XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures

Thierry Gautier[†], João V. F. Lima[*‡], Nicolas Maillard[‡], Bruno Raffin[†]
[*] *Grenoble University, France*
[†] *INRIA, Grenoble, France*
[‡] *Federal University of Rio Grande do Sul (UFRGS), Brazil*
thierry.gautier@inrialpes.fr, {joao.lima, nicolas}@inf.ufrgs.br, Bruno.Raffin@inria.fr

***Abstract*—Most recent HPC platforms have heterogeneous nodes composed of multi-core CPUs and accelerators, like GPUs. Programming such nodes is typically based on a combination of OpenMP and CUDA/OpenCL codes; scheduling relies on a static partitioning and cost model.**

We present the XKaapi runtime system for data-flow task programming on multi-CPU and multi-GPU architectures, which supports a data-flow task model and a locality-aware work stealing scheduler. XKaapi enables task multi-implementation on CPU or GPU and multi-level parallelism with different grain sizes. We show performance results on two dense linear algebra kernels, matrix product (GEMM) and Cholesky factorization (POTRF), to evaluate XKaapi on a heterogeneous architecture composed of two hexa-core CPUs and eight NVIDIA Fermi GPUs.

Our conclusion is two-fold. First, fine grained parallelism and online scheduling achieve performance results as good as static strategies, and in most cases outperform them. This is due to an improved work stealing strategy that includes locality information; a very light implementation of the tasks in XKaapi; and an optimized search for ready tasks.

Next, the multi-level parallelism on multiple CPUs and GPUs enabled by XKaapi led to a highly efficient Cholesky factorization. Using eight NVIDIA Fermi GPUs and four CPUs, we measure up to 2.43 TFlop/s on double precision matrix product and 1.79 TFlop/s on Cholesky factorization; and respectively 5.09 TFlop/s and 3.92 TFlop/s in single precision.

***Keywords*-High Performance Computing; Data-Flow task model; Heterogeneous architectures; Locality Aware Work Stealing; Dense Linear Algebra;**

I. Introduction

With the recent evolution of processor design, future generations of processors will contain hundreds of cores. To increase the performance per watt ratio, the cores will certainly be non-symmetric with few highly powerful cores and numerous, but simpler, cores. The success of these machines will rely on the ability to schedule the workload at runtime, even for small problem instances.

Several libraries [1], [2] or languages such as Cilk [3], X10, Fortress, Chapel or OpenMP are designed to improve productivity. They encourage programmers to express all the potential parallelism in an application at fine grain, while delegating to the runtime, possibly with the help of a compiler, the extraction of parallelism for the target multiprocessor. A parallel construct, such as *for each* (Cilk `cilk_for`, X10 `for_each`, OpenMP `#pragma omp for`), enables to identify potential concurrent instruction sets, an approach commonly used in numerical applications. Still, these softwares provide instructions to express task parallelism, such as `cilk_spawn` (Cilk), `async` (X10) or `#pragma task` (OpenMP). All these software runtimes rely on schedulers using variations of the work stealing algorithm [4], [5]. The main drawback of a *for each* construct is the addition of strong synchronization points to enforce the completion of a set of independent tasks, and the associated memory update, before a new task set can be executed. This synchronization points restrict the overall parallelism. This happens for instance in classical block matrix factorizations (Cholesky, LU, QR) where parallelism between tasks across the outer iterations exists [6].

On the other hand, the data-flow model simplifies programming by unfolding parallelism based on data-flow dependencies between tasks. Runtime systems with support for data-flow programming are nowadays *de facto* standard for parallel linear algebra libraries on multi-cores [6], [7], [8]. Besides, they can automatically move data between address spaces, such as on multi-CPU or multi-GPU architectures [9], [10], [11], [12] or clusters [13], [14], [15].

In the context of multi-CPU and multi-GPU heterogeneous architectures, a runtime system needs to offer a programming model that considers heterogeneity in terms of computing power and disjoint address spaces. Multi-core processors are more suitable to memory-bound or comparison oriented computations, whereas GPUs sustain performance on highly parallel, compute-bound, problems. Moreover, data transfers over different address spaces must be avoided in order to guarantee scalability and efficient work load balancing. A data-flow model is suitable to heterogeneous architectures since it describes all necessary data transfers, but processing heterogeneity still depends on both programming and scheduling.

The data-flow softwares cited above attain significant performance results on heterogeneous architectures, but rely on restricted programming models and a static partitioning scheduling. Since their programming models only consider one level of parallelism, without the capacity to adapt the computation to the resource, their scheduling algorithms handle statically defined work partitions and cost models. The HEFT heuristic used in StarPU [11] relies on a cost model to predict performance based on the expected task execution time on each type of target hardware resource. The uncertainties related to the estimations, in particular when communication costs are high, may impact the overall performance [16].

In this paper, we introduce XKaapi, a runtime system for data-flow task programming on heterogeneous architectures. XKaapi combines a C++ interface for data-flow programming (Section II) and a work stealing based scheduler to support multi-CPU and multi-GPU architectures (Section III). The main contributions we propose are:

- A locality-aware work stealing algorithm based on heuristics to manage data locality and tackle the cache-unfriendly problem of classic work stealing [17], which is critical on multi-GPU systems;
- A fully asynchronous task execution strategy on GPUs to overlap data transfers with GPU kernel executions;
- Low overhead tasks that permit to efficiently handle a high degree of parallelism;
- Lazy computation of dependencies to throw the overhead on the critical path rather than on the work;

We evaluate the XKaapi runtime with two dense linear algebra algorithms in double precision: matrix product and Cholesky factorization (Section IV). First, our experiments evaluate our asynchronous task execution strategy on a single-GPU. Second, we measure the performance of our locality-aware work stealing based on two heuristics. Our experiments show that XKaapi achieves a speedup of about 7.7 for matrix product and 6 for Cholesky factorization when using eight GPUs and four CPUs over one GPU and one CPU. In terms of raw performance, with matrices of size 40960×40960, we attain about 2.43 TFlop/s on double precision parallel matrix product (5.09 TFlop/s on single precision), and 1.79 TFlop/s on double precision Cholesky factorization (3.92 TFlop/s on single precision).

