Section 5.

5 Performance Results

5.1 Cholesky factorization across architectures

Figure 5, already mentioned in the previous section, illustrates the performance results in double precision (DP) arithmetic for the Cholesky factorization for three types of hardware and in both hybrid and native configurations. We use the same code to show its portability, sustainability, and ability to provide close to peak performance when used in native mode, on a single KNC (the blue curve), self-hosted pre-production single KNL (the red curve), 36 Broadwell CPUs only (the green curve), as well as in hybrid mode on 16 Sandy Bridge E5-2670 CPU cores with either one KNC (cyan curve) or two KNCs (purple curve). In addition to the portability, note that the results confirm the following observations. Our heterogeneous multi-device implementation achieves perfect scalability for large matrix sizes. In order to evaluate the performance of an algorithm we rate its performance compared to what we refer to practical peak which is the peak of the most compute-intensive and the most optimized Level 3 BLAS routine, the dgemm routine. The peak performance of the Intel MKL square dgemm on 36 cores Broadwell E5-2697 is about $1,140$ Gflop/s, on one KNC is 940 Gflop/s, and around $2,000$ Gflop/s for the KNL. The operands of the update phases have rectangular shapes reducing the update's performance to about 90% of the square gemm peak mentioned above. This also indicates that the panel factorization phase running on the CPUs is fully overlapped with the trailing matrix update running on the KNCs, and for that, the overall factorization performance reaches the Level 3 BLAS gemm performance. More attractive are the native performance results. We obtained about 772 Gflop/s on the KNC and about $1,500$ Gflop/s on the KNL, which is considered efficient and high performance. Note that when running in native mode for any hardware (CPUs, KNC, or KNL), the hardware is split over Master and Devices. The master is assigned to a small number of the hardware cores (in our experiments, about 10%), and the remaining cores are the ones that contribute to the trailing matrix update. As a consequence, in order to evaluate our algorithm, the peak now is the performance of the gemm on the remaining number of cores. The native codes are within 90+% of the hybrid ones, i.e., within 90+% of running just the Level 3 BLAS flops of the factorizations. A similar trend was observed for the QR and Cholesky factorizations.

Figure 7 shows a performance comparison of our results *vs.* Intel MKL for the Cholesky factorization on the KNL processor. MKL is optimized for all Intel Xeon and Xeon Phi architectures. The Intel MKL team puts additional effort into optimizing the Level 3 BLAS routines and the LAPACK LU, Cholesky, and QR factorizations, as these are some of the most commonly used routines in the library. The Cholesky factorizations are about the same performance with Intel MKL, slightly outperforming the POMPEI version.

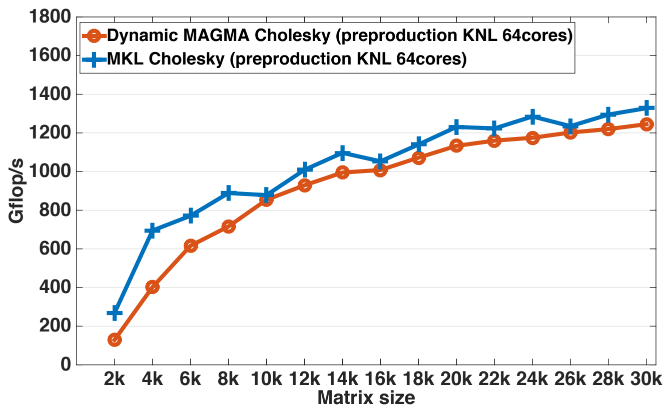


Figure 7: POMPEI *vs.* MKL Cholesky on the preproduction KNL in DP

5.2 Portability of Level-1 and Level-2 BLAS using OpenMP

Figure 8 shows the performance measurements of the daxpy kernel on GPU K20X (Left) and Xeon Phi KNC 7210 (Right) using three different methods of offloading and comparisons to the vendor optimized cuBLAS and KNL, respectively.

Figure 9 shows the performance measurements of the dgemv "T" kernel on a K20X GPU (Left) and Xeon Phi KNC 7210 (Right) using three different methods of offloading and comparisons to the vendor optimized cuBLAS and KNL, respectively.

