Performance Portability of a GPU Enabled Factorization with the DAGuE Framework

George Bosilca*, Aurelien Bouteiller*, Thomas Herault*, Pierre Lemarinier‡,
Narapat Ohm Saengpatsa*, Stanimire Tomov*, Jack J. Dongarra*†
{bosilca, bouteill, herault, lemarini, saengpat, tomov, dongarra}@eecs.utk.edu
* *Innovative Computing Laboratory, the University of Tennessee*
† *Oak Ridge National Laboratory*
‡ *IRISA, Université de Rennes 1*

***Abstract*—Performance portability is a major challenge faced today by developers on heterogeneous high performance computers, consisting of an interconnect, memory with non-uniform access, many-cores and accelerators like GPUs. Recent studies have successfully demonstrated that dense linear algebra operations can be efficiently handled by runtime systems using a DAG representation. In this work, we present the GPU subsystem of the DAGuE runtime, and assess, on the Cholesky factorization test case, the minimal efforts required by a programmer to enable GPU acceleration in the DAGuE framework. The performance achieved by this unchanged code, on a variety of heterogeneous and distributed many cores and GPU resources, demonstrates the desired performance portability.**

***Keywords*-cluster, GPU, linear algebra, DAG scheduling**

I. Introduction and Motivation

The current trend of High Performance Computing requires the effective exploitation of network-interconnected computing nodes, whose computing power is provided by a large number of cores sharing the main memory in a non-uniform way assisted by Graphic Processing Units or other accelerators. As a consequence, the computing elements inside a node are becoming extremely heterogeneous. The HPC community has tried different approaches to relieve the difficulties of dealing with such heterogeneous environments. MPI has been the tool of choice to enable application scalability between numerous distributed memory nodes, but has shown shortcomings when considering multicore machines. Different projects demonstrated the benefits of using GPU accelerators for scientific computation. However, most of the existing applications that exploit the computing power of GPUs are tailored specifically for GPU computation only (or GPU-CPU pairs), excluding CPU cores from the computational resources, or lack the capability to run on distributed systems. The increase in the number and diversity of computing units within nodes introduces a challenge for library and application developers, who need to adapt their code to more diverse target systems. Overall, there is a need for these orthogonal efforts to be unified in order to deliver a coherent and complete programming framework that comprehends the difficulties of parallel systems with many heterogeneous computing units, and provides the key feature of performance portability.

To harness the computing power of such architectures, and address the inherent code porting challenge, recent works propose to use tasks scheduling of micro kernels, letting the middleware schedule those on the computing resources, thereby allowing the application or library developer to focus on the implementation of the algorithms. The code is transformed by a development framework into a direct acyclic graphs of tasks, with data flowing between tasks. This approach provides a simple way to express and exploit fine-grain parallelism automatically. DAGuE [1] is one of the few micro-kernel scheduling frameworks able to ensure weak scalability in a distributed machine, in addition to strong scalability in a single node, harnessing the peak performance of many-core architectures.

In this paper we present how the DAGuE framework was extended to make use of the GPU's computing capabilities. By reusing a single node kernel ported to GPU programming, and extending the middleware, we demonstrate how to harness the computing power of a highly heterogeneous distributed system. We then consider the efforts required by the application developers for porting their code to GPU accelerated machines; the Cholesky Factorization, a classical linear algebra algorithm is adapted to GPUs, without changing the main algorithm representation. We also use this algorithm as a test case, to underpin the performance obtained on a variety of GPU accelerated systems, demonstrating that this same unchanged code can be ported to widely differing types of systems (accelerated, with multiple accelerators, distributed and accelerated, non accelerated), while still accessing first class performance.

The rest of the paper is organized as follows. In Sec. II, we present the related works; then, in Sec. III we introduce the modifications to handle GPUs in the DAGuE framework and investigate in Sec. IV the amount of work required by the application programmer to port a Tile Cholesky factorization to GPU hardware with DAGuE. Sec. V evaluates the performances of the framework on a variety of distributed and heterogeneous hardware, before we conclude in Sec. VI.

II. Related Work

A. DAG based scheduling and GPU computing

DAGs have a long history [2] of expressing parallelism and task dependencies in distributed systems. Previously, they have often been used in grids and peer-to-peer systems to schedule large grain tasks, mostly from a central coordinator. [3], [4] present a taxonomy of DAGs that have been used in grid environments. More recently, in [5], [6] DAGs are used to describe tile linear algebra algorithms on multicore CPUs. [7] presents the MAGMA project that leverages a similar approach for linear algebra on GPUs. [8] defines *codelets*, a task description language, to enable the execution of the same operation on different hardware. [9], [10] then demonstrates how micro-tasks scheduling can mix shared-memory cores and accelerators. Compared to our proposition, these approaches do not tackle the issue of executing on distributed memory hardware.