XKaapi results on the Cholesky factorization can be compared to the best performance obtained on a similar platform, of roughly 760 GFlop/s obtained on a system made of twelve CPUs and three GPUs [15]. Our parallel Cholesky achieves more than 800 GFlop/s on nine CPUs and three GPUs.

II. Data-Flow Task Programming with XKaapi

The XKaapi[1] task model [18], as in Cilk [3], Intel TBB [1], OpenMP-3.0, StarSs [10] or OmpSs [14], enables non-blocking task creation: the caller creates the task and proceeds with the program execution. The semantic remains sequential such as XKaapi's predecessors Athapascan [13] and KAAPI [18] (which was further specialized for multi-CPU/multi-GPU iterative applications [9]). Still, in this paper, we introduce a general scheduling algorithm for multi-CPU/multi-GPU systems that enforces a locality-aware work stealing (Section III).

XKaapi has several APIs (C, Fortran, C++) to program heterogeneous parallel architectures. In this paper, code fragments rely on the C++ API.

A. Data-Flow Task Definition and Creation

A XKaapi program is a sequential code complemented with annotations or runtime calls to create tasks. Parallelism is explicit, while the detection of synchronizations is implicit [18]: the dependencies between tasks and the memory transfers are automatically managed by the runtime.

A task is a function call that returns no value except through its effective parameters. Tasks are created by calling the template function `ka::Spawn`.

The code fragment of Figure 1 illustrates how to program a Cholesky factorization using the C++ API. The `ka::Spawn<Task>` creates a task of type `Task`. Each parameter `rk,rm,rn` corresponds to a range of indexes, and a construction such as `A(rm,rk)` represents the submatrix of elements `A(i,j)` where `i,j` are in the range `rm,rk`. The data type `range_2d` is an abstraction to view a memory region as a 2D array.

The right side of Figure 1 illustrates the definition of a task signature (`TaskSYRK`) that includes the task parameters and their access modes (read `R` and/or write `W` and/or concurrent write `CW`). An implementation for CPU is given by the specialization of the `TaskBodyCPU` template class.

B. Execution by Work Stealing

The runtime creates a system thread for each computational resource to be used. On multi-CPU, a resource is a core. A thread creates tasks and pushes them on its own work queue, which is represented as a stack. The enqueue operation is very fast, typically about ten cycles on the last x86/64 processors [2]. Alike Cilk and OmpSs [14], a running XKaapi task can create children tasks. This is not the case for the other data-flow programming software previously mentioned [10], [11], [19]. Once a task ends, the thread executes its children following a First-in First-out (FIFO) order by popping tasks from its own work queue.

Thanks to Cilk [3], [4], the work stealing technique has become popular and is often considered when it comes to dynamically balance the work load among processing units. The work stealing principle can be synthesized as follows. An idle thread, called a thief, initiates a steal request to a random selected victim. On reply, the thief receives a copy of one ready task, leaving the original task marked as stolen. Coherency between a thief and its victim is ensured by a Dijsktra-like protocol, as in Cilk [3]. To find a ready task, a thief thread iterates through the victim's queue from the

[1]http://kaapi.gforge.inria.fr

```
/* left looking Cholesky factorization */
for( k=0; k < N; k+= blocksize ) {
  ka::Spawn<TaskPOTRF>()( A(rk,rk) );
  for( m=k+blocksize; m < N; m+= blocksize)
    ka::Spawn<TaskTRSM>()( A(rk,rk), A(rm,rk) );
  for( m=k+blocksize; m < N; m+= blocksize) {
    ka::Spawn<TaskSYRK>()( A(rm,rk), A(rm,rm) );
    for( n=k+blocksize; n < m; n+= blocksize )
      ka::Spawn<TaskGEMM>()( A(rm,rk), A(rn,rk),
           A(rm,rn) );
  } }
```

```
/* Signature defines task parameters */
struct TaskSYRK: public ka::Task<2>::Signature<
    ka::R<ka::range2d<double> >,
    ka::RW<ka::range2d<double> > >{};

template<> struct TaskBodyCPU<TaskSYRK> {
  void operator ( ka::range2d_r<double> A,
                  ka::range2d_rw<double> C )
  { cblas_dsyrk( A→dim(0), A→dim(1), A→ptr(), A→ld(),
                  C→ptr(), C→ld() );
  } };
```

Figure 1. Example of a XKaapi C++ Cholesky factorization (left part). On the right: example of a XKaapi C++ implementation of a task. It shows a task *Signature* with its parameters and access modes, as well as a CPU implementation. The call to `cblas_dsyrk` is simplified.

least recently pushed task to the most recently one and it computes true data-flow dependencies for each task. The iteration stops on the first task found ready.

If a thread pops from its queue a task marked as stolen, then it suspends the task execution and switches to the work stealing scheduler that waits for dependencies to be met before resuming the task. Non-stolen tasks are performed in FIFO order without computation of data-flow dependencies, because the task queueing order on each thread is a valid sequential order of execution [13], [18].

C. Work-first Principle and Data-Flow Dependencies

The main difference between XKaapi and other software [11], [14], [19], [20] is that XKaapi computes data-flow dependencies only when an idle thread searches for a ready task.

Tasks share data if they have access to the same memory region. A memory region is defined as a set of addresses in the process virtual address space. The user is responsible for indicating the mode each task uses to access memory: the main access modes are *read*, *write*, *reduction* or *exclusive* [13], [18]. During a steal operation, the thief thread computes true dependencies (Read after Write dependencies) between tasks according to the access modes. At the expense of memory copy, the scheduler may solve false dependencies through variable renaming.

