Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs

Joao Vicente Ferreira Lima, Thierry Gautier, Nicolas Maillard, Vincent Danjean

Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs

João V. F. Lima*‡, Thierry Gautier†, Nicolas Maillard‡, Vincent Danjean*
* Grenoble University, France
† INRIA, Grenoble, France
‡ Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Brazil
Joao.Lima@imag.fr, thierry.gautier@inrialpes.fr, nicolas@inf.ufrgs.br, Vincent.Danjean@imag.fr

***Abstract*—The race for Exascale computing has naturally led the current technologies to converge to multi-CPU/multi-GPU computers, based on thousands of CPUs and GPUs interconnected by PCI-Express buses or interconnection networks. To exploit this high computing power, programmers have to solve the issue of scheduling parallel programs on hybrid architectures. And, since the performance of a GPU increases at a much faster rate than the throughput of a PCI bus, data transfers must be managed efficiently by the scheduler.**

This paper targets multi-GPU compute nodes, where several GPUs are connected to the same machine. To overcome the data transfer limitations on such platforms, the available softwares compute, usually before the execution, a mapping of the tasks that respects their dependencies and minimizes the global data transfers. Such an approach is too rigid and it cannot adapt the execution to possible variations of the system or to the application's load.

We propose a solution that is orthogonal to the above mentioned: extensions of the XKaapi software stack that enable to exploit full performance of a multi-GPUs system through asynchronous GPU tasks. XKaapi schedules tasks by using a standard Work Stealing algorithm and the runtime efficiently exploits concurrent GPU operations. The runtime extensions make it possible to overlap the data transfers and the task executions on current generation of GPUs. We demonstrate that the overlapping capability is at least as important as computing a scheduling decision to reduce completion time of a parallel program.

Our experiments on two dense linear algebra problems (Matrix Product and Cholesky factorization) show that our solution is highly competitive with other softwares based on static scheduling. Moreover, we are able to sustain the peak performance ($\approx$ 310 GFlop/s) on DGEMM, even for matrices that cannot be stored entirely in one GPU memory. With eight GPUs, we archive a speed-up of 6.74 with respect to single-GPU. The performance of our Cholesky factorization, with more complex dependencies between tasks, outperforms the state of the art single-GPU MAGMA code.

I. Introduction

Current processors are homogeneous chips containing many cores, whose number will increase again in the near future. The architectural trend is the emergence of hybrid systems with many tightly coupled processing units (PU) such as GPUs or accelerators (old Cell BE/Intel MIC) for high performance computing. During the last year, the number of GPU-based computers in the Top500 list has increased from 3.8% to 11% (from June, 2011 to June, 2012).

Such architectures have heterogeneous PUs in terms of computing power and programming model. The programmer is in charge of execution flow and memory consistency. Hence, in many cases, familiar algorithms need to be redesigned.

Algorithms in dense linear algebra, such as those found in the LAPACK library, and especially matrix factorizations, have already been redesigned to exploit multicore machines. The FLAME [1] and PLASMA [2] projects have demonstrated how much more interesting it is to exploit parallelism among the BLAS operations, than inside a given BLAS operation itself. These new algorithms are built on a software stack that allows to describe tasks with dependencies and to schedule them at runtime on multicore.

With hybrid architectures, this software has been extended to develop hybrid algorithms with multiple task implementations optimized for each kind of PU. MAGMA [3] allows to exploit one GPU; MAGMA/StarPU [4] reports experiments up to four GPUs; FLAME [1] also shows experiments on four GPUs. The first main difference with previous homogeneous architecture is that heterogeneous machine has non-uniform processing power elements. The scheduling theory community considers a heterogeneous machine as an *unrelated machine* where scheduling a task graph to minimize the makespan is a well-known NP hard problem. The second difference is that a heterogeneous architecture introduces a new memory level (on the accelerator), which is non cache-coherent with the main memory. Moreover, the bandwidth of PCI-Express interconnect between the host and the accelerator memory remains low, about 8 GB/s on PCIe x16, *vs.* the 32 GB/s between a CPU core and the main memory.

To reach high performance, it is essential to reduce the time required to transfer data. A good schedule may take the right decision to map a task onto a GPU resource which already stores data to be reused. A good runtime may also try to exploit the high capability to overlap communication with computation on the modern GPUs. There are many publications that report experiments on multi-GPU systems on some linear algebra factorization algorithms [3], [4], [5]. The authors report higher performance by computing at runtime a static schedule of the task graph coupled with overlapping strategy during execution. Such performances come at the expense of being able to compute cost models of the task graph and of data transfers.

Nevertheless, if the communication cost can be entirely overlapped, then classical dynamic work stealing [6] with the-

oretical performance guarantee should be almost as efficient, without requiring cost models anymore, and it can react well to inactivity due to system or application load variations.

This paper presents extensions of the XKaapi runtime that enable to exploit the full performance of multi-GPU systems by using a standard work stealing algorithm. We compare our design decisions with StarPU [7], that computes a static schedule of the task graph using an ingenious automatic tool to build cost models.