B. GPU computing on distributed systems

GPUs have also been used in distributed systems, outside DAG based approaches. [11] accelerated the LINPACK benchmark on heterogeneous clusters by providing hybrid kernels (to be executed simultaneously on both GPU and CPU cores) for the DGEMM and DTRSM routines. In [12], the authors explain how some computations of a large scale bag-of-task peer-to-peer system have been ported to GPU. Closer to the linear algebra community, [13] uses GPUs to execute critical kernels in a parallel application, in a more classical SPMD fashion. These approaches can successfully extract performance, but require tailored engineering. One of the contributions of the present paper is to demonstrate the excellent performance level and lower engineering cost of DAG based approaches for harnessing the power of GPU accelerated nodes in distributed systems. Most of the complexity and tuning of the heterogeneous hardware, and moving data back and forth between nodes and accelerators, is hidden inside the DAGuE runtime, while the same GPU kernels developed for MAGMA (or simply CUBLAS) can be used directly.

III. Support for GPUs in the DAGuE Framework

A. The DAGuE Project and Goals

DAGuE stands for Directed Acyclic Graph unified Environment. The goal of this framework is to relieve the developer from the burden of fine tuning its application to the intricate system-centric issues of current heterogeneous HPC architectures, such as explicit communication, overlapping communications with computations, mutual exclusion and synchronizations, load balance, cache reuse and memory locality on NUMA hardware. One missing feature in DAGuE, toward this unification goal, was to automatically use accelerators, when available.

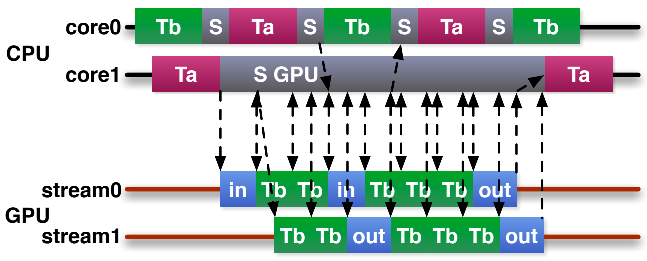

Figure 1. Schematic (not to scale) DAGuE execution, on a GPU enabled system; kernels Ta and Tb alternate with scheduling actions (S) and in/out GPU asynchronous memory accesses.

To achieve these ambitious goals, the DAGuE project includes a runtime library and several development tools to help with building and analyzing DAGs of micro-tasks. The DAGuE Runtime takes a symbolic and concise representation of a DAG of tasks, built by the DAGuE framework from different possible input languages (for more details, please refer to [1]). For regular algorithms, like the Cholesky Factorization, developers can use a SMPSS-like representation [14], very close to the algorithm description of Figure 2, automatically translated into the intermediate Job Data Flow (JDF) representation. In this format, the algorithm is divided into two parts, the depiction of the dependencies introduces by the flowing of data from task to task, and the sequential computing bodies that are to be applied to data. In this work, the first part is left unchanged, the programmer just needs to add GPU code to the second part. The proposed approach have to retain the core properties of DAGuE: a symbolic representation of the dependencies that can be evaluated in a problem-size independent and distributed way, implicit communications based on the static data distribution provided by the application, and dynamic scheduling and load balancing inside the computing nodes, with a scheduler that takes care of data consistency and cache reuse internally.

B. Scheduler Triggers for GPU management

In order to retain most of the automatic features of DAGuE regarding scheduling and data management, most of the GPU handling must be integrated deep inside the DAGuE scheduler. In the DAGuE runtime, each thread alternates between the execution of kernels and running the lightweight scheduler (see Figure 1). When some tasks needs to be executed on a GPU, the new Hybrid scheduler switches the thread into GPU support mode. From this point on, this thread orchestrates the submission of tasks for this GPU, and remains in this mode until this GPU has no more work to process. As a consequence, each GPU effectively subtracts a CPU core from the available computing power as soon as (and only if) it is processing. Because the typical compute time of a GPU kernel is tenfold smaller than a CPU one, should all CPU cores be processing, the GPU controls would be delayed to the point of, in average, make

```
FOR k = 0..TILES-1
    A[k][k] ← POTRF(A[k][k])
    FOR m = k+1..TILES-1
        A[m][k] ← TRSM(A[k][k], A[m][k])
    FOR n = k+1..TILES-1
        A[n][n] ← SYRK(A[n][k], A[n][n])
        FOR m = n+1..TILES-1
            A[m][n] ← GEMM(A[m][k], A[n][k], A[m][n])
```