Computing data-flow dependencies during steal operations reduces the overhead of normal task execution in recursive programs where the number of steals is dependent of the critical path. This original idea in XKaapi follows the work-first principle [3]: at the expense of a larger critical path, XKaapi moves the cost of computing ready tasks from the work performed by the victim during task's creations to the steal operations performed by thieves.

Thanks to our approach, the classical fine-grained recursive Fibonacci in a data-flow implementation shows an overhead T_1/T_{seq} of about 10 [2], which is of the same order as Cilk or TBB that do not handle data-flow dependencies. In XKaapi, the cost of task creation is several orders of magnitude lower than in StarPU [11], StarSs [10] or OmpSs [14].

Tchiboukdjian et al. [21] proposes a theoretical analysis of work stealing with dependent tasks, considering a XKaapi-like protocol.

D. Acceleration Data Structure for Ready Tasks

XKaapi implements an optimization to compute ready tasks at steal operation through the use of a list of ready tasks instead of a work stack. The runtime switches to this new structure when the cost becomes important, especially when the victim's stack contains many tasks, as for instance in the block linear algebra algorithms presented in this paper.

In a steal operation, the scheduler computes a list of tasks' successors from the work stack and attaches it to the stack. The successors of a task are tasks with true dependencies with the task. Subsequently using the successors list, activated tasks are pushed directly into a ready task list. If new tasks are created, the scheduler computes a new list of their successors. Therefore, the search for a ready task, which would be proportional to the number of tasks in the work stack (and to the number k of their parameter, to verify their dependencies), switches to take constant time (access to the first element of the list of ready tasks) plus an $O(k)$ overhead for the activation of the successors. Consequently, subsequent steal operations in a thread with a ready task list have lower cost.

Thus, our optimization moves the overhead of computing ready tasks from each steal operation to the steal operation that detects new tasks in the work stack.

III. Extension for Multi-CPU and Multi-GPU

This section describes the features to support multi-CPU and multi-GPU[2] in XKaapi through multi-versioning task implementation, locality-aware work stealing scheduling, concurrent GPU operations as provided by recent Fermi GPUs, and software cache memory.

[2]Our current version supports NVIDIA CUDA.

```
template<> struct TaskBodyGPU<TaskSYRK>{
  void  operator ( ka::gpuStream stream,
                   ka::range2d_r<double> A,
                   ka::range2d_rw<double> C )
  { cublasDsyrk( kaapi_cublas_handle(stream),
        A→dim(0), A→dim(1), A→ptr(), A→ld(),
        C→ptr(), C→ld() ); } };
```

Figure 2. `TaskSYRK`'s GPU version with the XKaapi C++ API.

A. Versioning Task Implementations

The extensions to the C++ interface provide a high level interface for multi-versioning a task implementation [9]. A task implementation for GPU (respectively CPU) is the specialization of the class `TaskBodyGPU` (respectively `TaskBodyCPU`). For instance the CPU implementation of `TaskSYRK` presented in Figure 1 can be complemented with a GPU implementation as written in Figure 2.

At least one implementation is expected per task signature (`TaskSYRK` in the example). The implementation can be recursive, calling the task signature and leaving to the scheduler the freedom to choose the more relevant implementation. All task implementations must conform to the task signature (`TaskSYRK` in the example), except for one optional formal parameter (here `stream`) that enables to pass information to the task implementation from the caller (during `ka::Spawn`) or from the runtime (here the current CUDA stream where the kernel should be launched).

B. Locality-Aware Work Stealing

We extend each CPU or GPU thread with a local queue named *mailbox* in which remote threads can push tasks. This is similar to the approach proposed in [22], but without explicit locality annotation. Our locality-aware work stealing pushes the successors of a task to selected remote resources (CPU or GPU) based on meta-data information attached to each user data.

We developed two heuristics for local optimization, called **H1** and **H2**, using these meta-data:

- **H1** – For each task to be activated, XKaapi first goes through every task input parameter and looks for the resources where the parameter is valid, and its size. The resource which owns the biggest sum of input bytes in valid state is then chosen as the host to run the task.
- **H2** – This second heuristic is based on the data access modes. It tries to reduce the invalidations of the data replicas: the scheduler pushes a newly activated task on the mailbox of the resource that has a valid copy of its *write* or *exclusive* accessed parameters. If more than one resource is eligible, then the scheduler simply selects a resource at random among the set of eligible ones.

Experiments show that the heuristic *H2* usually makes better local decisions, except for embarrassingly parallel applications, such as matrix product, where they both lead to a similar performance.

C. Asynchronous Task Execution

Once a task is selected, the runtime ensures consistency of its input data on the GPU device before the GPU kernel executes. The runtime assumes that the GPU task implementation launches the GPU kernels asynchronously. Once a task implementation has launched computations on a GPU, the scheduler starts the execution of the next selected task by sending its input data in advance. This enables to overlap data transfers with kernel executions.

We empirically found that the best performance gain is obtained when having two tasks being processed per GPU. Starting more tasks do not increase performance significantly and reduce the capacity to balance the work load, because tasks can not be aborted neither reactivated after the start of a GPU transfer.

Data transfers and kernel invocation on a GPU are handled asynchronously as well as the completion of these operations. We have gathered these functionalities in an extension of CUDA streams presented in the next section.

D. Concurrent GPU Operations

Recent GPUs, such as NVIDIA's Fermi and Kepler, support new features for asynchronism. For instance, Fermi GPUs have one execution engine and two copy engines, enabling to concurrently perform a kernel execution and memory transfers (two-way host-to-device and device-to-host), under the condition that no explicit nor implicit synchronization occurs.

We developed a mechanism to take advantage of this asynchronism for multi-GPU systems. XKaapi splits the execution of a GPU task in two basic operations: host-to-device input transfers (H2D); and `TaskBodyGPU` execution (*i.e.* launch of CUDA kernels) (K). The device-to-host output transfers (D2H) depend on the software cache write-back policy and the scheduling decision of future tasks that access data in read mode.