Our main contribution is to demonstrate that, on current GPUs, the overlapping capability is at least as important as computing a good scheduling decision to reduce the completion time of a parallel program. Moreover, because XKaapi uses an online scheduling algorithm, it can be performant for a wide range of applications, including those irregular, when the cost of the tasks depends of their input.

First, we evaluate raw performances on matrix product (`DGEMM`) and Cholesky factorization for single-GPU systems. Second, we study the scalability of our implementation on up to 8 GPUs, and compare it with state-of-the-art libraries. Finally, we complete our initial experiments with various grain and problem sizes, and with the volume of data transfers, in order to estimate the limitation of pure work stealing for multi-GPUs. Our experiments show that we are able to sustain the peak performance on `DGEMM`, even for matrices bigger than the GPU memory, with a 6.74 speed-up on 8 GPUs (relatively to 1 GPU), which is better than the other tools. For the Cholesky factorization, our XKaapi version outperforms the state of the art MAGMA single-GPU code; it suffers from the same scalability limitations as StarPU on multi-GPUs.

The remainder of this paper is organized as follows: in Sections II and III we describe the XKaapi programming model and the designed runtime extensions to support multi-GPUs. We discuss our performance results in Section IV. In Section V we present related works on runtime tools for GPUs and linear algebra (LA) libraries. We conclude and present future works in Section VI.

II. Data Flow Task Programming with XKaapi

The XKaapi[1] task model [8], as in Cilk [6], Intel TBB [9], OpenMP-3.0 [10] or StarSs [11], [12], enables non-blocking task creation: the caller creates the task and continues the program execution. The semantic remains sequential like in XKaapi's predecessor Athapascan [13], but the runtime has been redesigned [8] and then specialized for multi-CPUs/multi-GPUs iterative applications [14]. Here we present the extension of our previous work on multi-CPUs to a general scheduling algorithm on multi-GPUs.

A. Design Choices

More than a runtime, XKaapi is a fully featured software stack to program hybrid parallel architectures. The core stack is written in C and was designed using a bottom up approach: each layer is kept as specialized as possible to fit a specific need. Currently, the stack includes: a runtime supporting multicores and multiprocessors; a set of ABIs (QUARK [15], OpenMP runtime libGOMP [16]); a set of high level APIs (C, Fortran and C++); and a source to source compiler [17] based on the ROSE compiler framework. Only the C++ API currently supports multi-CPUs/multi-GPUs applications, and we will use it in the code fragments of this paper.

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

B. Data Flow Task Model

A XKaapi program is composed of sequential code and some annotations or runtime calls to create tasks. The parallelism in XKaapi is explicit, while the detection of synchronizations is implicit [8]: 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 the shared memory and the list of its effective parameters. Depending of the APIs, tasks are created using code annotation (`#pragma kaapi task` directive) if the XKaapi compiler [17] is used, or by library function (`kaapic_spawn` call using XKaapi's C API, or by calling the template function `ka::Spawn`), or by low level runtime function calls.

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. This set has the shape of a multidimensional array. 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], [8], [17]. When required ([8]), the runtime computes true dependencies (Read after Write dependencies) between tasks thanks to the access modes. At the expense of memory copy, the scheduler may solve false dependencies through variable renaming.

A thread creates tasks and pushes them on its own work queue. The work queue is represented as a stack. The enqueue operation is very fast, typically about ten cycles on the last x86/64 processors. As for Cilk, a running XKaapi task can create child tasks, which is not the case for the other data flow programming softwares previously mentioned [15], [12], [7], except the recent StarSs extension OmpSs [11]. Once a task ends, the runtime executes its children following a First-in First-out (FIFO) order. During task execution, if a thread finds a stolen task, it suspends its execution and switches to the work stealing scheduler that waits for dependencies to be met before resuming the task. Otherwise, and because sequential execution is a valid order of execution [13], [8], tasks are performed in FIFO order without computation of data flow dependencies.

C. Blocked Linear Algebra Algorithms with XKaapi

Previous works have shown that fine granularity and asynchronism are keys to unfold parallelism on hybrid architectures [2], [3]. The "hybridization" strategy for linear algebra algorithms decomposes factorizations in a collection of BLAS-based fine-grained tasks with dependencies among them. The tasks can then be properly scheduled on the available resources.

```
for( k=0; k < N; k+= blocsize ){
  ka::Spawn<TaskPOTRF>()( A(rk,rk) );
  for( m=k+blocsize; m < N; m+= blocsize)
    ka::Spawn<TaskTRSM>()( A(rk,rk), A(rm,rk) );

  for( m=k+blocsize; m < N; m+= blocsize){
    ka::Spawn<TaskSYRK>()( A(rm,rk), A(rm,rm) );
    for( n=k+blocsize; n < m; n+= blocsize )
      ka::Spawn<TaskGEMM>()( A(rm,rk), A(rn,rk),
          A(rm,rn) );
  }
}
```

Fig. 1. Example of a XKaapi C++ Cholesky factorization.