Figure 2. Tile Cholesky factorization pseudocode

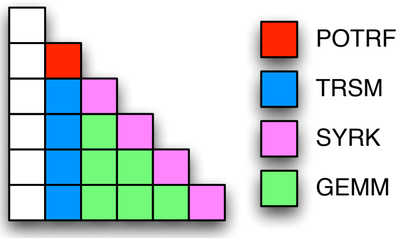

Figure 3. Step k of a Cholesky factorization.

the GPU run at the CPU speed. However, as GPU tasks are submitted asynchronously, a single CPU thread can fill all the streams of hardware supporting concurrent executions (such as NVIDIA Fermi); similarly, we investigated using a single CPU thread to manage all available accelerators, but that solution proved not scalable, as the CPU processing power is overwhelmed and cannot treat the requests reactively enough to maintain all the GPUs occupied.

C. GPU Data Consistency and Tracking

Another new issue introduced by GPU accelerators is data movement back and forth from the accelerator memory, which is not a shared-memory space. The DAGuE hybrid scheduler is extended to handle GPU-host memory movements. The hybrid scheduler, when in GPU mode, multiplexes the different operations asynchronously, using multiple streams to enable overlapping of I/O and GPU computation.

The regular scheduling strategy of DAGuE is to favor cache reuse, by selecting when possible a task that reuses most of the data touched by prior tasks. The same approach is extended in the hybrid scheduler, to prioritize on the GPU tasks whose data have already been uploaded. Similarly, the scheduler avoids running tasks on the CPU if they depend on data that have been modified by the GPU (to reduce CPU/GPU data movements). It moves data from the CPU to the GPU, places kernel execution orders, and moves data back from the GPU to the CPU, as well as detects tasks completions and triggers the corresponding actions in the task scheduling system. In addition, in order to maintain the data consistency between the main memory and the GPU memory, a *Modified Owned Exclusive Shared Invalid* (MOESI) [15] coherency protocol is implemented. The hybrid scheduler is also in charge of retrieving data that are needed to satisfy remote dependencies on distributed systems.

D. Codelets for Accelerator Support in the DAGuE Bodies

The last introduced feature in the GPU DAGuE runtime is the support for codelets. A codelet is a piece of code that encapsulates a variety of implementation of an operation for a variety of hardware. In DAGuE, codelets are sequential, in the sense that they target a single processing unit, either a core, a GPU stream, even though they can still contain some internal parallelism, such as vector SIMD instructions. Practically, that means that the application developer is in charge of providing multiple versions of the computing bodies. The relevant codelets, optimized for the current hardware, are loaded automatically during the algorithm initialization (one for the GPU hardware, one for the CPU cores, etc). Today, the DAGuE runtime supports only CUDA and CPU codelets, but the infrastructure can accommodate easily to other accelerator types (OpenCL, FPGAs, Cell, ...). If a task features multiple codelets, the runtime scheduler chooses dynamically, between all these versions, in order to execute the operation on the selected hardware. Because multiple versions of the same codelet kernel can be in use at the same time, the workload of this type of operations, on different input data, can be spread on both CPU cores and GPUs simultaneously.

IV. Accelerator Enabled Tile Cholesky Factorization

A. The Tile Cholesky Factorization Use Case

A typical DAGuE application is a regular MIMD application that calls numerical routines implemented with DAGuE. In this section we present the extension of the Cholesky factorization routine to use GPU accelerators. The Cholesky factorization (or Cholesky decomposition) $A = LL^T$ is mainly used for solving linear systems of the form $Ax = b$, where A is a symmetric and positive definite matrix, and L is a lower triangular matrix with positive diagonal elements. Such systems often arise when solving partial differential equations, or in physics applications where the modeled phenomenon is symmetrical. The tile Cholesky algorithm (introduced in [16]) is identical to the classical LAPACK block Cholesky algorithm, except for processing the matrix by tiles (see Figure 2). Its distributed implementation in DAGuE (fully described in [17]) is based on automatically transforming this sequential algorithm in the data flow dependencies between the different tasks (BLAS kernels). The sequentiality of the execution is relaxed, and the resulting algorithm explores the execution space in an opportunistic dependency-driven order, oblivious of the original loop nesting.

B. Selection of the GPU Enabled Kernels

As discussed in the previous section, to enable accelerators in a DAGuE implementation, the programmer is required to only provide sequential GPU kernels functionally equivalent to the CPU kernels. The Cholesky factorization is based on four kernels. Figure 3 depicts how, at each iteration k, these kernels are applied to some blocks of a $N \times N$ matrix. We describe below each of the kernels, and their contribution to the total factorization time.