The results illustrate that the offload model does not affect the performance of the kernels and even more impressive, that this behavior has been demonstrated across multiple platforms. Current work is on the dgemm kernel. The dgemm is compute bound to get top performance vendor optimized implementations are hardware-specific and use assembly language. Therefore, we expect that we can not reach the performance of vendor-optimized implemntations. Note that our model allows as to use vendor-optimized versions, when available, but for independence, when vendor libraries are not available or intentionally avoided, we are developing our own dgemm kernel.

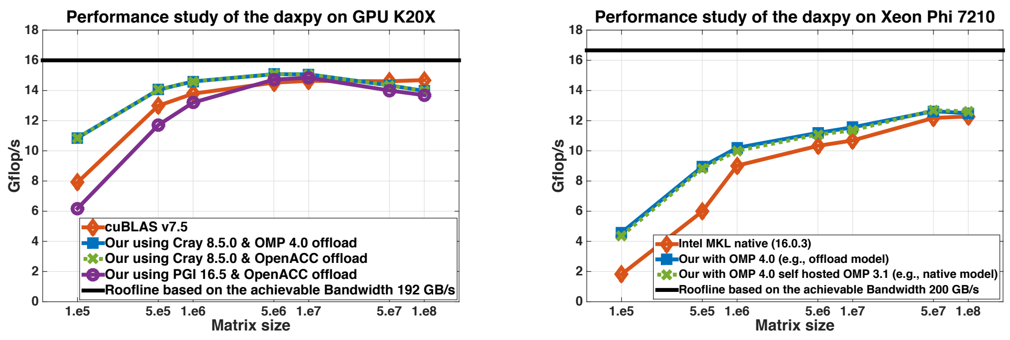


Figure 8: **Left**: Performance measurement of the daxpy kernel on GPU K20X using three different methods of offloading and comparison to the vendor optimized cuBLAS Library. **Right**: daxpy kernel on Xeon Phi KNC 7210 using the offload model and comparison to itself in native model and to MKL.

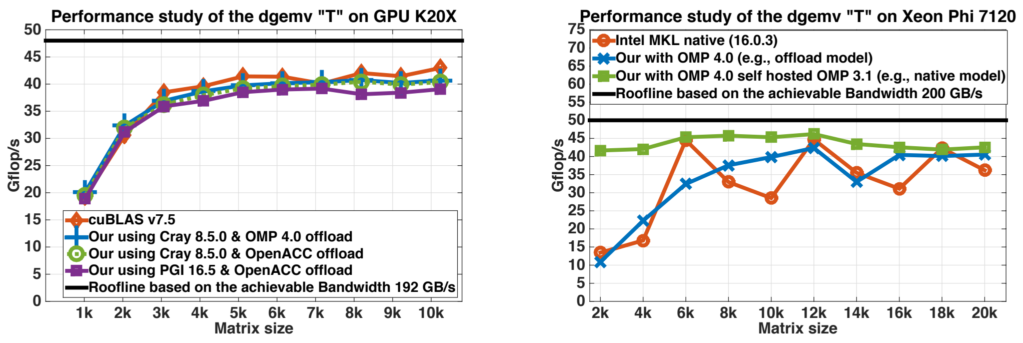


Figure 9: **Left**: Performance measurement of the dgemv “T” kernel on GPU K20X using three different methods of offloading and comparison to cuBLAS. **Right**: dgemv “T” kernel on Xeon Phi KNC 7210 using the offload model and comparison to itself in selfhosted model and to MKL.

5.3 Cholesky factorization using OpenMP tasking

Figure 10 shows a performance comparison of the Cholesky factorizations from Section 4 against the Apple optimized implementation in vecLib available in Accelerate Framework. Note that the reference implementation is very slow as it is based on Level-1 and Level-2 BLAS only. The blocked implementation is comparable to vecLib in Accelerate, and the tasking implementations in POMPEI outperforms both.