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 sub-matrix of elements `A(i,j)` where `i,j` are in the range `rm,rk`.

Let us note that a sub-matrix does not need to be contiguous in memory as with OmpSs [11]. It allows faster code parallelization, even if better performance may be obtained at the cost of converting row major or column major matrix representation to block representation.

D. Versioning Task Implementations

Extensions to the C++ interface provide a high level interface for multi-versioning a task implementation [14]. First, a task is associated with a signature that includes the task parameters and their access mode (read `R` and/or write `W` and/or concurrent write `CW`). Each CPU or GPU implementation is encapsulated in a functor object, which must respect its task signature. This concept of multi-versioning and task implementation allows a clear separation between the task definition and its implementations. StarPU uses a similar approach called *codelet* [7]. Thus, architecture details are abstracted from the algorithm. Moreover, the signature allows the runtime to automatically take care of memory transfers in case of distributed memory. The application programmer does not need to code any explicit memory transfer. Figure 2 shows an example of a task with CPU (*TaskBodyCPU*) and GPU (*TaskBodyGPU*) implementations conforming to its *Signature*. The runtime expects at least one implementation of a task signature.

III. XKaapi Runtime Support to Multi-GPUs

The XKaapi runtime extensions implement a programming model that offers asynchronous execution of GPU tasks and abstracts memory details. Algorithms on top of XKaapi describe the execution flow through the task dependencies and the runtime decides the target resource (CPU or GPU) and performs memory transfers as necessary. Our current version supports NVIDIA CUDA and it relies on the features of recent Fermi GPUs. This section describes the features to support

```
/* 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 )
  { /* CPU implementation */ }
};

template<> struct TaskBodyGPU<TaskSYRK>{
  void operator( ka::gpuStream stream,
                 ka::range2d_r<double> A,
                 ka::range2d_rw<double> C )
  { /* GPU implementation */ }
};
```

Fig. 2. Example of a XKaapi C++ program for hybrid architectures. It shows a task *Signature* with its parameters and access modes, as well as CPU and GPU implementations.

multi-GPUs in XKaapi through scheduling, concurrent GPU operations and memory abstraction.

A. Work Stealing Scheduling Algorithm

The XKaapi version for multicore architectures implements a list scheduling algorithm based on Cilk's Work Stealing and has specific optimizations for fine-grained parallel algorithms, which have been sketched in section II-B. For each used GPU, the runtime launches a thread on the host machine that runs a modified work stealing algorithm.

For each task executed on a GPU, the runtime first transfers the input data and allocates memory on the GPU for the output. The runtime assumes that the GPU task implementation launches the GPU kernels asynchronously. Data transfers and kernel launch on GPU are handled asynchronously by an extension of CUDA streams described in the next section.

In comparison with original multi-CPUs work stealing, multi-GPUs work stealing adds a new state in the task state diagram that corresponds to a task for which input data are under transfer. The GPU thread polls regularly the completion of previous asynchronous GPU operations.

A task that completes its execution, when the asynchronous kernel launch has completed, activates the successor tasks (according to the data flow dependencies) that become ready. These new ready tasks are pushed on the tasks' queue attached to the current GPU and they may be stolen by one CPU or another GPU.

B. Concurrent Operations between CPU and GPU

Recent GPUs such as NVIDIA's Fermi allow new techniques to explore asynchronism in multi-GPU systems. Fermi GPUs have one execution engine and two copy engines capable of concurrent execution and transfers (two-way host-to-device and device-to-host), under the condition that no explicit nor implicit synchronization occurs. This section details how we exploit these capabilities.

XKaapi has an execution strategy for GPUs that avoids CUDA's implicit synchronizations and exploits concurrent memory transfers in the two ways along with kernel execution. It splits the execution of a GPU task in three basic operations: host-to-device input transfers (H2D), *TaskBodyGPU* execution (*i.e.* launch of CUDA kernels) (K), and device-to-host output transfers (D2H). Write-only parameters are directly allocated before launching CUDA kernel.

Since concurrency between data transfers and kernel launches must use CUDA streams, we define 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** structure allows to insert a request for one of the three types it handles (H2D, K, or D2H). A callback function and its argument can be specified for each request insertion. Moreover, after each request insertion, the **kstream** inserts a CUDA event to detect the completion of the asynchronous operation. Once the **kstream** detects the event completion, it calls the callback function with its argument as parameter. It is the responsibility of the client of the **kstream** structure to regularly poll for the completion of asynchronous requests by calling a specific function.

Figure 3 illustrates the way our **kstream** structure allows to pipeline concurrent operations on a Fermi GPU.

