Hierarchical DAG Scheduling for Hybrid Distributed Systems

Wei Wu*, Aurelien Bouteiller*, George Bosilca*§, Mathieu Faverge§, Jack Dongarra*†‡
*The University of Tennessee, Knoxville, USA
†Oak Ridge National Laboratory, Oak Ridge, USA
‡University of Manchester, Manchester, UK
§Bordeaux INP and Inria Bordeaux - Sud-Ouest, Talence, France

***Abstract*—Accelerator-enhanced computing platforms have drawn a lot of attention due to their massive peak computational capacity. Despite significant advances in the programming interfaces to such hybrid architectures, traditional programming paradigms struggle mapping the resulting multi-dimensional heterogeneity and the expression of algorithm parallelism, resulting in sub-optimal effective performance. Task-based programming paradigms have the capability to alleviate some of the programming challenges on distributed hybrid many-core architectures. In this paper we take this concept a step further by showing that the potential of task-based programming paradigms can be greatly increased with minimal modification of the underlying runtime combined with the right algorithmic changes. We propose two novel recursive algorithmic variants for one-sided factorizations and describe the changes to the PaRSEC task-scheduling runtime to build a framework where the task granularity is dynamically adjusted to adapt the degree of available parallelism and kernel efficiency according to runtime conditions. Based on an extensive set of results we show that, with one-sided factorizations, i.e. Cholesky and QR, a carefully written algorithm, supported by an adaptive tasks-based runtime, is capable of reaching a degree of performance and scalability never achieved before in distributed hybrid environments.**

***Keywords*-PaRSEC runtime; GPU; dense linear algebra; heterogeneous architecture**

I. Introduction

Throughput-oriented architectures, such as GPUs, are becoming ubiquitous assistants for computationally intensive tasks in scientific applications. Compared with traditional CPU, GPU has much higher peak performance and memory bandwidth. For example, the peak double precision floating point performance of the Nvidia Kepler K40 is approximately 1.43 Tflop/s, dwarfing the performance of any existing CPU family. As a consequence, an increasing number of production systems feature accelerators[1], a trend that is expected to persist in the future.

Open CL, Nvidia CUDA, and compiler extensions like OpenACC provide a friendly programming environment to support general computing on GPUs. Based on these concepts and APIs, numerous computational linear algebra routines optimized for GPU offload execution are provided by companion libraries, such as cuBLAS. However, the non-trivial integration of these routines into the already complex ecosystem used to program large scale distributed systems raises complex composition issues. While an analysis of the different programming models is outside the scope of this paper, one can note that the current de-facto programming paradigm is based on message passing, which encourages a static, hard-coded execution flow with explicit synchronizations between compute phases. Such a rigid execution model limits the capability of applications to adapt to the dynamic execution conditions faced on hybrid environments.

As a reaction to the multiplication of hardware trends challenging the status-quo, the dataflow programming paradigm has seen a revival, with the emergence of numerous task-based programming frameworks, where an algorithm is divided into computations entities (tasks) connected by data dependencies. This programming paradigm has been successfully used in different projects to depart from tightly coupled or fork-join programming paradigms, and express the parallelism in a form that allows for more execution flexibility and portability across many types of hardware resources. One of the early adopters of this programming paradigm is the PaRSEC [1] framework; which encompasses a toolbox to help express algorithms in the dataflow programming paradigm, and a runtime component whose role is to efficiently schedule the resultant Direct Acyclic Graph (DAG), on large scale distributed hybrid systems.

In the context of linear algebra, DAGs have been demonstrated to be an extremely effective way to describe tiled linear algebra algorithms [2]. In DAG based tile algorithms, each node of a DAG represents a compute task, called a kernel, and edges represent the dependencies resulting from the dataflow between these tasks. The matrix is divided into square tiles and each kernel operates on tiles. The size of tiles is one of the critical tuning parameters that impacts the efficiency of kernels, the degree of parallelism and the communication volume. Due to their different architectures, CPU and GPU require different tile sizes to achieve peak performance: usually GPUs require large tasks while CPUs benefit from smaller ones. Traditional tiled algorithms [3], which require all kernels to have a unique tile size, reach reasonable performance, but fail to provide the runtime with the means to achieve an adapted load-distribution on heterogeneous systems. In this paper, we propose a method to adapt the granularity of tasks with a multi-level approach,

[1] http://top500.org/

where tiles of different sizes coexist in the runtime. This method is called "hierarchical DAG" and is more suitable in hybrid execution environments. In the hierarchical DAG approach, large granularity tasks are organized in an outer DAG level, and GPU kernels operate directly on these large granularity tiles; Each large granularity task can be dynamically subdivided into a finer granularity inner DAG, operating on smaller tiles, so that the larger number of finer granularity tasks increases the available parallelism to levels adequate for multi-core processors. While this method is general, we will focus the rest of the discussion on the motivating case of linear algebra.

The contributions of this paper are: *a*) the hierarchical DAG approach to maximize the occupancy of computational kernels on heterogeneous computing resources; *b*) the hierarchical DAG implementation within PaRSEC– a high performance, production quality distributed dataflow framework; and *c*) the application of the principle to the Cholesky and QR factorizations and their evaluation on accelerated GPU clusters, notably, the tiled QR algorithm is a modified new version to enable hierarchical parallelism.

The rest of this paper is organized as follows. Section II introduces the related work. Section III outlines the general principle of hierarchical DAG for linear algebra algorithms. Section IV, presents how the PaRSEC runtime is modified to handle variable task granularity for GPUs, followed in Section V by a detailed description of the Cholesky and QR factorization examples. Section VI is dedicated to evaluating the performance in a variety of shared and distributed memory heterogeneous systems. Section VII concludes the paper and depicts potential future work.