POTRF: The kernel performs the Cholesky factorization of a diagonal (triangular) tile T and overrides it with the final elements of the output matrix. There is one POTRF on the diagonal tile at position (k, k). The completion of the POTRF releases the dependencies on all the TRSM described below. With only one appearance per iteration, this kernel is a minor contributor to the overall workload. Its GPU implementation is slower than its CPU equivalent, hence it does not need to be GPU enabled.

TRSM: The operation applies an update to a tile A below the diagonal tile T, and overrides the tile A with the final elements of the output matrix. The operation is a triangular solve. There are $N - k - 1$ TRSM operations during iteration k. Each of these TRSM, once completed, releases the dependencies of a set of update GEMMs and a SYRK on the corresponding row. Again, this operation appears only $O(N^2)$ time during the algorithm, and is slower on GPU than on CPU.

SYRK: The kernel applies an update to a diagonal (triangular) tile B, resulting from factorization of the tile A on its left. The operation is a symmetric rank-k update. There are $N - k - 1$ SYRK per iteration k, one per remaining diagonal tile. Although this operation can be efficient on GPU, we decided not to enable it: the CUBLAS kernel currently provided is slow, and the $O(N^2)$ overall contribution to the workload does not mandate considering a custom GPU implementation.

GEMM: The operation applies an update to an off-diagonal tile C, resulting from factorization of two tiles A on its left. The operation is a matrix multiplication. The GEMMs are totally independent of one another, and are therefore embarrassingly parallel. The $O(N^3)$ overall workload of GEMMs usually accounts for more than 90% of the computation time. Moreover, the GEMM kernel on GPU is significantly faster, and many GEMM kernels are applied on the same data, which reduces GPU to host traffic. Hence, we focused our efforts on enabling GPU computing for this particular kernel.

C. Optimized GPU GEMM Kernels

The only modification to the original DAGuE Cholesky factorization is the addition of a GPU SGEMM body codelet that embeds the call to the MAGMA, or CUBLAS kernels, depending on the runtime detected underlying accelerator. For the Tesla and Fermi GPUs, CUDA SGEMM kernels are reused from the MAGMA library. The Fermi kernel had to be further tuned for the DAGuE framework. The originally developed one [18] uses 64×4 thread blocks, each computing a 96×96 block of the resulting matrix. This kernel binds textures to global memory for direct access to the data that would be texture cached. Although this approach gives about a 5% performance improvement, reading through texture is not Error Correction Code (ECC) protected. We removed the texture reads and reduced the blocking size to 80×80 to increase inter-tile parallelism, while still retaining high efficiency. When the GPU accelerator ecosystem becomes more mature, one can expect that the BLAS library (CUBLAS for this hardware) will be adequately tuned by default, as is MKL for Intel CPUs, reducing the programmer's actions to only adding prefixes to function names in the DAGuE body codelets.

V. Performance Evaluation

A. Experimental Conditions

The purpose of this performance evaluation is to investigate how the generic Cholesky factorization performs on differing heterogeneous hardware. We use two very different platforms, one featuring many GPU accelerators on a single node, the second featuring a cluster of GPU accelerated nodes.

Mordor: It is a single node Tesla S1070 blade. It includes four Tesla C1060 boards, each with 240 1.44GHz CUDA FPUs and 4GB of memory. The theoretical peak of this accelerator is 4.14 Tflop/s in single precision, but for all practical purpose, it has to be reduced to 2.76 Tflop/s as it has extraneous ADD units that are not balanced by the number of MUL units. The host features 4 quad core AMD Opteron SE8358 at 2.2GHz (16 cores total). Each NUMA socket is served by 8GB of memory (32GB total). The accelerator S1070 blade is connected by two PCI-Express x16 lanes, each lane shared by two C1060 cards.

Dancer: It is an 8 nodes, 64 cores, Infiniband 20G cluster. The nodes feature two NUMA Nehalem Xeon E5520 at 2.53GHz (hyperthreading is disabled), with 2GB of memory (4GB total). The accelerator board and Infiniband NIC are connected on independent PCI-E lines. Half of the nodes are equipped with a Tesla C1060, while the other four feature the more recent Fermi C2050. The Fermi accelerator features 448 1.15GHz CUDA cores, which translates into a peak performance of 1.03 Tflop/s in single precision. The board also contains 3GB of memory. We refer to deployments using both types of nodes, in equal number, as the Hybrid cluster.