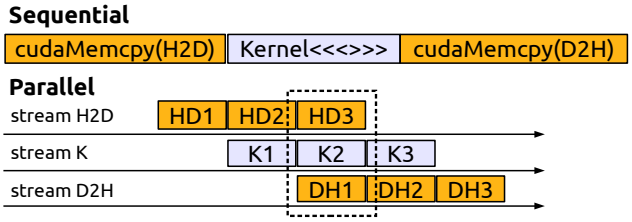

Fig. 3. Sequential and concurrent operations in a Fermi GPU card. Fermi GPUs have two copy engines and one execution engine capable of concurrent transfer operations (host-to-device and device-to-host) and kernel execution.

This design allows concurrent execution between CUDA streams of each type. The **kstream** represents three flows of FIFO ordered GPU operations whose execution are independent from each other. The FIFO order is only respected among operations of the same type (H2D, K or D2H). The callback mechanism permits to compose a sequence of operations and it 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.

C. Memory Management

In order to enable asynchronous memory transfers with CUDA, all the user data must register to XKaapi's runtime.

XKaapi manages GPU memory through a software cache, based on the Least Recently Used (LRU) replacement policy. Each GPU thread maintains two FIFO queues in order to keep track of allocated blocks. One queue stores blocks in read-only (RO) mode and the other stores blocks in read-write (RW), write-only (WO) modes or concurrent write (CW) modes. The first positions of the RO and RW/WO queues contain the blocks more recently accessed, and the last positions the blocks less recently accessed. When a GPU task requires to access a host memory block that is not present on the GPU, the runtime will allocate memory and insert it in one of the two queues, based on its access mode, after it has initiated the data transfer (for data in read access mode). If the GPU memory is full, it verifies first at the end of the RO queue and, then, into the RW/WO queue, respectively, if a memory block bigger or equal than the requested size is not accessed anymore. If possible, unused blocks are reused without being freed. Otherwise, it may free blocks from RO and RW/WO/CW queues as needed. This optimization avoids unnecessary CUDA calls. Furthermore, the use of two queues (RO and RW/WO/CW) ensures that data produced by one task in RW/WO/CW mode remains on the GPU if RO data can be reclaimed.

Consistency is guaranteed by a lazy strategy using a write-back policy. Data transfers to or from 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. All transfer operations are asynchronous and rely on the use of our **kstream** data structure to signal the completion of operations. In the case of GPU-to-GPU transfers, the runtime first performs a transfer device-to-host from a GPU with a valid copy, followed by a host-to-device transfer to the GPU that owns the task. Since its version 4.1, CUDA includes transfers between two GPUs directly by DMA called Peer-to-Peer Device Access. This feature is available when the function `cudaDeviceCanAccessPeer` returns true. The current version of XKaapi does not make use of this feature, because of its unpredictable behaviour concerning the GPU copy engines: with it, we could not guarantee the coherence of concurrent data copies, nor their overlapping with a kernel execution.

IV. Experiments

The experiments of this section show respectively: XKaapi's ability to overlap the communication with tasks; a comparison of XKaapi's performance with state-of-the art tools, and on Multi-GPUs platforms; an analysis of the granularity and its impact on the performance with XKaapi; and, finally, how XKaapi uses efficient data transfers.

All experiments have been conducted on an hybrid, multi-GPUs system, which is named "Idgraf". Idgraf is composed of two six-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 running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB total). Figure 4 illustrates the hardware topology of Idgraf.

All experiments using up to 4 GPUs always run on GPUs that do not share any PEX 8647 multiplexer in order to reserve the full PCIe 16x bandwidth for each pair GPU/CPU. Experiments using more than 4 GPUs lead to having some

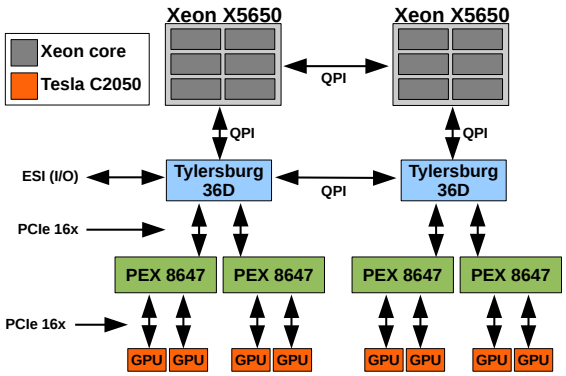

Fig. 4. Idgraf hardware topology with two six-core CPUs and eight GPUs. Here the Tylesburg-36D is a QPI-PCIe bridge and the PXE8647 is a PCIe switch for two GPUs.