II. Related Works

In order to motivate and illustrate our discussion, we will discuss the case of dense linear algebra, but the proposed approach is general and the runtime adaptations to support tiled linear algebra also apply to other types of algorithms that can benefit from varying task granularity. Dense linear algebra is one of the computing fields most likely to benefit early from any increase in the computational power of the hardware. Thus, it is not unexpected that every evolution at the hardware level is rapidly reflected in dense linear algebra libraries. MAGMA [2] is a linear algebra library designed for GPUs, using CUDA and OpenCL. MAGMA, in addition to the GPU, can take advantage of the CPU through multi-threaded BLAS libraries, but its static scheduler distributes tasks equally over GPUs, limiting the overall performance on complex heterogeneous architectures. The current version of the MAGMA library is solely usable on shared memory, without a version spanning over distributed resources. Fogue et al. ported the existing PLAPACK library to GPU-accelerated clusters [4]. Similarly, Quintana-Orti et al. extended the SuperMatrix runtime to shared-memory machines with GPUs [5]. However, these solutions require that GPUs take most of the computation (only the computationally less-intensive diagonal blocks are factorized on CPU), which produces load imbalance between CPU and GPUs. Additionally, all these prior works require an identical tile size, consequently, all tasks have the same granularity, and the efficiency of CPU and GPU is decreased.

There are a few prior works trying to resolve the tile size mismatch between CPUs and GPUs. Song et al. presented a heterogeneous tile algorithm [6] which divides square tiles into a skinny tall rectangle tile for CPU and places the remainder on GPU. It uses a non-uniform 1D partitioning and data is statically distributed between GPUs, hence, it is likely to cause imbalance in the Cholesky factorization. Kim et al. adapted the libFLAME library to support different block sizes on different devices in a shared memory environment [7]. However, its write-through GPU data caching policy may incur too many unnecessary data movements between the host and the GPU. Lima et al. reported a similar work for Intel Xeon Phi [8]. However, the decision to recursively split a task is made statically at submission time without runtime insight. Furthermore, in their Cholesky factorization, only the POTRF kernel is recursively split. Our approach uses a 2D block cyclic data distribution for each host, and data is dynamically assigned to GPUs to maintain good load balance. We maximize the throughput, by allowing all operations with the GPU to be asynchronous, overlapping data movements and task submission to the GPU, and allowing threads to migrate between GPU management and CPU execution. Thanks to the parameterized DAG of our solution, the decision is taken dynamically at runtime and is not limited to a single kernel, an important distinction as several kernels can compose the critical path of an application. Moreover, our approach is adapted to both shared and distributed systems.

III. Hierarchical DAG

A. Problem Statement

Tiled linear algebra is a representative class of algorithms that can be expressed efficiently with a dataflow: the parallelism between operations is represented with a DAG that symbolizes the flow of data between several tasks called kernels, which are described as nodes in the DAG. A "tile" can be considered as a sub-matrix of the original matrix. In tiled linear algebra algorithms, an $N \times N$ matrix is split into $NT \times NT$ tiles, each of size B ($\lceil N/B \rceil = NT$). Therefore, instead of computing element by element, each kernel executes on tiles. The tile size is a key tuning parameter that affects the efficiency of kernels tremendously. In most linear algebra algorithms, the tile size has been assumed to be constant for all kernels.

In most heterogeneous systems, a computing node features several CPU cores and one or more GPUs (note that in this paper we use the term GPU broadly to describe any similar type of accelerator, including Intel Xeon Phi.).

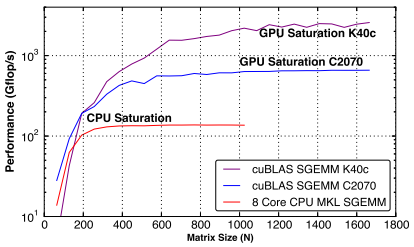

Figure 1. Performance of compute kernels on CPU and GPU depending on problem granularity

Kernels are executed on CPU cores or GPUs depending on their performance profile and the occupancy on the target execution unit. Some kernels experience a huge performance boost when executed on an accelerator, while some experience a mild speedup and are better kept executing on CPU cores. Some can be scheduled efficiently on either the CPU or GPU, and ideally the decision is then taken based on online load balance between the CPU and GPU units.

Compared with CPU cores, a GPU has many more lightweight computing units; GPU kernels reach their optimal efficiency for larger tile sizes, as they need to dispatch computation on many individual units to keep the occupancy high. On the other hand, CPU cores often reach good efficiency for moderate or small tile sizes. Figure 1 shows the different optimal tile sizes for the SGEMM (real single precision general matrix-matrix multiplication) kernel, on different environments. Intel MKL SGEMM, running on 8 cores of an Intel Nehalem Xeon E5520 CPU, reaches its peak performance starting from tile sizes larger than 200; while in cuBLAS SGEMM, the optimal tile size is larger than 1000 on a Fermi C2070, and larger than 1500 on a Kepler K40c. As a consequence, in a heterogeneous environment, selecting the proper tile size becomes a dilemma:

- If using the CPU optimal tile size, GPU kernels are not able to fully utilize the computing resources of the GPU since the small problem size cannot efficiently span over all GPU execution units.
- If using the GPU optimal tile size, given a certain matrix size N, the amount of exploitable parallelism is limited by the number of tiles, directly depending on the tile size (N/B). Thereby, for a fixed size problem, increasing the tile size proportionally decreases the parallelism. Furthermore, certain kernels (especially memory bound kernels) are less efficient than their functionally equivalent decomposition into smaller but more compute bound kernels. Executing these large kernels is thereby adding synchronous choke points that delay the execution of other dependent kernels, further decreasing the occupancy of all compute resources.

Our previous work employed a middle ground solution [9] by selecting an intermediate tile size, larger than the CPU optimal, but smaller than the GPU optimal. Clearly, this tradeoff solution fails to maximize the computing resource usage for both the CPU and GPU. To address this issue, we propose here a new solution called "hierarchical DAG", in which the tile size decomposition varies depending on the target unit executing the kernel, a decision taken dynamically based on the available parallelism.