Since concurrency between data transfers and kernel launches must use CUDA streams, we defined a new data structure, called *kstream*, that groups together three types of CUDA streams: a stream for host-to-device transfer, a stream for kernel execution and a stream for device-to-host transfer. The *kstream* allows to insert a request for one of the three types (H2D, K, or D2H) and to specify a callback function with one argument. After each request insertion, the *kstream* inserts a CUDA event to detect its completion. Once the *kstream* detects the event completion, it calls the callback function with its argument as parameter. This is the responsibility of the *kstream* client to regularly poll for the completion of asynchronous requests by calling a specific function. The XKaapi work stealing algorithm polls each time a GPU thread is idle.

This design allows concurrent execution between each type of CUDA stream. The *kstream* represents three flows of FIFO ordered GPU operations, which execution is independent from each other. The FIFO order is only respected among operations of the same type (H2D, K or D2H). The callback mechanism enables to compose a sequence of operations and is typically used by the GPU work stealing algorithm, first to insert data transfers for the input of a task, and then to invoke the kernel launch when the transfer ends.

E. Data Management and Software Cache

XKaapi manages GPU memory through a software cache, based on the Least Recently Used (LRU) replacement policy. Each GPU thread maintains a FIFO queue of allocated memory blocks. When a GPU task requires accessing a host memory block that is not present on the GPU, the runtime will allocate memory and insert it in its own queue. In order to enable asynchronous memory transfers with CUDA, user data is page-locked through specific CUDA library function (`cudaHostRegister`).

If its memory is full, a GPU tries to evict the least recently used memory block of its own queue (LRU policy). If possible, unused blocks are reused without being freed. This optimization avoids unnecessary CUDA calls.

Consistency is guaranteed by a lazy strategy using a write-back policy. Data transfers to or from the GPU occur only when a task accesses data and when the data is in an invalid state in the target address space. This policy avoids unnecessary transfers, unlike write-through policy [7], [11], [14]. All transfer operations are asynchronous and rely on the use of our *kstream* data structure to signal the completion of operations.

IV. Experiments

All experiments have been conducted on a heterogeneous, multi-GPU system, named “Idgraf“. Idgraf is composed of two hexa-core Intel Xeon X5650 CPUs (12 CPU cores total) running at 2.66 GHz with 72 GB of memory. It is enhanced with eight NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores (scalar processors) running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB total). Figure 3 illustrates the hardware topology of Idgraf. The machine has four PCIe switches to support up to eight GPUs. When two GPUs share a switch, their aggregated PCIe bandwidth is bounded to the one of a single PCIe 16x. Experiments using up to four GPUs always use one GPU per PCIe switch to avoid this bandwidth constraint. On the other hand, experiments using more than four GPUs have to share some pairs of GPUs through the PCIe switch.

We used as software environment GNU/Linux Debian *squeeze* x86/64, the compiler GCC 4.4, CUDA 4.1, and the library ATLAS 3.9.39 (BLAS and LAPACK).

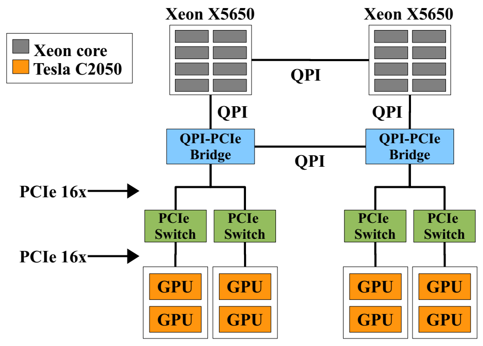


Figure 3. Idgraf hardware topology with two hexa-core CPUs and eight Tesla C2050 GPUs.

A. Dense Linear Algebra Benchmarks

Our experiments use the same parallel version of the dense linear algebra problems matrix product (doing $C \leftarrow C + AB$) and Cholesky Factorization, as found in PLASMA [8]. The algorithms have been re-implemented in XKaapi to use its low overhead task creation. The matrix data layout is the same as in PLASMA (tile data layout). All elementary tasks access to one or more tiles. The parallel Cholesky factorization is a two levels parallel algorithm: at the upper level, we use the PLASMA algorithm with 1024×1024 tiles; at the lower level the panel Cholesky factorization (`DPOTRF`) is parallelized using the same parallel algorithm as at upper level by decomposing one tile in sub-tiles of size 128×128. We have not used auto-tuning to select the sizes of the tile and sub-tile, but an empirical approach: after a few experiments showing their average good performances, we have decided to use theses values.

We calculate the number of Flops according to PLASMA [8] algorithms. If N denotes the dimension of the matrices, the number of Flops for a matrix product is $2N^3$, and $\frac{1}{3}N^3 + \frac{1}{2}N^2 + \frac{1}{6}N$ for the Cholesky factorization. All results, except when specified, are in double precision floating-point operations.

Each result is a mean of 30 executions. The 95% confidence interval is represented on the graphs.

B. Overlapping Data Transfers with Kernel Executions

This section presents experiments to evaluate the capacity of our design to exploit asynchronous data transfers in concurrence with GPU kernel executions. Our experiment measures the performance of the matrix product algorithm. Matrices A and B of dimension $N \times N$ are decomposed into tiles (or blocks) of size $s \times s$. We devised our implementation such that all computations are performed on the GPU. Matrix computation is done with double precision, each block-matrix product launches CUDA kernels using the CUBLAS `DGEMM` routine.

We compare the performance of three versions:

- *CUBLAS (no copy)*: CUBLAS when the time to copy input and output matrices is not considered.