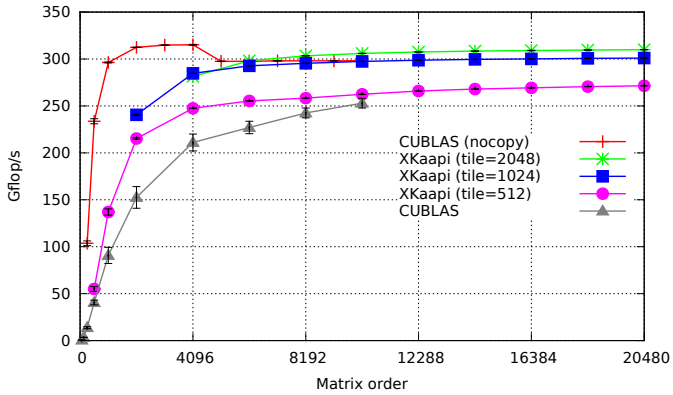

Fig. 5. Performance results from DGEMM on Idgraf for single CPU and single GPU and different block sizes.

pairs of GPUs share the PCIe 16x between the Tylersburg chip and the PEX multiplexer.

Our experiments use the dense linear algebra problems Matrix Product (DGEMM doing $C \leftarrow C + AB$) and Cholesky Factorization, as PLASMA [2] (the same algorithms have been implemented in XKaapi). As in [4], we consider the peak performance from DGEMM (from CUBLAS) as an upper bound on the actual performance in double precision (DP) that one may obtain with GPUs. We used as software environment the compiler GCC 4.4 and CUDA 4.1, and the library ATLAS 3.9.39 for the CPU versions of BLAS and LAPACK. We also used MAGMA 1.1.0 for linear algebra algorithms and StarPU 1.0.1 (with its HEFT scheduling algorithm) as performance references. Each result is a mean of 30 executions, which is enough for StarPU to calibrate its internal cost model. The standard deviation is represented on the graphs.

A. Overlapping Data Transfers with GPU 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 runtime of a block matrix product algorithm. Matrices A and B are decomposed into $k \times k$ blocks of size $s \times s$. We devised our implementation such that all the 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 performances of our XKaapi implementation with the performances obtained by native calls to CUBLAS DGEMM on the whole $ks \times ks$ matrices, taking into account the time to copy the matrices to/from the device or not.

Figure 5 illustrates the results of DGEMM with XKaapi; with CUBLAS when the time to copy the matrices is not accounted for (label *CUBLAS (no copy)*), that measures the GPU peak performance; and with CUBLAS with copy time included. Each measure of the XKaapi runtime includes all the costs of CUDA memory allocations and data transfers. XKaapi experiments always take into account all data transfers, including those required to get final results in main memory.

First, the peak performance from *CUBLAS nocopy* is about 315 GFlop/s for square matrices of dimension 4096. Then, the performance decreases to 293 GFlop/s for bigger matrices.

Our XKaapi version, with blocks of size 1024 and 2048, reaches the GPU peak performance except for small matrices. For matrices bigger than 8192 XKaapi's implementation sustains 309 GFlop/s with block of size 2048, which is more than the GPU peak DGEMM from CUBLAS (293 GFlop/s). This algorithm with block size of about 1024 generates numerous tasks that can be exploited by our runtime to pipeline and overlap data transfers with computation. Our good performances confirm that we are able to overlap a very high percent of the data transfers with GPU kernel.

Moreover, thanks to the XKaapi software cache and to our design to exploit concurrent GPU operations, our blocked DGEMM algorithm sustains its 309 GFlop/s performance peak even after the GPU runs out of memory with matrix orders larger than 10240, which requires ≈ 2.4 GB of device memory out of 3 GB on Tesla C2050 cards.

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 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 generates 8 tasks for each sub-matrix product, and it reaches 240 GFlop/s, that correspond to 157% of improvement over CUBLAS when data transfers are taken into account.

B. Surpassing CUBLAS GPU Peak

A deeper look on the results presented in Figure 5 shows an interesting phenomenon. For matrix dimension 4096×4096 using block size 512, our blocked DGEMM reaches 247.5 GFlop/s. For bigger matrix dimension, 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. Thus, the XKaapi version increases the performance by 16%.

We are not certain about the actual factor that improves the performance on the Fermi GPUs. One hypothesis is that the gain is explained by better GPU occupancy, or specific CUDA optimizations for small matrices, which includes new CUBLAS batched `*GEMM` routines since version 4.1.

C. Cholesky Single-CPU/Single-GPU

Figure 6 reports our results using one CPU and one GPU for Cholesky factorization. We compare our work stealing based runtime XKaapi to StarPU [7] and single-GPU MAGMA [3]. StarPU [7] schedules at runtime the entire task graph using the HEFT static scheduling algorithm. In XKaapi and StarPU, the Cholesky factorization of the diagonal block is sequential and executed on the CPU. The MAGMA version uses a more sophisticated implementation where part of the diagonal block factorization is exported on the GPU.