Software: Both systems are running the Linux 64bit operating system, version 2.6.31.2 on Dancer, and version 2.6.35 on Mordor. The software is compiled with the Intel compiler suite 11.1 (including MKL). The network backend

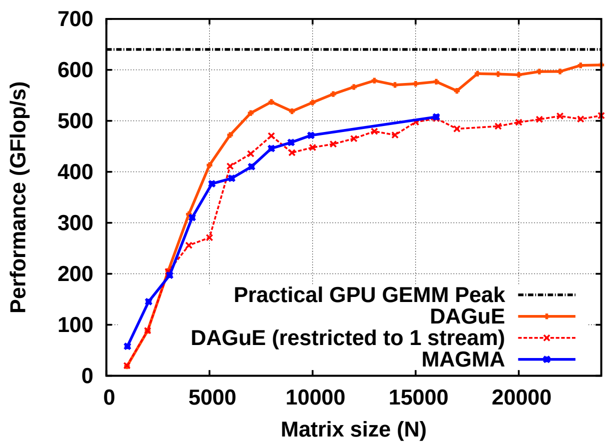

Figure 4. Fermi (C2050) Performance of Cholesky factorization according to problem size. Comparison between DAGuE and MAGMA (Dancer, single node).

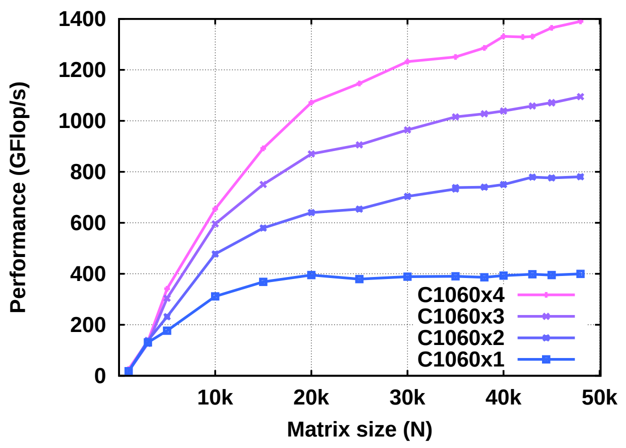

Figure 5. Multiple Teslas (C1060) Performance of Cholesky factorization according to problem size (Mordor, single node).

in DAGuE uses Open MPI 1.4.2. The version of CUDA is 3.1 on Mordor and 3.2 on Dancer. Experiments use single precision arithmetic (appropriate to Tesla hardware).

B. Single GPU Single Node

Figure 4 presents the performance of DAGuE and MAGMA for the Fermi nodes on the Dancer system. MAGMA is the state of the art implementation of a linear algebra library for GPU accelerated single node machines. On this experiment, two DAGuE setups are presented, one with streams and one without. In a CUDA kernel, the GEMM input matrix is divided into inner blocks, called *warps* which are then mapped to the grid of CUDA cores. To reach parallel efficiency on distributed machines, DAGuE favors a smaller tile size; as a consequence, there is no exact divisor between the DAGuE tiling and the CUDA cores grid, leading to some imbalanced warps at the end of every kernel. Without streams, the GPU cores (including idle ones) are all locked while those warps are executed. Because MAGMA does not have to accommodate for distributed resources, it can take as input the entire untiled matrix and then apply an inner tiling of its choice that maps to the CUDA grid. This explains why, although not benefiting from the computing power of the CPU cores, MAGMA can compete very closely with DAGuE, when restricted to one stream.

On the Fermi hardware, Nvidia introduced the capability of concurrently running several kernels. When this concurrent execution feature is enabled, the load imbalance in DAGuE from the outer tiling can be recovered by the extra parallelism expressed by the DAG representation between different GEMM kernels. As a consequence, as soon as enough GEMM kernels are ready simultaneously, for matrices larger than 2000, the tuning advantage of MAGMA can be negated by this extra inter-kernel parallelism. Moreover, as the DAGuE scheduler takes care of data movement between CPUs and GPUs, it can accommodate for larger matrices that do not fit entirely in the GPU memory. Overall, the DAGuE code competes favorably with MAGMA, thanks to the extra computing power provided by the CPU cores.

C. Multiple GPU Single Node

To further stress how the DAGuE Cholesky test case performs on a variety of hardware, Figure 5 depicts the performance on the Mordor multi-GPU shared memory machine. Overall, based on the performance of a single GPU in this configuration (around 400 GFlop/s), a perfectly scalable framework is expected to deliver around 1500 GFlop/s on all four GPUs (the contribution of the CPUs being accounted for only once). With up to 2 GPUs used at the same time, the scalability is almost perfect. However, the measured performance out of the 4 GPUs is slightly lower than expected. This is mostly a consequence of the 2 GPUs boards sharing a single PCI-E lane: a careful observation of the traffic on the PCI-E bus indicates that due to the increase in the number of requests to move data to and from the GPUs, the shared bus becomes a bottleneck (a similar effect is discussed in another context in Table I). Despite this intricate hardware aspect, the DAGuE runtime is able to harness a major speedup from this architecture as well, from the same unchanged code.