B. The Hierarchical DAG Approach

The hierarchical method described below can be generalized to any number of hierarchies, but for the sake of the explanation we will consider a two levels hierarchy, GPU and CPU. Let the optimal tile size for a GPU be B, and the one for a CPU be a smaller tile size b. First, the input matrix is divided into $NT \times NT$ tiles of size $B \times B$, and the linear algebra algorithm is represented by a DAG whose task granularity is B. At the top level, all kernels in the DAG operate on large tiles, and the corresponding tasks are pushed into scheduling queues. When retrieving these tasks from the scheduling queues, a decision algorithm (described in Algorithm 1) is executed. If it is a GPU kernel, then it can be executed directly by calling the GPU kernel functions (as a cuBLAS function). If the kernel needs to be scheduled on a CPU core (because the kernel does not map well on GPU, or because GPUs are overloaded with pending work), then, the CPU kernel is called only if the granularity is bellow b. Otherwise instead of calling the CPU kernel functions directly on the large tile, the task is decomposed into a finer granularity DAG operating on the smaller tiles of size b.

Algorithm 1 Generic TASK_X(A) code in the "hierarchical DAG" approach. (b:small tile size)

```
if OnGPU ||((nbrows(A) < b)||(nbcols(A) < b)) then
    ComputeTaskX( A ) // by calling kernel function
    ReleaseDeps( Task_X, A )
else
    o = CreateDAG( Task_X, A,
                   ReleaseDeps( Task_X, A ) )
    Submit(o)
end if
```

When a large grain task is scheduled onto a CPU, the "hierarchical DAG" capable runtime decomposes the CPU workload into a finer grain parallelism that is more adequate for this type of execution units. The creation of the metadata representing the fine grain DAG happens online; no preprocessing or static decomposition is required. The runtime engine creates a local data descriptor, a different view of the input submatrix representing the large tile divided into smaller tiles. A new DAG is created to represent the fine grain decomposition of the task's algorithm applied on these smaller tiles. Tasks operating on large tiles that are scheduled for execution on the CPU are divided into finer grain tasks operating on $nt \times nt$ tiles of size b ($B = nt \times b$); the shape of the resulting multi-level graph for the Cholesky factorization is presented in Figure 2. These fine grain tasks are pushed

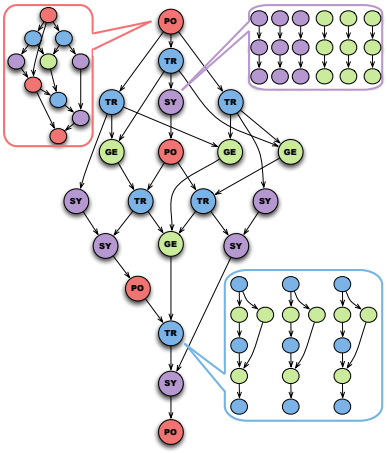

Figure 2. DAG of "hierarchical DAG" Cholesky factorization, whose size is 4×4 large tiles and then each CPU task is split into 3×3 small tiles.

into the scheduling queues and can execute on any available CPU core. Upon the completion of the final task in the finer grain DAG, the parent coarse grain task is completed through a callback system added as extra-information to the fine grain DAG: the metadata representing the fine grain DAG is released and the dependent coarse grain tasks are pushed into the scheduling queues. Multiple coarse grain tasks can be decomposed simultaneously and the resultant fine grain tasks scheduled concurrently on the available CPU cores.

IV. Hierarchical DAG Support in PaRSEC

PaRSEC focuses on task based applications, expressed as a DAG of tasks with edges designating data dependencies. PaRSEC takes this graph-based representation and assigns work to the computing resources, overlaps communications and computations and uses a dynamic, fully-distributed scheduler based on cache awareness, data-locality and task priority. In this section, we describe the PaRSEC support for GPU, and the modifications to enable the decomposition and the scheduling of hierarchical DAG programs.

A. Hierarchical DAG Task Scheduler

In a classical PaRSEC program, creating an instance of a DAG object, which represents the dataflow dependencies of an algorithm, is a collective operation across the entire distributed memory domain. The creation operation generates the local handle that contains the metadata used to track the state of the progress in the dataflow algorithm, but also allocates a unique identifier used to tag the internal messages exchanged between nodes to perform the distributed scheduling and data transfer. However, the DAG object instances spawned from within the coarse grain tasks to create the fine grain DAG cannot, and do not need to be created collectively. The scheduling between the distributed domains operates on the coarse grain DAG, without even the knowledge of these sub-graphs; thus it can remain unchanged. A new, thread-safe and non-collective DAG object creation operation has been added. It allocates the instance identifier in a local range that never collides with the global identifiers used for collectively allocated DAG objects. Beside this initial difference, the local DAG object instances are similar and can be managed concurrently by the same scheduler (with the exception that these tasks must all be scheduled on a shared memory local domain).

B. Employing Multiple GPU Streams

The execution of each GPU task is decomposed in three operations: moving data from the main memory into the GPU, kernel execution, and moving data back to the main memory. In order to overlap data movement and kernel execution, each operation type runs in a separate CUDA stream. Since PCI-E is bidirectional, we reserve one CUDA stream to handle data movement from CPU to GPU and another CUDA stream to handle the opposite direction. A single stream per direction is sufficient to saturate the PCI-E bandwidth and adding supplementary streams does not improve data movement speed. The GPU scheduler aggressively prefetches data to the GPU memory. As soon as a data is ready, and GPU memory is not full, input data is immediately pushed, even before the associated tasks become first in line of the scheduling queues. This ensures that tasks running on the GPU can start without delay and overlaps the host-device transfers with kernels computation.

GPU streams are also employed to partially circumvent the issues stemming from the conflicting goals of preserving parallelism with smaller tasks and improving per-task GPU efficiency with larger tasks that can employ all execution units of a GPU. By scheduling multiple GPU kernels simultaneously on multiple CUDA streams, the PaRSEC runtime improves the occupancy of the GPU units when moderately sized tasks are submitted: each task employs only a subset of the GPU processing units, but concurrently submitted tasks can employ the unused units. Performance results will demonstrate that even this optimization is insufficient to achieve maximum compute throughput without employing hierarchical DAG.