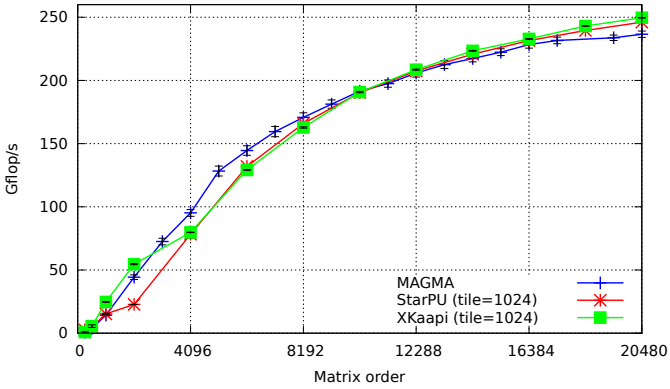

Fig. 6. Cholesky performance results on Idgraf for single-CPU and single-GPU with block size 1024×1024.

XKaapi and StarPU, with runtime scheduling decisions, outperform MAGMA when the matrix dimension is bigger than 10240. The whole matrix size is 800 MB, and can be stored into the 3 GB of device memory. Only the last matrix of dimension 20480 can not be stored into the GPU memory. The main difference between XKaapi and StarPU *vs* MAGMA is that MAGMA is unable to exploit parallelism across the main iteration (see code fragment of Figure 1), as is done by XKaapi and StarPU.

For small matrix dimensions (less than 2048), the performance of MAGMA and XKaapi are similar, but StarPU seems to suffer from a higher overhead. XKaapi has a little drop and then reaches the performances of StarPU.

D. Multi-GPUs Results

For the multi-GPUs evaluation, our experiments measure the performance of `DGEMM` and Cholesky using from one to eight GPUs, with matrix dimension of 16384 and block size of 1024. XKaapi is compared again to StarPU. Figure 7 shows the performance results for `DGEMM`. XKaapi outperforms StarPU in all cases and attains 2023.14 GFlop/s (or speed-up 6.74 on 8 GPUs with respect to single-GPU).

Figure 8 reports the results for the Cholesky factorization. The algorithm corresponds to the code fragment in Figure 1. In

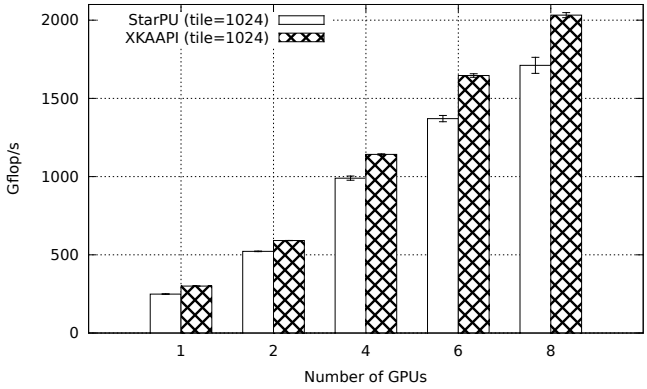

Fig. 7. `DGEMM` performances up to 8 GPUs. The matrix order is 16384 with block size 1024×1024.

the StarPU and XKaapi programs, all tasks, except the block factorization `TaskPOTRF`, are performed by a GPU. Unlike the `DGEMM` case, the Cholesky factorization acceleration, up to eight GPUs, is below the expected: neither XKaapi nor StarPU implementations do scale. StarPU reaches 680.82 GFlop/s (or speed-up 2.94 with respect to single-GPU). Experiments with bigger matrices (up to 20480×20480) show the same behaviors. This means that, when using more that 4 GPUs, communications costs can not be neglected.

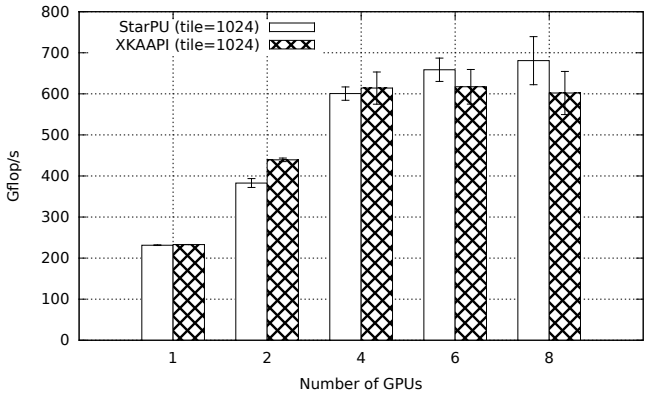

Fig. 8. Cholesky factorization performances up to 8 GPUs. The matrix order is 16384 with block size 1024×1024.

E. Granularity Impact

In this set of experiments we vary the matrix dimension and the block size for our blocked `DGEMM` with StarPU and XKaapi. The results of Figure 9 permit to conclude that our runtime, with dynamic work stealing scheduler, has a lower overhead than StarPU. Thus, XKaapi is able to exploit more performance on `DGEMM`. The overhead is related to the number of tasks in the system: for matrix dimensions greater or equal than 10240 and for block size greater or equal than 1024, the difference between XKaapi and StarPU remains about the same in all the cases.