6 Conclusions and future directions

The work for this time period made significant contributions to the overall thrusts of the project. The use of OpenMP was evaluated for developing kernels as well as higher level algorithms, like the Cholesky factorization. A framework was designed and a prototype Cholesky was implemented using OpenMP tasking. The Level-1 and Level-2 BLAS results show that the offload model does not affect the performance of the kernels and this behavior has been demonstrated across multiple platforms. These BLAS implementations can be used in task-based approaches with OpenMP, where we demonstrated very good performance for the Cholesky factorization. Further, OpenMP tasking will allow the use of vendor opti-

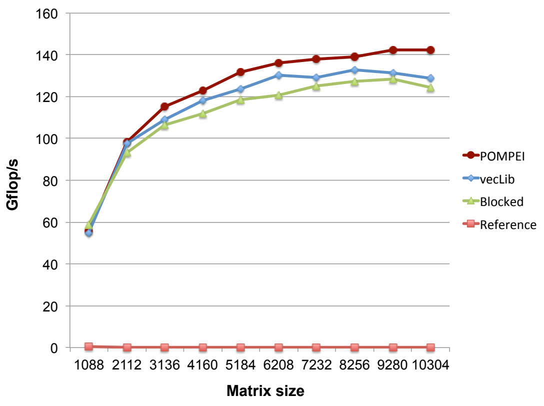


Figure 10: Performance comparisons of the Cholesky factorizations from Section 4 in double precision on 4 core Intel Core i7 @2.3GHz against the Apple optimized implementation in vecLib from the Accelerate Framework.

mized BLAS, if available, which use is the fundamental approach for obtaining performance portability in DLA libraries.

The work during this period opened a number of possibilities for extension and future work. First, it is of interest to port the framework presented to OpenMP. Second, the current abstractions for portability must be extended/replaced by data-structure abstractions that give unified view of data. In particular, this can be accomplished through the SharP library [12, 13]. The use of SharP and the integration of SharP data abstractions in POMPEI is the main target of our future plans. Finally, the development of performance portable Level-3 BLAS using OpenMP and SharP is a very high-value proposition for many applications.

Acknowledgments

This research was supported by the U.S. Department of Energy under subcontract No. 4000148194 issued by UT-Battelle, c/o ORNL. The work was also partially supported by Nvidia and Intel.

References

[1] Edward Anderson, Zhaojun Bai, Christian Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven J. Hammarling, Alan McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

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

[3] Reference BLAS implementation. Website: http://www.netlib.org/blas/. Accessed on Feb 15, 2016.

[4] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[5] M Lopez, V Larrea, W Joubert, O Hernandez, A Haidar, S Tomov, and J. Dongarra. Towards achieving performance portability using directives for accelerators. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16), Third Workshop on Accelerator Programming Using Directives (WACCPD)*, Salt Lake City, Utah, 11-2016 2016. Innovative Computing Laboratory, University of Tennessee, Innovative Computing Laboratory, University of Tennessee.

[6] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.

[7] M. Graham Lopez, Verónica Larrea, Wayne Joubert, Oscar Hernandez, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Evaluation of directive-based performance portable programming models. *International Journal of High Performance Computing and Networking (IJHPCN)*, (In Press), 2017 2017.

[8] Maksims Abalenkovs, Ahmad Abdelfattah, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki, and Asim YarKhan. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomputing Frontiers and Innovations*, 2(4), 10-2015 2015.

[9] Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.

[10] Azzam Haidar, Asim YarKhan, Chongxiao Cao, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Flexible linear algebra development and scheduling with cholesky factorization. In *17th IEEE International Conference on High Performance Computing and Communications*, Newark, NJ, 08-2015 2015.

[11] Azzam Haidar, Stanimire Tomov, Konstantin Arturov, Murat Guney, Shane Story, and Jack Dongarra. LU, QR, and Cholesky Factorizations: Programming Model, Performance Analysis and Optimization Techniques for the Intel Knights Landing Xeon Phi. In *IEEE High Performance Extreme Computing Conference (HPEC'16)*, Waltham, MA, 09-2016 2016. IEEE, IEEE.

[12] M. G. Venkata, F. Aderholdt, and Z. Parchman. Sharp: Towards programming extreme-scale systems with hierarchical heterogeneous memory. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 145–154, Aug 2017.

[13] Z. W. Parchman, F. Aderholdt, and M. G. Venkata. Sharp hash: A high-performing distributed hash for extreme-scale systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 647–648, Sept 2017.