C. Data Coherency between Host and GPUs

PaRSEC minimizes data movement with a careful selection of the computational unit where a task is to be executed, based on the current load of the unit but also on the cost of moving the data needed for the task execution into the unit memory. Multiple copies of data are supported, coexisting in different memory layers. They are tracked using a data coherence protocol, a simplified version of the MOESI [10] cache coherency protocol. Tasks and data transfer may be reordered to promote the execution of tasks for which a copy of the data is already available in a target GPU memory, with the effect of reducing the amount of data transiting between the host and the GPUs. Regarding the qualitative aspect of the transfers, PaRSEC prioritizes the transfer for tasks closer

to the critical path of the algorithm. This guarantees that when the main PaRSEC scheduler follows the critical path of the algorithm as closely as possible, the tasks offloaded to an accelerator adhere to the same imperatives. The data coherency protocol operates at the coarse grain level and is oblivious of the existence of fine grain data copies.

D. CPU/GPU Load Balance

In a complex heterogeneous system, composed by CPUs and GPUs, one additional constraint is to be taken into account. The tasks generated by the algorithm that are distributed on the different computing resources should maintain a balance between the load of the different computing units. Without a load-balance mechanism, the overall computational throughput will decrease as some of the resources will become overloaded while other will starve. Many ways to avoid such situations are available in the literature. In general, these solutions require knowledge about the duration of each task execution for each computational resource, information we decided to ignore. Instead, our mechanisms are simpler, close to a greedy approach in which we strive to maintain all resources occupied simply based on the current workload of all devices (CPUs and GPUs).

When scheduling a hierarchical DAG program, the runtime load balancing mechanism has two separate levels. The first level separates the workload between CPUs and GPUs at the coarse grain level. Based on the assumption that each task type have similar durations, and that the driving difference between them is the cost of moving the required data to and from a device, the runtime computes the inverse of the theoretical peak performance of a specified device, and uses it as the weight of a task on this device; a device with a higher computing capacity will have a smaller cost per task. When a new task is considered by the scheduler, its cost is computed for each device, and the task is then assigned to the device which has the lowest current workload. However, to minimize data movement, the selection of the GPU execution device is also determined according to the current data locality: we prioritize the placement of the computation on a GPU that already owns most of the data that will be accessed by the task. The second level of load balancing is realized between fine grain tasks executed on CPU cores, where job stealing according to locality proximity is employed to equilibrate the fine grain tasks workload. Using this simple, yet efficient, workload management, PaRSEC can distribute tasks on different heterogeneous devices and maintain good load balance, as it can be seen in the Section VI.

V. Case Study: Hierarchical DAG Algorithms for Dense Linear Algebra

Cholesky and QR factorizations are linear algebra applications that are widely used for solving linear systems $Ax = b$, and for computing eigenvalues and singular values. Those factorizations are classic algorithms introduced in many recent libraries [3], [11] and exploited to illustrate tile algorithms and their adaptation to DAG representations. They are both composed of four kernels that are successively applied on the trailing sub-matrix at each step, as illustrated in Figure 3 for matrices of 6×6 tiles at iteration $k = 2$.

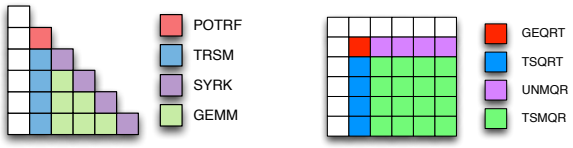

Figure 3. Step k=2 of tiled Cholesky (left) and QR (right) factorizations.

In heterogeneous systems, the developer determines which kernels are offloaded to the GPUs. In practice, the implementation of these kernels rely on the BLAS library (MKL on Intel CPUs, cuBLAS for Nvidia GPUs). We made the choice of offloading to GPU only the most computationally intensive kernels, respectively GEMM for the Cholesky and TSMQR for the QR factorization. These kernels represents the bulk of the computation time and experience a great speedup when executed on GPU, while the outlook for other kernels is not as favorable. All kernels (including GPU enabled ones) can also be scheduled on CPUs (when GPUs are overloaded, referring the load balance strategy discussed in section IV-D), in which case the hierarchical DAG strategy decompose them into a fine grain DAG. We will now discuss how the factorization algorithms can be adapted to take advantage of the adaptive task granularity, and the associated implementation of the TSMQR kernel for GPUs that is required to support sub-decomposition of tiles in QR.

A. GPU TSMQR kernel for the QR factorization

Algorithm 2 GPU TSMQR kernel.
(B:large tile size, $W1$, $W2$: workspaces)

1: **function** PARFB(A_1, A_2, V, T, W_1, W_2)
2: DtoDMemCpy(W_1, A_1) // $W_1 = A_1$
3: cublasGEMM(V, A_2, W_1) // $W_1 = W_1 + V^t A_2$
4: cublasGEMM(V, T, W_2) // $W_2 = VT^t$
5: cublasGEMM(T, W_1, A_1) // $A_1 = A_1 - T^t W_1$
6: cublasGEMM(W_2, W_1, A_2) // $A_2 = A_2 - W_2 W_1$
7: **end function**
8: **function** TSMQR(A_1, A_2, V, T, W_1, W_2)
9: **for** i **do** = 0 to B, step ib
10: PARFB($A_{1(i:B,1:B)}$,A_2, $V_{(1:B,i:i+ib)}$,
11: $T_{(1:ib,i:i+ib)}$,W_1, W_2)
12: **end for**
13: **end function**

An efficient implementation of the QR update kernel, so-called TSMQR, is essential to an efficient QR factorization, but also difficult to obtain because of the intrinsic "fork-join" nature of this kernel. Translating the CPU TSMQR

kernel directly from the PLASMA library leads to sub-optimal performance mainly due to it's usage of vector-matrix multiplication. We partially redesigned the CPU implementation of TSMQR kernel for the GPU to exploit the strengths of this architecture; the resulting function is presented in Algorithm 2. Triangular multiplications (TRMM) are replaced by general matrix-matrix multiplications due to the small granularity of the triangle (T is $ib \times ib$) at line 4. And, with an extra workspace, we experimentally verified that it is more interesting to compute twice $T^t W_1$ (a first time explicitly at line 5, and a second time hidden at line 6 in the $W_2 W_1$) than storing the result of this computation to replace line 5 by a sum of two matrices.