F. Memory Transfers

XKaapi and StarPU allow to monitor the execution by collecting *post-mortem* traces of performance counters. We have collected, for one instance of `DGEMM` and of the Cholesky

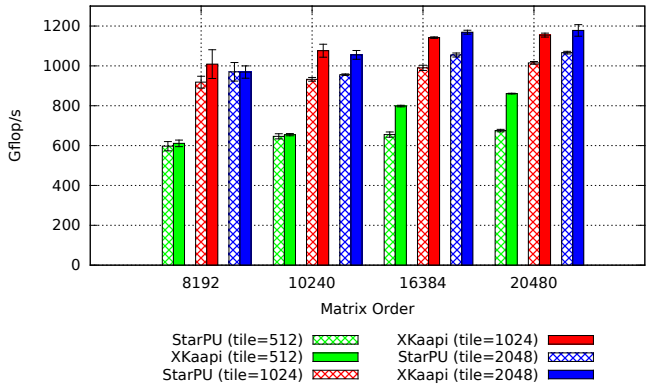

Fig. 9. Results for DGEMM using 4 GPUs on various grains.

factorization, the total number of bytes exchanged between the main memory and the GPUs.

Table I shows in GB the total memory transfers with StarPU and XKaapi on the DGEMM code. The matrix dimension is 16384. Surprisingly, StarPU generates bigger data exchanges than XKaapi up to 4 GPUs, although it uses an *a priori* HEFT algorithm that minimizes data transfers. For a matrix of size S bytes, if the GPU memory could store the entire data, DGEMM implies $3 \times S$ bytes from host to device transfer and S bytes of transfer to get back the result. For a DP matrix of dimension $16384 = 2^{14}$, the data transfer volume is 8 GB. On one GPU, XKaapi's schedules order the tasks according to the sequential order, which is memory efficient. However, StarPU's schedule seems to order the execution using a breadth-first strategy, generating too much parallelism, and thus it increases the memory consumption of the device. Then, the StarPU software cache eviction policy generates a lot of traffic that degrades the performances.

	DGEMM transfers (GB)				
GPUs	1	2	4	6	8
XKaapi	8.00	10.27	16.49	23.23	29.29
StarPU	22.54	14.97	16.98	20.09	24.35

TABLE I
MEMORY TRANSFERS OF DGEMM IN GIGABYTES WITH MATRIX ORDER 16384 AND BLOCK SIZE 1024 × 1024. THE SUM OF INPUT AND OUTPUT DATA TRANSFERS IS 8 GB.

On the Cholesky factorization, as illustrated in Table II, the memory transfer generated by XKaapi is larger than StarPU's. As with the DGEMM program, the scheduling strategy used by XKaapi orders the tasks close to the sequential order of their creation, which generates too much parallelism and thus too much data consumption from the device. The HEFT scheduling of StarPU minimizes the makespan and reduces the memory consumption.

We previously showed that bad scaling of Cholesky factorization exhibited on figure 8 was due to bad overlap of communication by computation. The table II explains why XKaapi performs worst than StarPU: it has more data transfers. Moreover, when using more than 4 GPUs, the architecture share some PCIe 16x links between GPUs (Figure 4), so

	Cholesky transfers (GB)				
GPUs	1	2	4	6	8
XKaapi	3.71	7.38	12.28	12.89	14.62
StarPU	2.23	3.81	6.55	7.38	9.00

TABLE II
MEMORY TRANSFERS OF CHOLESKY IN GIGABYTES WITH MATRIX ORDER 16384 AND BLOCK SIZE 1024 × 1024. THE SUM OF INPUT AND OUTPUT DATA TRANSFERS IS 4 GB.

bottlenecks on data transfers become worst.

Using work stealing directed by data affinity would allow XKaapi to reach StarPU performances. But the amount of data transfers would still be a bottleneck. To overcome this limitation, one would need to use bigger blocks on GPU. Still, using bigger blocks means that TaskPOTRF would become a bottleneck as this task is currently run only on CPU. A parallel implementation, partially on GPU (such as MAGMA), for this task would then be required.

V. RELATED WORKS

OmpSs [11] is a programming tool that provides a set of OpenMP-like pragmas and a runtime system to schedule tasks while preserving dependencies. It offers different scheduling strategies and coherence protocols such as write-back and write-through. To our knowledge, OmpSs has concurrent execution and data transfers in GPUs but it shows some issues with matrix sizes that can not be entirely stored into the GPU memory. It also performs asynchronous operations in GPUs by pinned-memory buffers, which adds additional memory copies and transfer overheads. Moreover, as shown in [11], performance on multi-GPU systems remain difficult to compare with our results: the public downloaded version seems to have some problems on multi-GPUs and the performances reported in the paper achieve about 400 GFlop/s on single precision matrix product while we measured about 625 GFlop/s for the same problem size on similar Fermi GPU processor. The cited CUDA 3.2 version used by the authors is too old for correct comparisons.