D. Accelerated Clusters

1) GPU/NIC PCI-E bandwidth contentions: The first question, when considering a distributed system encompassing at the same time GPU accelerators and high performance network interface cards, is to what extent the fact that both type of hardware feature DMA chipsets, that compete for the PCI-E and memory bandwidth, introduces perturbations on the achieved performance. Table I presents the results of performing concurrent accesses to the memory and the

Table I
IMPACT OF CONCURRENT ACCESSES BETWEEN GPU AND NIC ON THE BANDWIDTH (GB/S) WITH A TILE SIZE OF 384

Perturbation	none	remote die	same die	interleave
Network	-	11.533	11.363	11.001
GPU 0 push	29.250	26.497	12.897	25.919
GPU 1 push	21.509	21.580	11.457	21.553
GPU 0 pull	13.746	12.897	11.366	12.060
GPU 1 pull	13.089	11.457	9.636	10.767

PCI-Express bus with GPU accelerators and HPC network interfaces. In these experiments, two Dancer nodes were added an extra GPU (Nvidia 8600GT). The NetPIPE ping-pong benchmark has been modified to spawn two extra threads, in order to stress the memory and PCI-E subsystems with concurrent CUDA memory traffic to and from a GPU. When no Infiniband interference is taking place, the GPU memory copy aggregated bandwidth is over 50Gb/s pushing data to the GPUs and 26.7Gb/s retrieving data. When the Infiniband traffic is pinned to a different socket from the one hosting the GPU threads, the GPU aggregated bandwidth is slightly reduced to 48Gb/s and 24.3Gb/s. The worst case scenario is to pin both Infiniband operations and GPU traffic to the same socket, which reduces the performance to 33.1Gb/s and 20.9Gb/s. The Infiniband bandwidth is almost unaffected. When the accessed memory is spread on all NUMA banks (numactl interleave mode), the performance penalty is comparable to the remote die setup; on the Dancer system, all the PCI-Express buses are separated, therefore the perturbations are mostly the consequence of memory bank contentions. This setup mimics the floating network thread of the DAGuE environment, demonstrating that, on average, it avoids interference with the GPU operations.

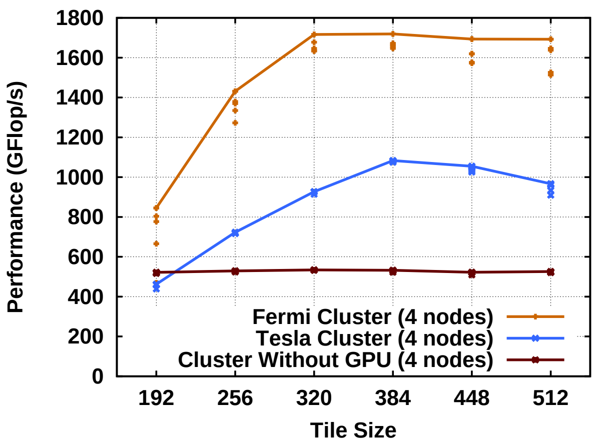

Figure 6. Performance of the Cholesky factorization as a function of the tile size, for a problem size of 34560 (Dancer cluster, 4 nodes).

2) Mixed Hardware Types: On heterogeneous system that includes many different components, like Dancer, tuning the size of the tiles on which the DAG will executed impacts several parameters, from the speed of the BLAS kernels on the differing computing units, to the efficiency of the network transfers. Figure 6 presents the performance of the Cholesky factorization on a 4 node cluster, when varying the tile size, for a fixed problem size of 34560. The CPU-only experiment illustrates that the DAGuE framework is flexible enough that the network and CPU efficiency are unaffected by the tile size, hence the tuning can focus on GPU efficiency only. The performance of the GPU accelerators are indeed strongly dependent on the tile size. On the Fermi cluster, the performance increases steeply when growing the tile size up to 320, but remains constant for larger tiles. On the Tesla cluster, the performance drops when using tile sizes larger than 384 (due to unavailability of multiple streams). Overall, DAGuE allows for a single set of tuning parameters that performs adequately on all the considered hardware of Dancer, even when the setup is mismatched.