B. Multi-level Decomposition: Cholesky and QR

As discussed in Section III, tile size is a very important factor to achieve the best performance. Figure 2 shows that the number of independent kernels grows gradually with the number of tiles. Using a large tile size decreases the number of kernels, increase the execution time of each CPU kernel and thereby delays the release of dependent kernels. The two effects combine to reduce the efficiency of the CPU and generate idle time due to task starvation. When the hierarchical DAG approach is applied, the original DAG is dynamically transformed into a new DAG (Figure 2) featuring an adapted granularity for both CPU and GPU units. Ideally, all types of task should have a fine grain decomposition. In the Cholesky factorization, all four kernels: POTRF, TRSM, SYRK, and GEMM are available in the DPLASMA library as tiled algorithms. When a large tile is decomposed into an $nt \times nt$ tiled matrix with tiles of size $b \times b$, as presented in section III, the normal tiled Cholesky factorization kernel can be directly replaced by a tiled algorithms to create the fine grain DAGs, as shown on Figure 2 where tiles in the coarse DAG are split into matrices of 3×3 small tiles.

Such implementations of the tiled QR factorization kernels are not available in dense linear algebra libraries as of today. To apply the hierarchical DAG approach on QR factorization, tasks-based implementations of the four underlying kernels: GEQRT, UNMQR, TSQRT, and TSMQR have been added to the DPLASMA library. One can refer to [3], [12] for details on the sequential implementation of the kernels. Two implementations of those kernels could be made. The first one, similar to the Cholesky factorization, splits the kernels as small square tiles of size $b \times b$. The GEQRT kernel of the coarse-level DAG can then call recursively the tiled QR algorithm on the diagonal tiles. However, the Householder reflectors generated with the tiled QR algorithm on the small tiles are different from the ones normally generated at the large tile size B. This solution would imply a modification of all kernels at the outer level to adapt them to this new Householder reflectors representation (in particular the backward substitution to compute the solution of a linear system needs to be altered). Furthermore, those kernels would suffer from smaller granularity within the matrix-matrix operations, which should be avoided for GPU efficiency. Ballard et al. [13] recently proposed an algorithm to reconstruct the full Householder reflectors, avoiding this drawbacks at an additional increase in the flops counts.

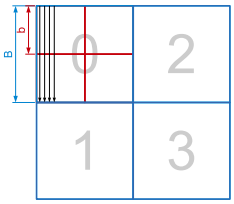

Figure 4. Different data layout: tile and LAPACK. For sub-tiles in the fine grain DAG (red), the data layout is the same as the LAPACK layout with interleaved data, while tile layout (blue) is used for large tiles and permits a much more efficient data transfer to/from the accelerators.

The alternate approach we have followed is to express the task-based algorithm of the four kernels with tasks working on rectangular tiles of size $B \times b$. This provides the smaller granularity researched for the CPU implementation that induces parallelism, and does not affect the kernels at the large level, as the computed Householder reflectors are the same. The four kernels have been implemented following this idea and integrated to the DPLASMA library to be used in our hierarchical DAG approach on QR factorization. Although, it is possible to use any multiple of ib (the inner block size specific to QR) for the b parameter, experiments have shown than using $b = ib$ gives the best performance, so all experiments have been performed with this setting.

C. Hybrid Data Layout

The hierarchical DAG algorithm divides a matrix into a set of small and large tiles. In a regular tiled algorithm, data of each tile is stored in contiguous memory (the so called tile layout). When the hierarchical DAG approach is applied, tiles used by CPU kernels are treated as a full matrix and a finer grain algorithm is applied on smaller sub-tiles. However, in these sub-tiles, the data layout is not contiguous anymore. Instead, sub-tiles are in the LAPACK data layout, where iterating from one column to the next jumps over a stride. Figure 4 shows the resultant hybrid data layout in the hierarchical DAG algorithm. We have adapted our tile algorithms to work indifferently on either tile of LAPACK layout. As a consequence, our algorithms can be applied directly onto fine or coarse grain tasks. It should be noted that this versatility may come at a performance price since employing LAPACK layout on small tiles may decrease data locality, but we expect (and demonstrate in the performance section) a profitable tradeoff. Another approach would be to perform in-place translation, but this carries a cost of its own, and in the light of the satisfying performance results obtained when operating on LAPACK format directly, we did not pursue such a speculative gain.

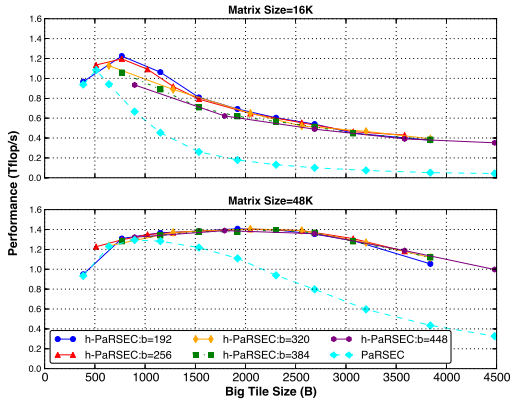

Figure 5. Performance for different tile size parameters (DPOTRF, using 1 GPU on Bunsen).

VI. Performance Evaluation

Experimental Setup: In this section, we investigate the impact of the hierarchical DAG approach. For a fair comparison, both the regular tiled and hierarchical DAG factorizations are implemented using the PaRSEC system (called h-PaRSEC when it executes hierarchical DAG algorithms). We also compare our implementation with the state-of-the-art implementation from MAGMA [14] (version 1.4). All the results presented in this paper use the real double precision variants DPOTRF and DGEQRF for Cholesky and QR, respectively. Experiments are carried out on two systems:

- Bunsen is a machine with 3 Nvidia Kepler K40c GPUs (12GB of memory per GPU) and 2 Intel Xeon E5-2650v2 (16 cores total). We use CUDA 5.0.35 and the Intel compiler 2013.4.183 (includes MKL BLAS).
- Keeneland Full Scale (KFS) is an FDR Infiniband cluster. Each node is equipped with 3 Nvidia Fermi M2090 GPUs (6GB of memory per GPU) and 2 Intel Xeon E5-2670 (16 cores total). We use CUDA 5.5 and the Intel compiler 14.0.1 (includes MKL BLAS).