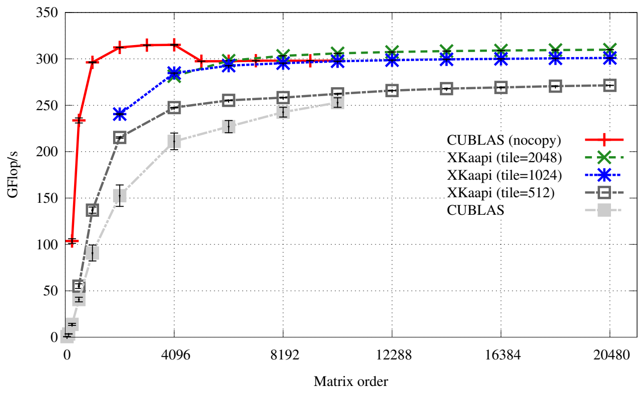


Figure 4. Performance results from DGEMM on Idgraf for a single CPU and a single GPU, and different block sizes.

- *CUBLAS*: CUBLAS with copy time included.
- *XKaapi (tile=s)*: our XKaapi implementation with the performance obtained from native calls to CUBLAS DGEMM on $s \times s$ tile sizes. Each measure includes all the costs of CUDA memory allocations and data transfers.

1) Raw Performance: Figure 4 illustrates the results of DGEMM with the three versions and different tile sizes for XKaapi. Note that *CUBLAS (no copy)* measures the GPU peak performance.

CUBLAS (no copy) reached its peak performance (about 315 GFlop/s) for 4096×4096 square matrices. For larger matrices, the performance decreases to 293 GFlop/s.

Our XKaapi version, that takes data transfers into account, with blocks of size 1024×1024 and 2048×2048, reaches the GPU peak performance for matrices bigger than 6144×6144. For matrices bigger than 8192×8192 XKaapi's implementation sustains 309 GFlop/s for a 2048×2048 block size, outperforming CUBLAS (293 GFlop/s). This algorithm with block size of 1024×1024 generates numerous tasks that can be exploited by our runtime to pipeline and overlap data transfers with computations. Our good performance confirms that we are able to overlap an important amount of the data transfers with the GPU kernel executions.

2) Out of Core Performance: Thanks to the XKaapi software cache and to our design to exploit concurrent GPU operations, our blocked DGEMM algorithm sustains a 309 GFlop/s performance peak even after the GPU runs out of memory with matrices larger than 10240×10240, which require ≈ 2.43 GB of device memory out of the 3 GB available on the NVIDIA Tesla C2050 cards.

3) Taking into Account Copies with CUBLAS: For small matrices, because the number of tasks remains low, the data transfer is not entirely overlapped by computation. Even in this case, XKaapi presents good results. For instance, the performance of *CUBLAS nocopy* with matrices of size 2048×2048 is about 312 GFlop/s. Performance drops to 152 GFlop/s if we take into account the data transfers. Our XKaapi DGEMM for this matrix dimension and with block size of 1024×1024 generates 8 tasks for each sub-matrix product, and it reaches 240 GFlop/s, that corresponds to 157% of improvement over CUBLAS when data transfers are taken into account.

4) Outperforming the CUBLAS GPU Peak Performance: A deeper look at the results presented in Figure 4 shows an interesting phenomenon. For matrix dimension 4096×4096 using a 512×512 block size, our blocked DGEMM reaches 247.5 GFlop/s. For bigger matrix dimensions, using the same block size, the performance increases up to 271 GFlop/s. However, a simple analysis shows that our blocked DGEMM algorithm performance should be upper bounded by CUBLAS DGEMM, since our implementation only calls it to compute each of its blocks. Besides, the same figure shows that the performance of CUBLAS DGEMM, without taking into account data transfers, is 233 GFlop/s for a 512×512 matrix. Here XKaapi enables to increase the performance by 16%. It is still unclear why we get such a gain. It could be related to a better GPU occupancy, or to specific CUDA optimizations for small matrices, which includes new CUBLAS batched $\star$GEMM routines since version 4.1.

In the subsequent experiments, we only consider XKaapi programs with tile size 1024×1024 for Cholesky factorization and 2048×2048 for parallel DGEMM.

C. Comparison of Work Stealing Heuristics

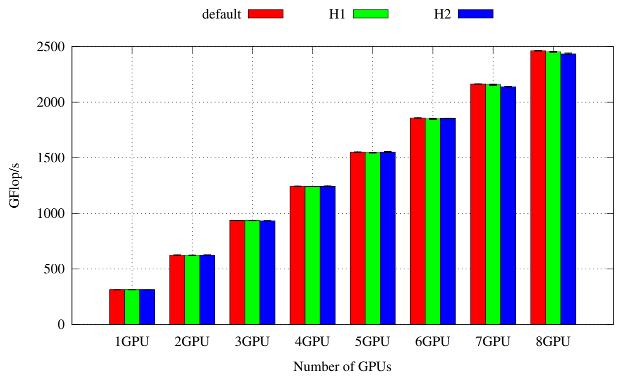
In this section, we compare the performance of the $H1$ and $H2$ heuristics (see section III-B) against the default work stealing algorithm (label *default*) on the matrix product (DGEMM) and Cholesky factorization (DPOTRF). The matrix size is constant (40960×40960) while the number of GPUs varies.

1) Parallel Matrix Product: Figure 5 reports the performance of the parallel DGEMM using up to eight GPUs.

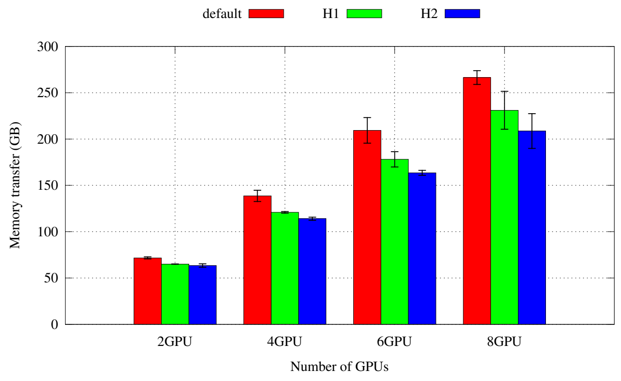
For all heuristics, the speedup linearly increases with the number of GPUs (Figure 5(a)). The peak performance is 2426.40 GFlop/s for eight GPUs (2.43 TFlop/s). This corresponds to a sustained performance of 303 GFlop/s per GPU, which is very close to the peak (315 GFlop/s) on the DGEMM kernel. For matrices of size 16384×16384 XKaapi reaches 2.0 TFlop/s, and 1.6 TFlop/s for matrices of size 8192×8192.