StarPU is a runtime system for scheduling a DAG of tasks on hybrid systems optimized for numerical algorithms [7]. In a similar way, StarPU provides a programming model for hybrid architectures and exposes an API to describe a scheduling policy which allows flexibility in work distribution. Recently StarPU has supported concurrent write (CW) access of task parameters. It has similar features to XKaapi but StarPU lacks of concurrent operations on GPUs. It uses data prefetch to predict memory transfers before task execution and provides a lazy coherence protocol. However, according to StarPU examples, each GPU task needs a synchronization at the end to ensure all kernels are finished. It forces a synchronization point in the GPU copy engines and does not allow concurrent GPU operations. This issue could not be significant in earlier GPU families, but it is crucial on Fermi GPUs. Besides, its scheduler uses the static HEFT algorithm to schedule the entire DAG thanks to cost models for data transfer and task executions. Such an approach does not allow to react to system load or task execution variations as our work stealing algorithm does.

In the case of linear algebra algorithms, PLASMA [2] provides fine-grained parallel linear algebra routines with dynamic scheduling through QUARK, which was conceived specially for numerical algorithms. FLAME [1] is a high-level notation to express algorithms for dense linear algebra operations on multi-CPUs/multi-GPUs. MAGMA [4] implements static linear algebra algorithms for hybrids systems composed of GPUs. Recently it has included some methods with dynamic scheduling in multi-CPUs and multi-GPUs on top of QUARK or StarPU, in addition to static multi-GPUs version. NVIDIA CUDA Toolkit provides some BLAS routines with CUBLAS [18].

VI. Conclusion

We have presented extensions of the XKaapi runtime system to program multi-GPU architectures. The current version provides work stealing that efficiently exploit concurrent GPU operations. Our experimental results have been obtained on blocked matrix product (`DGEMM`) and the Cholesky factorization, with an hybrid system composed of eight Fermi-based NVIDIA GPUs.

Our main contribution is the design of an asynchronous approach of concurrent GPU operations that achieves an almost ideal overlapping of data transfer and kernel execution with `DGEMM` algorithm for single-GPU. Thanks to this overlapping, the use of a pure dynamic work stealing algorithm permits to reach high performance as the theory predicts for shared memory machine without communication costs. Thus, overlapping almost enables to hide the heterogeneity of the memory accesses on a multi-GPUs system.

Second, our work stealing implementation outperforms in most cases the static MAGMA approach for Cholesky factorization on single-GPU and the static HEFT scheduler from StarPU. We obtained significant performance results with both XKaapi and StarPU for the Cholesky implementation. But, our scheduling strategy makes decisions at runtime and does not rely on any cost model.

Nonetheless, our scheduling strategy based on work stealing lacks of more sophisticated decisions in order to consider data locality and processing power of available PUs. In our experiments with blocked Cholesky factorization, StarPU attains better results for executions with more than 4 GPU because its scheduling strategy considers communication costs. Coupling our online scheduling with data locality or communication information might enable to get a scalable Cholesky implementation on multi-GPUs.

As future works, we plan to extend XKaapi with an OpenMP-like interface for GPUs. Also, we will design dynamic strategies to take into account the affinity between tasks and data in order to reduce the data transfers in multi-GPU executions, as well as new adaptive algorithms using XKaapi for Cholesky, QR and LU factorizations.

VII. 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 ANR Project, and CAPES/Brazil.

References

[1] 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," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504196

[2] 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.

[3] 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.

[4] 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 International Parallel & Distributed Processing Symposium (IPDPS 2011)*, EUA, 2011.

[5] M. Horton, S. Tomov, and J. Dongarra, "A class of hybrid lapack algorithms for multicore and gpu architectures," in *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, ser. SAAHPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 150–158. [Online]. Available: http://dx.doi.org/10.1109/SAAHPC.2011.18

[6] 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. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[7] 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.

[8] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *Proceedings of PASCO'07*. New York, NY, USA: ACM, 2007.

[9] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[10] OpenMP Architecture Review Board, "http://www.openmp.org," 1997-2008.

[11] J. Bueno, J. Planas, R. M. B. Alejandro Duran, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *26th IEEE International Symposium on Parallel and Distributed Processing, 2012 (IPDPS 2012)*, 2012.

[12] 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 *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin / Heidelberg, 2009, vol. 5704, pp. 851–862.

[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 *Proceedings of PACT'98*, ser. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 88–.

[14] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-gpu and multi-cpu parallelization for interactive physics simulations," in *Euro-Par 2010*. Springer Berlin / Heidelberg, 2010, vol. 6272, pp. 235–246.

[15] 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.

[16] 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.

[17] F. Lementec, T. Gautier, and V. Danjean, "The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming," INRIA, Rapport Technique RT-0418, Dec. 2011. [Online]. Available: http://hal.inria.fr/hal-00648245

[18] NVIDIA, "Nvidia developer zone," January 2012. [Online]. Available: http://developer.nvidia.com