A. Tile Size Tuning for Hierarchical DAG

Tuning the tile size has traditionally been a difficult issue for linear algebra software [15]. In the hierarchical DAG approach, two tile sizes need to be tuned in unison. Figure 5 presents the performance of DPOTRF on the Bunsen machine, when both the inner (b, executed on CPU) and outer (B, executed on GPU) tile sizes vary. The experiment is repeated for different matrix sizes (N=16K, 48K) to emphasize the impact of the tile size on the amount of parallelism available. Each curve represents a different value for b, for which B varies (on the x-axis). In addition, the performance of standard PaRSEC is also presented (then, the x-axis represents the single tile size used on both GPUs and CPUs). B is set as a multiple of both b and 64 (due to the physical organization of the CUDA warps on most Nvidia cards).

On Bunsen, sequential BLAS kernels executed on the CPU usually obtain their peak performance for $b > 180$. However, and although GPU kernel performance remains sub-optimal for tile sizes smaller than 1K (see Figure 1), the overall performance of standard PaRSEC (in dashes on Figure 5) on a heterogeneous platform decreases when increasing the tile size. Two intermingled effects are explaining this phenomenon. First, by increasing the tile size, the number of GEMM operations in the update of the trailing matrix is reduced, leading to reduced parallelism. Second, the factorization of the panel itself becomes a bottleneck: the associated operations apply to a single column of tiles, yet further progression is conditioned on their completion. With large tiles, panel parallelism is drastically reduced and the more parallel trailing matrix update is delayed, leading to under-utilization of computing resources. As can be seen, this effect persists even for large matrix sizes.

On the contrary, thanks to hierarchical subdivision of tasks into sub-DAGs, h-PaRSEC is much less subject to starvation from lack of parallelism (the panel factorization is divided into many small tasks whose granularity is adapted to reach CPU peak performance). Obviously, if the GPU tile size B is set too small (less than 512), the overall performance suffers from poor compute kernel efficiency. Increasing the value of B delivers the expected performance boost from the compute kernels' efficiency improvement, without suffering as much from lack of parallelism and poor performance on the CPU-executed panel factorization. Another interesting note is, when using the hierarchical DAG approach, finding a value of B that delivers acceptable performance is easier than when tuning for a single tile size. Even for small matrices that are prone to exacerbate lack of parallelism, the amplitude of performance difference is reduced; while for larger matrices, a very wide band of values deliver more than 90% of the best performing tuning. Developers can select the smallest tile size that maximizes CPU performance as the value for b, and then pick any reasonable multiple (around 1K) to set B. In the remainder of the experiments of DPOTRF, we apply such a tuning, and b is set to 192, while B varies between 384 and 1152 depending on the matrix size. The same tuning method can be applied for DGEQRF, and B varies between 384 and 1536 based on our tuning results and performance of DTSMQR in Figure 7.

B. Performance on One Node with Multiple GPUs

Figure 6 presents the performance of the Cholesky and QR factorization in both h-PaRSEC and PaRSEC implementation on the Bunsen machine. In both implementations, the tile size is tuned to perform best for this particular matrix size (the sizes used by h-PaRSEC are illustrated with a background color in the figure, the sizes employed in regular PaRSEC are similarly tuned).

For all matrix sizes, h-PaRSEC always exhibits better performance than standard PaRSEC, even for small matrices,

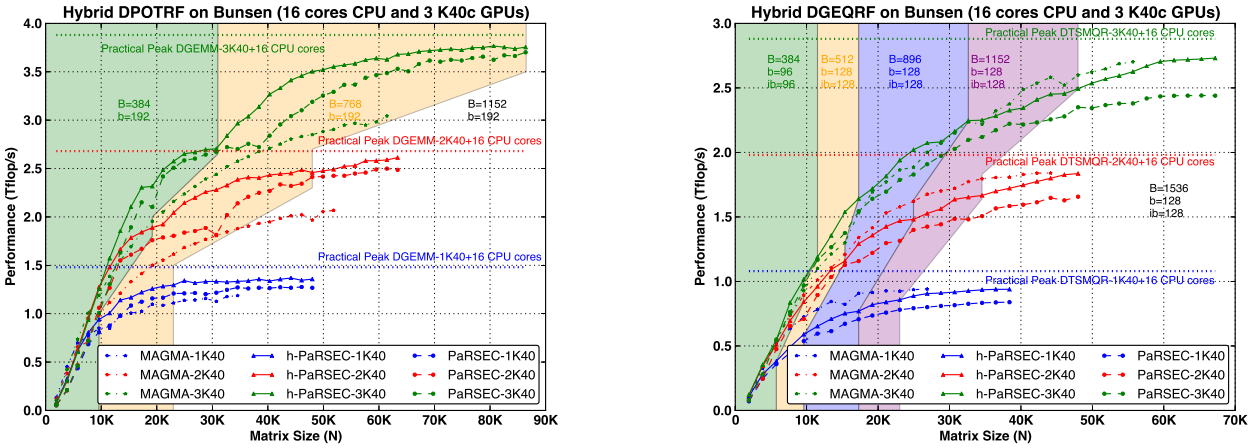

Figure 6. Performance of h-PaRSEC Cholesky and QR compared with regular PaRSEC and MAGMA.

when both employ the same tile size for kernels executed on the GPU. In this case, the advantage comes from employing a smaller tile size of 192 for computations executed on CPUs. For larger sizes, h-PaRSEC reaches 1.36Tflops/s for Cholesky using 1 GPU, which is 10% faster than standard PaRSEC, outlining that when more parallelism is available, higher kernel efficiency gives h-PaRSEC an extra boost.

Since the peak performance of cuBLAS DGEMM on 1 K40c is 1.2 Tflop/s, then based on the performance result from the 1 GPU experiment (1.36 Tflop/s), it can be inferred that CPUs contribute 160 Gflop/s on this platform. Based on these numbers, a perfectly scalable implementation of Cholesky would achieve approximately 2.56 Tflop/s using 2 GPUs and 3.76 Tflop/s using 3 GPUs (the contribution of the CPUs being accounted for only once). In practice, we obtain 2.5 Tflop/s with 2 GPUs and 3.7 Gflop/s with 3 GPUs, which demonstrates the scalability up to 3 GPUs is almost perfect. The same reasoning holds for QR.