The three heuristics (default, $H1$ and $H2$) show similar GFlop/s performance. When looking at the total amount of data transfered, presented in Figure 5(b), the heuristic $H2$ outperforms the two other approaches with transfers reduced up to 24%. In such a parallel problem, the overlapping capability of XKaapi allows to mask almost all the delays in data transfers. The measured performance is not impacted whatever the chosen heuristic is.

2) Cholesky factorization: Figure 6 illustrates our performance results. In addition to GPUs, we involve in the computations all remaining CPU cores, out of the twelve

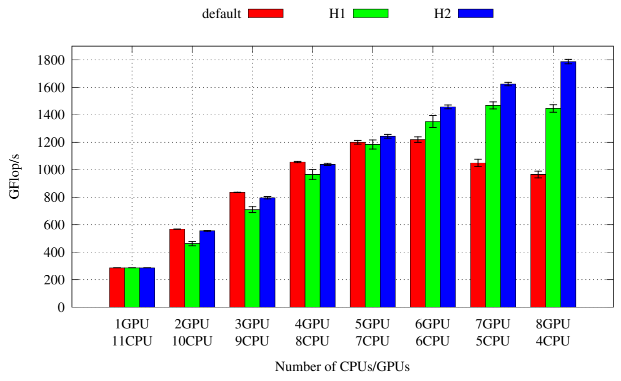


(a) Performance in GFlop/s.

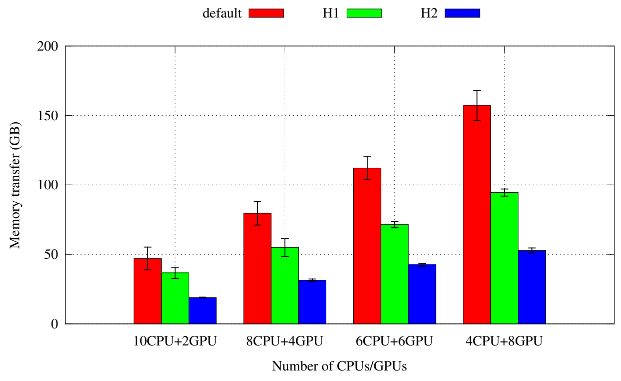


(b) Total data transfers in (GB).

Figure 5. Performance results of DGEMM on eight GPUs for a matrix size of 40960 $\times$ 40960, with three load balancing heuristics.



(a) Performance in GFlop/s.



(b) Total data transfers (GB).

Figure 6. Performance results of DPOTRF on eight GPUs and four CPUs for a matrix size of 40960 $\times$ 40960.

available, after removing the ones each GPU monopolizes to run its GPU thread.

We conclude that: (a) the default heuristic has a bigger communication footprint that explains its poor scalability on more than four GPUs; (b) heuristic $H1$, which reduces the communication volume, enables a gain in scalability up to six GPUs; (c) heuristic $H2$ has the lowest volume of data transfers and scales up to eight GPUs. The peak performance with $H2$ is 1.79 TFlop/s in double precision and 3.92 TFlop/s in single precision.

Note that with more than four GPUs, at least two GPUs share the same PCIe-16x bus. Consequently, a scheduling algorithm that introduces a lot of memory transfer is more penalized on such hardware.

3) Conclusion: The local optimization decisions made by the heuristics do not ensure global reduction of data transfers. The second heuristic $H2$ tries to minimize cache invalidations and seems to be more interesting: its effect is to keep data local to the resources, applying the classical “owner compute rule”. The gain here is that the runtime automatically computes the right device to schedule the tasks without any programmer annotation.

We note that our heuristic allows to obtain very good results. To our knowledge, this is the first time that teraflop performances in double precision are reported on a multi-core machine with up to eight GPUs. Moreover, these results were obtained using a purely work stealing algorithm.

D. Overlapping on Multi-GPU

This section refines the analysis of performance impact when data transfers are overlapped with kernel executions on multi-GPU. Figure 7 shows the performance results of the Cholesky factorization using the default work stealing and our $H2$ heuristic on four CPUs and eight GPUs. For the default work stealing strategy, the overlap improves performance by 160.28 GFlop/s for the largest matrices (40960 $\times$ 40960). The gain is significantly higher for the $H2$ heuristic, where the performance gain is about 550.48 GFlop/s for the largest matrices. In addition, even in the case without any overlapping, $H2$ heuristic improves performance over the default work stealing strategy by 431.45 GFlop/s for the largest matrices (40960 $\times$ 40960). It reinforces our claim that our $H2$ heuristic can enable significant performance gains.

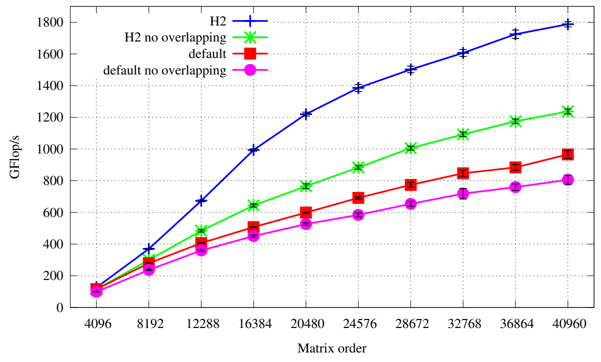


Figure 7. Impact of overlapping in XKaapi default and locality-aware work stealing algorithm with DPOTRF on four CPUs and eight GPUs.