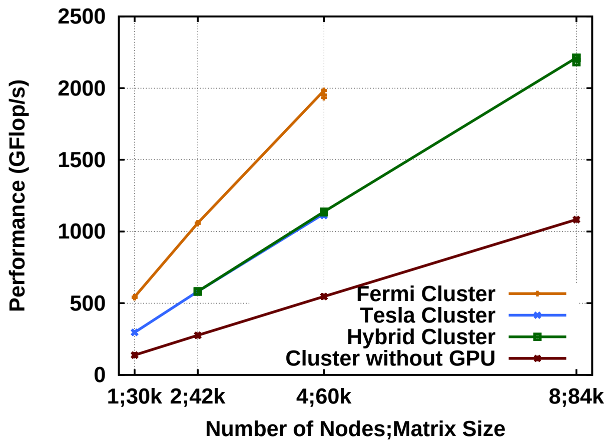

Figure 7. Weak Scalability: performance of the Cholesky factorization as a function of the number of nodes, with a problem size scaled accordingly (Dancer cluster).

3) Scalability: Figure 7 exhibits the weak scalability, i.e., the performance of the system when increasing both the number of computing resources and problem size in order to keep the workload per node constant. On platforms featuring similar nodes (either all Tesla, all Fermi or all CPU), the DAGuE runtime can harness the maximum speedup from the distributed architecture.

Distributed platforms can be heterogeneous in two different ways. First, by featuring heterogeneous computing units inside the nodes, a feature that is expected to become mainstream for HPC systems in a near future and is a main motivating factor for DAGuE existence. Second, by gathering nodes of differing computing capacity, as is often the case in desktop grids computing, but is not typical of HPC. Because of the hardware features of our test machine, to present 8 nodes scalability, we were forced to use the Hybrid platform nonetheless. The typical linear algebra

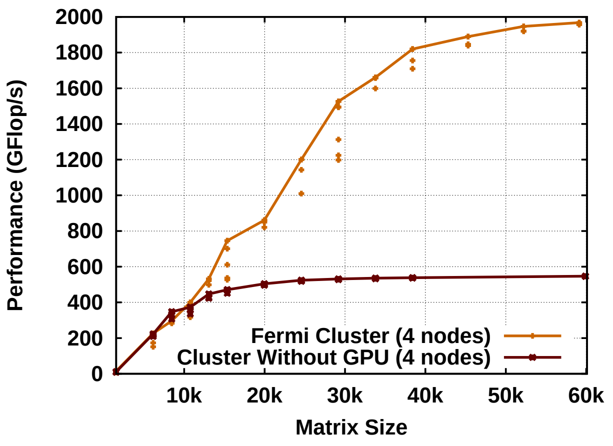

Figure 8. Problem Scaling: performance of the Cholesky factorization as a function of the problem size, (4-node Fermi Cluster).

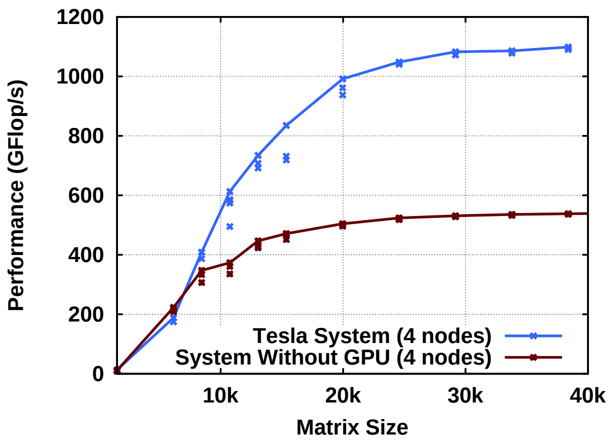

Figure 9. Problem Scaling: performance of the Cholesky factorization as a function of the problem size, (4-node Tesla Cluster).

distribution used by the Cholesky algorithm is regular 2D block cyclic, meaning that the workload is equally divided between all nodes, regardless of their computing capacity. As a direct consequence the maximum achievable performance of the distributed system is constrained by the performance of the slower Tesla nodes. A non regular distribution could have improved the performance, but tackling the issue of non-uniform capacity nodes is outside the goals of DAGuE.

The experiment demonstrates a perfect weak scalability, under the constraints imposed by the initial data distribution; the overall performance at 8 nodes for the hybrid system is double of that of the Tesla or Hybrid setups with 4 nodes. Therefore the heterogeneity within nodes and the communication to computation ratio imbalance, originating in accelerator capabilities, do not have a negative effect on the weak scalability property of DAGuE (demonstrated without GPUs up to thousands of cores in [1], [17]).