Last, Figure 6 also presents the performance of the state-of-the-art MAGMA GPU linear algebra package for reference (please note that the MAGMA results do not include the cost of the initial transfer of the dataset to the GPU memory, whereas this cost is implicitly included for h-PaRSEC, when the relevant data are transferred in the background meanwhile computation is progressing). The comparison between MAGMA and h-PaRSEC demonstrates that by retaining a dynamic distribution of tasks, and dynamic load balancing between GPUs, while at the same time improving the efficiency of compute kernels by employing hierarchical DAG subdivision, h-PaRSEC can outperform (as seen for Cholesky) production quality software like MAGMA, whose data distribution and load balancing are static. Even though h-PaRSEC employs the tiled QR factorization (to improve parallelism in distributed memory deployments), which performs more floating point operations than the LAPACK layout QR algorithm employed by MAGMA, it can still compete closely on this single node experiment.

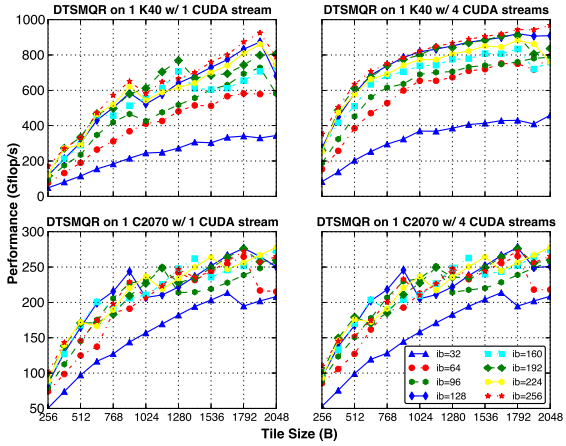

Figure 7. Performance of DTSMQR kernel on Fermi C2070 and Kepler K40 with 1 and 4 CUDA streams.

C. Multiple CUDA Streams

CUDA Streams, which represent multiple available execution contexts mapped onto the same physical GPU, can drastically improve the occupancy of GPU units by allowing the device to overlap executions from different streams on all available computational units. The potential for improvement is magnified when executing multiple small grain tasks, as is the case when employing an improperly tuned tile size.

Figure 7 shows the performance of DTSMQR runs on Kepler K40 and Fermi C2070 in different CUDA streams configurations. All cuBLAS calls comprising a GPU TSMQR kernel are submitted to a single CUDA stream, and based on the fact that B is usually about 10 times larger than ib, each TSMQR expands into more than 40 cuBLAS calls. The Fermi GPU architecture is unable to look ahead in the existing streams to execute tasks concurrently from multiple streams, hence, the independent TSMQR kernels cannot take advantage of multiple CUDA streams on the Fermi GPU. The only option to fill-up all computing on this GPU is to employ a larger B. On the other hand, the

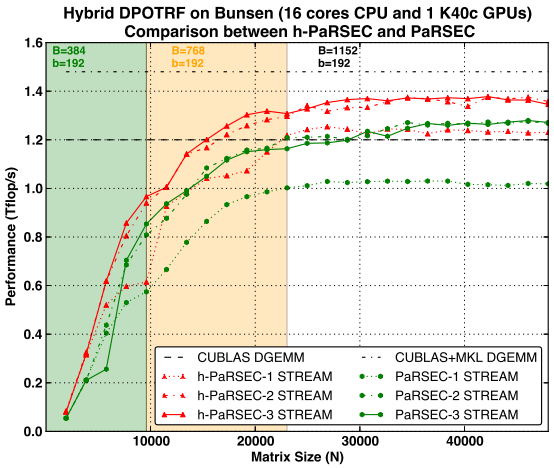

Figure 8. Performance difference between hierarchical DAG and the standard version on Cholesky with a varying number of CUDA streams (Bunsen using 1 K40c GPU).

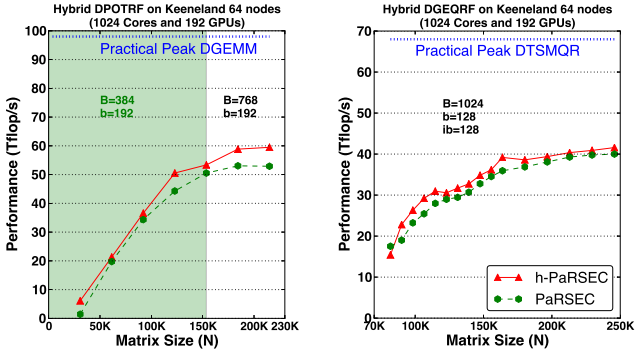

Figure 9. DPOTRF and DGEQRF performance with varying problem size; the dotted line is the DGEMM/DTSMQR peak performance (on 1 node, multiplied by 64)

Kepler architecture nicely handles multiple streams, leading to smoother and accrued performance.

Beside pure kernel performance, Figure 8 presents the performance with 1 Kepler K40c for the entire Cholesky factorization, when employing a variable number of streams to submit multiple GPU tasks. Employing several CUDA streams drastically improves the throughput of the GPU for both the standard PaRSEC and h-PaRSEC. However, even with this optimization, the performance of standard PaRSEC can only match that of h-PaRSEC restrained to 1 stream. When multiple streams are also employed in h-PaRSEC, it outperforms standard PaRSEC for all matrix sizes and stream configurations. Overall, these results outline that multiple streams are not a sufficient optimization to alleviate the need for employing the hierarchical DAG approach.

D. Distributed Memory

Last, we investigate the effect of hierarchical DAG on the performance of distributed memory machines. Figure 9 presents the performance of h-PaRSEC and standard PaRSEC Cholesky and QR factorizations, on 64 nodes (192 GPUs, 1024 cores) of the Keeneland KFS cluster, when the matrix size (N) varies. For intermediate matrix sizes, the advantage enjoyed by h-PaRSEC is small; but when the matrix is either small (thus with reduced available parallelism), or large (where kernel throughput dominates), h-PaRSEC outperforms standard PaRSEC by 10%, reaching 59 Tflop/s on Cholesky, which represents 60.5% of the practical GEMM peak (GEMM performance on 1 node, multiplied by 64), and reaching 42 Tflop/s on QR, which represents 62% of the practical TSMQR peak. Although the overall efficiency is not as high as in the shared memory machine, one has to consider that the execution platform is compute over provisioned: Even for compute intensive algorithms such as QR and Cholesky, the Infiniband 40G network is insufficient to feed 3 GPUs. This behavior is customary and can also be observed when comparing the efficiency per core of ScaLAPACK versus LAPACK.