CPUs	Matrix order				
	4096	8192	16384	32768	40960
1	53.85	206.38	622.55	962.21	1052.58
	±0.98	±2.70	±7.90	±31.77	±20.53
4	115.16	391.05	755.91	1013.65	1022.45
	±1.02	±2.64	±6.89	±7.81	±37.55
8	138.34	439.70	782.21	999.46	1045.53
	±1.06	±3.38	±10.51	±6.90	±4.19

Table I
GFLOP/S USING FOUR GPUS AND VARIABLE NUMBER OF CPUS.

E. Multi-CPU Performance Improvement

In this section we analyze the gain of using several CPUs for the Cholesky panel factorization. Table I shows, using heuristic H2, the performance results when using up to eight CPUs and four GPUs for different matrix sizes. Matrices up to a 16384×16384 size show significant performance gains. For matrices larger than 32768×32768 the factorization does not benefit from additional CPUs. The reason for these different performance gains is the influence of the compute-bound tasks in the factorization. Level-3 BLAS operations such as DGEMM dominate the overall execution in a $O(N^3)$ growth order with respect to panel factorizations, which increases in order $O(N)$.

To illustrate executions, Figure 8 displays the Gantt diagrams for two configurations. On the top, one CPU and four GPUs compute the Cholesky factorization of a small matrix of size 6144×6144. This configuration reaches a performance of 122.61 GFlop/s. We can see that GPUs are idle, because they wait for the panel factorization performed by the CPU (the factorization task is on the critical path of the execution). By increasing the number of CPUs to four (bottom part of the figure), the performance increases to 243.80 GFlop/s.

These extra CPUs enable to accelerate the panel factorization with a 8.4 GFlop/s gain per CPU, but more importantly they enable to reduce the idle time of GPUs leading to a global gain of 121.18 GFlop/s. As reported by [6], [8], [15] the acceleration of the tasks on the critical path (panel factorization) is important.

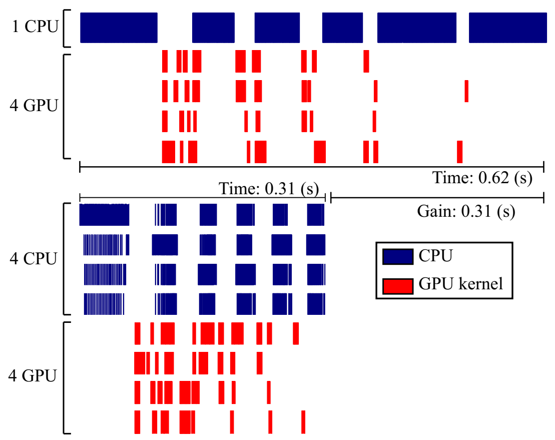


Figure 8. Gantt chart of DPOTRF tasks for a matrix size 6144×6144 on four GPUs (red) and up to four CPUs (blue).

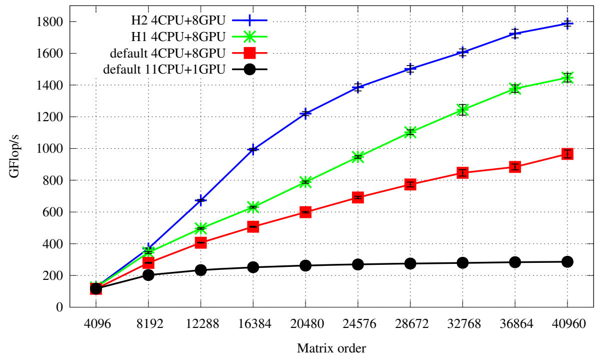


Figure 9. Scalability of the two heuristics and default work stealing with DPOTRF on eight GPUs and four CPUs compared to one GPU and one CPU execution.

F. Scalability of the Cholesky factorization

Figure 9 gives an overview of the performances that have been achieved on the Cholesky factorization for different matrix sizes on eight GPUs and four CPUs using our two heuristics and default work stealing. Except for matrices of size 4096×4096, which results are almost equal, $H2$ gives the best performance for all matrix sizes and scales as the matrix size grows.

V. RELATED WORKS

OmpSs [14] is a programming tool that provides a set of OpenMP-like pragmas and a runtime system to schedule tasks while preserving dependencies. It extended SMPSs [20] and GPUSs [10] by providing simpler code annotations with the capacity to have recursive tasks. OmpSs does not offer any library API to write a program and the user depends of the Mercurium compiler [14]. It offers different scheduling strategies and coherence protocols such as write-back and write-through, but the write-back is on average always the best, as reported in [7]. OmpSs locality-aware scheduling is similar to our $H1$ heuristic. To our knowledge, OmpSs has concurrent execution and data transfers in GPUs but we experienced some issues with matrix sizes that can not be entirely stored into the GPU memory. Moreover, as shown in [14], performances on multi-GPU systems (up to four GPUs) remain difficult to compare with our results since their software environment is different and their experiments only reports results in single precision. No result with Cholesky factorization is reported on multi-GPU/multi-CPU architectures.

StarPU is a runtime system for scheduling a DAG of tasks on heterogeneous architectures optimized for numerical algorithms [11]. In a similar way, StarPU provides a programming model and exposes an API to describe a scheduling policy that allows flexibility in the work distribution. Its API for tasks uses a similar approach to XKaapi called *codelet*. StarPU uses data prefetch to anticipate memory transfers before task execution and provides a lazy coherence protocol. However, each GPU task needs a final synchronization to ensure that all kernels are finished. Besides, its scheduler uses the HEFT [16] algorithm to schedule all ready tasks, thanks to cost models for data transfer and task execution. Such an approach does not allow to react to system load or task execution variations as our work stealing algorithm does. In a recent work [12], we showed that our dynamic work stealing without heuristic (labelled 'default' in experimental section) reaches the same level of performances as StarPU on the matrix product and Cholesky factorization with a sequential panel factorization.