4) Problem Scaling: Figure 8 presents the performance of the Cholesky factorization as a function of the problem size, using all available nodes for the Fermi homogeneous distributed system. The x-axis represents the leading dimension of the square matrix. We compare the evolution of the performance with the performance of an accelerator-free system. The peak, with GEMM running on a single core, has been measured at 19.55 GFlop/s, leading the aggregated CPU peak of 625.6 GFlop/s. The DAGuE implementation without accelerators reaches 87.3% of the GEMM peak. Taking into account the accelerators, the DAGuE implementation using Fermi cards requires larger matrices to reach full efficiency, as the parallelism within the nodes is not sufficient to overlap the communication of tiles between GPUs and CPUs with computations inside the GPU. Nonetheless, for a larger matrix size, the Fermi system obtains 1.97TFlop/s, which represents 77% of the maximum GEMM performance aggregating all CPUs and GPUs (practical peak). Similarly, on the Tesla system (see Figure 9) the runtime is able to reach 76% of the practical peak. Again, the same GPU-enabled Cholesky code can achieve very good performance on a variety of distributed system, without modifications.

VI. Conclusion

With CPU frequencies reaching their pinnacle, high performance supercomputers maintain their growth in computing power by multiplying the number of cores and adding computing accelerators, such as GPUs or FPGAs. Different programming paradigms have been proposed to address this sharp increase in the number of heterogeneous computing elements. One of these paradigms is scheduling of DAGs of micro tasks. Using this approach, the DAGuE framework has successfully harnessed the performance of large scale clusters of many cores for various linear algebra operations.

In this paper, we presented how this framework has been adapted to heterogeneous hardware, enabling the use of GPUs. The fully distributed scheduler of DAGuE has been adapted to a hybrid scheduler, capable of selecting codelets and orchestrating GPU operations when beneficial for the performance of the application. The hybrid scheduler extends the cache-reuse heuristics of DAGuE to the GPU in order to minimize data movements. Porting an application using a regular DAGuE algorithm to use GPUs consists solely of the addition of the necessary GPU codelets (and kernels), as is illustrated by the presented Cholesky factorization algorithm. The performance evaluation, on a variety of heterogeneous hardware, including distributed clusters of GPUs and multi-GPU shared memory nodes, demonstrates not only that the DAG approach reaches excellent levels of performance, but it also does so while using the same code on all considered platforms, thanks for the DAGuE runtime taking care of handling the intricacies of heterogeneity. In the future, we plan to scale the evaluation on larger scale distributed systems, when they become available, and to port other numerical operations.

References

[1] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing." in *Proceedings of the 16th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-11)*. IEEE, May 2011.

[2] J. A. Sharp, Ed., *Data flow computing: theory and practice*. Ablex Publishing Corp, 1992.

[3] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," Journal of Grid Computing, Tech. Rep., 2005.

[4] O. Delannoy, N. Emad, and S. Petiton, "Workflow global computing with YML," in *7th IEEE/ACM Int. Conf. on Grid Computing*, september 2006.

[5] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," in *Applied Parallel Computing. State of the Art in Scientific Computing, 8th Int. Workshop, PARA*, ser. LNCS, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, Eds., vol. 4699. Springer, 2006, pp. 1–10.

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

[7] 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," *Journal of Physics*, vol. 180, 2009.

[8] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[9] H. Ltaief, S. Tomov, R. Nath, P. Du, , and J. Dongarra, "A scalable high performant cholesky factorization for multicore with GPU accelerators." LAPACK Working Note, Tech. Rep. 223, Nov. 2009.

[10] 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 *25th IEEE IPDPS*, Anchorage, USA, May 2011.

[11] M. Fatica, "Accelerating LINPACK with CUDA on heterogenous clusters," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 46–51.

[12] T. J. Desell, A. Waters, M. Magdon-Ismail, B. K. Szymanski, C. A. Varela, M. Newby, H. J. Newberg, A. Przystawik, and D. P. Anderson, "Accelerating the milkyway@home volunteer computing project with gpus," in *Parallel Processing and Applied Mathematics, 8th Int. Conference, PPAM 2009, Wroclaw, Poland, September 13-16*, ser. Lecture Notes in Computer Science, vol. 6067, 2009, pp. 276–288.

[13] M. Fogue, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Retargeting plapack to clusters with hardware accelerators," in *HPCS'10*, 2010, pp. 444–451.

[14] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE Int. Conf. on*, 29 2008-oct. 1 2008, pp. 142 –151.

[15] AMD, "Amd64 architecture programmers manual volume 2: System programming," AMD64 Technology, Tech. Rep., 2011.

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

[17] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma," in *12th IEEE Int. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11)*. IEEE, may 2011.

[18] R. Nath, S. Tomov, and J. Dongarra, "An Improved MAGMA GEMM for Fermi GPUs," LAPACK Working Notes, Tech. Rep. 227, 2010.