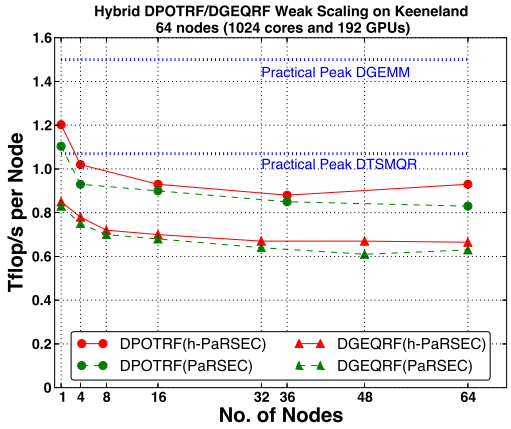

Figure 10. Weak Scalability: DPOTRF and DGEQRF performance as a function of the number of nodes, with a problem size scaled accordingly (Keeneland, 3 M2090 GPUs and 16 cores per node)

Figure 10 presents the weak scalability performance of h-PaRSEC and standard PaRSEC for the Cholesky and QR factorizations. In a weak scalability experiment, the problem size is set in accordance to the number of nodes, so that the workload per node remains constant when the number of nodes increases (N is 215K for Cholesky on 64 nodes, 245K for QR on 64 nodes.) The experiment demonstrates a good weak scalability for both standard PaRSEC and h-PaRSEC. However, as the node count gets higher, the hierarchical DAG approach exhibits a better scalability. At 64 nodes, h-PaRSEC obtains 78% of the ideal scalability on both Cholesky and QR factorization (performance at 1 node, multiplied by 64). When deploying data over PxP nodes based on 2D cyclic, for each task, the chance of a particular input data being local is $1/P^2$. When P is very small, many tasks can execute w/o communications. For any larger deployment, the communication/computation ratio is much lower, but the effect of varying P for large values of P is negligible. Therefore the scalability curve drops at

first, and then trends to be leveling. Overall the hierarchical DAG strategy better mitigates the heterogeneity within nodes which translates into a seizable gain on distributed systems.

VII. Conclusion

As heterogeneous compute nodes, featuring different types of processing units such as CPU cores and accelerators, become more pervasive, the need for a programming paradigm capable of providing portability and efficiency across a large range of hybrid environments becomes critical. The dataflow programming model, in which the inherent parallelism of the application is expressed as DAG, coupled with a runtime to manage tasks on any available computing resources according to dynamic conditions, has been proven to outperform legacy approaches while at the same time masking most of the complexity inherent in hybrid programming. In this paper, we have proposed a hierarchical DAG approach that further improves the applicability of the dataflow model to accelerated compute nodes. Task granularity becomes variable, and the runtime arbitrates depending on the type of the target computing unit.

The performance analysis demonstrates that such an approach improves the asymptotic performance for large matrices by employing the appropriate task grain on accelerators, while retaining a suitable amount of parallelism for CPU computations. It also enhances the scalability of the underlying algorithms, by providing a recursive approach capable of mapping the algorithm on all potential computing resources, with an immediate positive impact on the performance of small workloads, which is of critical importance for strong scalability. Beside a two level decomposition, the principle can be extended to a multi-level decomposition on architectures where this may be necessary, or to relieve the memory pressure from overly parallel DAGs whose diameter largely exceeds the number of computing units.

Acknowledgments

This material is based upon work supported by the Department of Energy under Award Number DE-SC0010682, by the National Science Foundation under Grant Number CCF-1244905 and by the Inria associated team MORSE.

References

[1] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parrallel Computing*, vol. 38, pp. 37–51, January 2012.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.

[4] M. Fogue, F. D. Igual, E. S. Quintana-Orti, and R. A. van de Geijn, "Retargeting PLAPACK to clusters with hardware accelerators," in *High Performance Computing and Simulation (HPCS)*. IEEE, 2010, pp. 444–451.

[5] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 121–130, 2009.

[6] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 365–376.

[7] K. Kim, V. Eijkhout, and R. A. van de Geijn, "Dense matrix computation on a heterogenous architecture: A block synchronous approach," FLAME Working Note, Tech. Rep. 63, August 2012.

[8] J. Lima, F. Broquedis, T. Gautier, and B. Raffin, "Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor," in *Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2013, pp. 105–112.

[9] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, N. O. Saengpatsa, S. Tomov, and J. J. Dongarra, "Performance portability of a GPU enabled factorization with the DAGuE framework," in *Cluster Computing*, vol. 12. IEEE, September 2011, pp. 395–402.

[10] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices, May 2013, no. 24593-rev.3.23, pp. 169–172.

[11] E. Chan, F. G. Van Zee, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van De Geijn, "Satisfying your dependencies with supermatrix," in *Cluster Computing, 2007 IEEE International Conference on*. IEEE, 2007, pp. 91–99.

[12] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators," in *International Parallel & Distributed Processing Symposium*, Anchorage, United States, May 2011.

[13] G. Ballard, J. Demmel, L. Grigori, E. Solomonik, M. Jacquelin, and H. D. Nguyen, "Reconstructing Householder Vectors from Tall-Skinny QR," in *International Parallel & Distributed Processing Symposium*, Phoenix, United States, May 2014.

[14] S. Tomov, R. Nath, H. Ltaief, and J. J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Parallel & Distributed Processing Workshops (IPDPSW)*. IEEE, April 2010, pp. 1–8.

[15] Y. Sawa and R. Suda, "Autotuning method for deciding block size parameters in dynamically load-balanced BLAS," in *Software Automatic Tuning*, K. Naono, K. Teranishi, J. Cavazos, and R. Suda, Eds. Springer, 2010, pp. 33–48.