In the context of dense linear algebra algorithms, PLASMA [8] provides fine-grained parallel linear algebra routines with dynamic scheduling through QUARK, which was conceived specially for numerical algorithms. QUARK schedules tasks with data-flow dependencies and suffers from a higher overhead than XKaapi [2]. Recently [19] QUARK has been extended to have the capacity to process *parallel tasks* for which a set of threads are coordinated to execute the same task body. This is different from our approach were we support a single programming model based on recursive tasks to generate more parallelism.

All the above cited software compute dependencies at task creation, even if most of them will never be really used. The lazy dependencies computation of XKaapi is unique.

FLAME [7] is a high-level notation to express algorithms for dense linear algebra operations on multi-CPU/multi-GPU. MAGMA [23] implements static scheduling for linear algebra algorithms on heterogeneous systems composed of GPUs. Recently it has included some methods with dynamic scheduling in multi-CPU and multi-GPU on top of QUARK or StarPU, in addition to the static multi-GPU version.

To the knowledge of the authors, this is the first time that results on classical linear algebra subroutine are reported with more than four GPUs. Moreover, almost all previous reported results with high level of performances on a multi-GPU/multi-CPU are based on static scheduling. In [15], the authors attained about 760 GFlop/s (double precision) for Cholesky factorization using three GPUs (Fermi) and nine CPUs. They based their work on 2D block cyclic distribution with a owner compute rule to map tasks to resources. On the same number of GPUs (Fermi) resources and CPUs, we obtained 837 GFlop/s with the same task-based program, but dynamically scheduled tasks. DAGuE [24] is a parallel framework focused on multi-core clusters and supports single-GPU nodes. The Cholesky factorization performance is only evaluated in single precision where only the `GEMM` task is GPU accelerated.

Our $H2$ heuristic is inspired from [22], but with an automatic scheme to (locally) reduce the number of cache invalidations instead of the explicit annotation of the user code. In SLAW [17] a similar heuristic is experimented. As in [22], the programmer is responsible to explicitly specify the location where his task need to run.

In the context of overlapping on GPUs, Huynh et al. [25] proposed a code-to-code framework on multi-GPU architectures. It represented tasks as a DAG and mapped graph partitions to each GPU statically in order to overlap data transfers with kernel executions.

VI. Conclusion

In this paper, we presented XKaapi, a runtime system for data-flow task programming on heterogeneous architectures. XKaapi enables dynamic scheduling based on work stealing for multi-CPU and multi-GPU architectures. The key contributions of this paper include (1) an original locality-aware work stealing for multi-GPU systems based on local reduction of cache invalidations, (2) a fully asynchronous task execution strategy on GPUs to overlap transfers with kernel executions, (3) a light representation of tasks that allows to generate high degree of parallelism at low cost, (4) a lazy computation of dependencies with an optimization that enables to move the overhead on the critical path rather than on the work.

Our experiments report results up to eight GPUs on classical linear algebra problems. The performance results obtained are about 1.79 TFlop/s for Cholesky factorization and 2.43 TFlop/s for matrix product in double precision. As far as the authors know, this is the best performance in double precision measured on a heterogeneous architecture with up to eight GPUs. Previous related works on the same problems only report results up to four GPUs using static scheduling or static data distribution. For an equivalent configuration our dynamic approach obtains similar — or better — results.

Future works include new experimental evaluations on other linear algebra problems such as LU and QR factorizations with new incoming accelerators such as the future Intel Xeon Phi (Intel MIC) or the next Kepler GT110 GPU.

Acknowledgments

The authors would like to thank Fabien Lementec for providing early implementations on GPU support. This work has been partially supported by the ANR-11-BS02-013 HPAC Project, the ANR 09-COSI-011-05 Project Repdyn, and CAPES/Brazil.

References

[1] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in TBB," in *Proc. of the IEEE IPDPS*, 2008, pp. 1–8.

[2] F. Broquedis, T. Gautier, and V. Danjean, "libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms," in *IWOMP*, Rome, Italy, 2012, pp. 102–115.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors." *Theor. Comp. Sys.*, vol. 34, no. 2, pp. 115–144, 2001.

[6] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 15–44, 2010.

[7] 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," *SIGPLAN Not.*, vol. 44, no. 4, pp. 121–130, 2009.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[9] E. Hermann, B. Raffin, F. c. Faure, T. Gautier, and J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *Proc. of Euro-Par*, vol. 6272. Springer, 2010, pp. 235–246.

[10] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proc. of Euro-Par*, vol. 5704. Springer, 2009, pp. 851–862.

[11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[12] J. V. F. Lima, T. Gautier, N. Maillard, and V. Danjean, "Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs," in *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. New York, USA: IEEE, 2012.

[13] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille, "Athapascan-1: On-line building data flow graph in a parallel language," in *Proc. of PACT'98*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 88–95.

[14] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Proc. of the IEEE IPDPS*, 2012.

[15] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *Proc. of ACM SPAA*. New York, NY, USA: ACM, 2012, pp. 91–100.

[16] C. Boeres, G. Chochia, and P. Thanisch, "On the scope of applicability of the ETF algorithm," in *Proc. of the 2nd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, ser. IRREGULAR '95. London, UK, UK: Springer-Verlag, 1995, pp. 159–164.

[17] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in *Proc. of IEEE IPDPS*, 2010, pp. 1 –12.

[18] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proc. of PASCO'07*. London, Canada: ACM, 2007.

[19] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," University of Tennessee, Tech. Rep. ICL-UT-11-02, 2011.

[20] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Parallelizing dense and banded linear algebra libraries using SMPSs," *Concurr. Comput.: Pract. Exper.*, vol. 21, pp. 2438–2456, 2009.

[21] M. Tchiboukdjian, N. Gast, and D. Trystram, "Decentralized list scheduling," *Annals of Operations Research*, pp. 1–23, 2012.

[22] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proc. of ACM SPAA*, ser. SPAA '00. New York, NY, USA: ACM, 2000, pp. 1–12.

[23] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

[24] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37–51, 2012.

[25] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable framework for mapping streaming applications onto multi-GPU systems," in *Proc. of the 17th ACM PPoPP'12*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 1–10.