Evaluation of two topology-aware heuristics on level-3 BLAS library for multi-GPU platforms

Thierry Gautier, Joao Vicente Ferreira Lima

Evaluation of two topology-aware heuristics on level-3 BLAS library for multi-GPU platforms

Thierry Gautier*, João V. F. Lima†
*LIP Laboratory, INRIA, CNRS, UCL – Lyon, France
†Graduate Program in Computer Science – Universidade Federal de Santa Maria – Santa Maria, Brazil
thierry.gautier@inrialpes.fr, jvlima@inf.ufsm.br

***Abstract*—Nowadays GPUs have dominated the market considering the computing/power metric and numerous research works have provided Basic Linear Algebra Subprograms implementations accelerated on GPUs. Several software libraries have been developed for exploiting performance of systems with accelerators, but the real performance may be far from the platform peak performance with multiple GPUs. This paper presents two runtime heuristics to gain in performance when task based programs are performed on heterogeneous architecture such as multi-GPU systems. The first is a topology-aware policy to takes into account the heterogeneity of the high speed links that interconnect GPUs. The second is an optimistic heuristic that favor communication between devices. These have been implemented in the XKBLAS library BLAS-3 library. We made experiments on a NVIDIA DGX-1 with up to 8 GPUs V100 on a set of Basic Linear Algebra Subroutines. Experimental results on kernels showed that XKBlas outperformed most implementations including the overhead of creation and scheduling of dynamic tasks.**

***Index Terms*—Multi-GPU architecture, Runtime for task based programs, Topology-aware, BLAS.**

I. Introduction

Dense linear algebra and operations on matrices are fundamental subprograms in scientific applications and deep learning. The design of BLAS [1] library [1] makes the development of high performance applications easy. BLAS enables the cooperation of a good numerical method for solving accurately a domain-specific problem and a highly tuned library implementation for dense linear algebra operations. Several commercial (Intel MKL, Intel oneAPI, AMD AOCL, AMD ROCm, IBM ESSL, NVIDIA cuBLAS, LibSci by Cray/HPE) and open source (OpenBLAS, ATLAS) implementations propose highly tuned algorithms. Hence, BLAS ensures *performance portability* of linear algebra routines and it became a standard building block in HPC despite acceptable criticism about the approach has been formulated [2], [3].

HPC platforms have changed significantly since the creation of BLAS in the 70's. Memory hierarchies in the 80's were captured by the definitions of BLAS level 2 and level 3 with higher arithmetic intensity. Since the arrival of GPUs in HPC about a decade ago, they have continuously demonstrated its performance/energy ratio which makes them unavoidable for extreme scale computing. Nowadays NVIDIA V100 SMX2 has a peak performance of 7.8 TFlops/s in double precision floating point number (DP). The new NVIDIA A100 GPU reaches a peak performance of 9.8 TFlops/s[2]. In comparison the high-end Intel Xeon Gold 6238 Cascade Lake peak DP performance is $2.9TFlops/s$. Hence, offloading computation to single or multi-GPUs has been an active research field [2], [4]–[14]. To exploit the whole available performance of GPUs, programmers must deal with challenging problems concerning the latency of communications between the host and GPU(s), memory limitation, and load balancing in heterogeneous architectures. Nevertheless, accelerating BLAS on multi-GPUs in legacy applications imposes a trade-off between performance and (heavy) code refactoring, such as changing the matrix data layout, and the gain of drop-in replacement libraries.

Obtaining performance on matrix multiplication (GEMM) from a legacy application with LAPACK matrix layout is easy on large matrices because GEMM kernel achieves good occupancy on GPUs. However, a BLAS library assumes that memory used for matrices has been already pinned, and obviously, accounting the pinning time degrades performance. In addition, real applications schedule several BLAS kernels with dependencies. Except XKBlas [15], [16] all experimented libraries with LAPACK matrix layout have synchronous semantics with strong guarantees about the CPU memory coherency after the operations. Data on a GPU may be transferred back and forth after the end of a BLAS routine if a new BLAS is scheduled on the GPU, which is a strong limitation. The lack of support to take into account composition of (BLAS) kernels may result in a significant performance penalty.

The capacity to separate communication and computation for better compositions is not enough to reach high performance on multi-GPU systems. All published results with BLAS-3 subroutines [2], [5], [7]–[14], [17] are based on the exploitation of several levels of parallelism thanks to the decomposition of matrix operations into smaller sub-matrix operations. The resulting computation exhibits a higher degree of parallelism exploited by a runtime system on the different CPU or GPU resources thanks to a scheduling algorithm for heterogeneous processing resources. Overhead due to communications between resources are either reduced thanks to a locality aware scheduler or by trying to overlap data transfers and computations. Except BLASX [14] that organizes its software cache to favor GPU to GPU data transfers between

[1]Basic Linear Algebra Subroutine

[2]And even more: 19.5 on the new IEEE FP64 Tensor core arithmetic.

GPUs having a direct PCIe bus, no libraries address the complex communication topology of highly integrated multi-GPU systems such as the NVIDIA DGX systems, *e.g.* Fig. 1 or the IBM Power9 and NVIDIA V100 nodes of the Summit or Sierra US supercomputers.

This paper presents two topology-aware heuristics to improve performance on multi-GPUs system through the BLAS-3 library XKBlas [15], [16]. XKBlas is based on the XKaapi runtime system [6], [11], [18] for data-flow task programming and scheduling. The main contributions of the paper

1) *A topology-aware device-to-device data transfer.* When a task is waiting for data, the data transfer is initiated from a copy inside the memory of the GPU with a highest performance link.
2) *Optimistic heuristic for device-to-device data transfer.* To favor device-to-device data transfers, we use the following heuristic: we wait for the end of the reception of a copy of the data before forwarding it to the destination GPU.
3) *A performance report* of the 6 main BLAS level 3 subroutines with 7 libraries on a system with 8 GPUs.

We compared the gain of our heuristics in XKBlas on several linear algebra subroutines. Then we compare performances against a wide set of libraries in this domain: BLASX [14], cuBLAS-XT [19], cuBLAS-MG [20], PaRSEC [17], Chameleon [9]/StarPU [5], and Slate [21] over a multi-GPU system NVIDIA DGX-1 with 8 GPUs interconnected through a high speed NVLink network depicted in Fig. 1. XKBlas consistently outperformed them in all cases except on SYR2K (symmetric rank-2k update) and SYRK (symmetric rank-k update) routines when we consider Chameleon tile algorithms. On double-precision floating-point GEMM (*e.g.* DGEMM), XKBlas performs up to $2.84\times$ faster than current NVIDIA cuBLA-XT, $2.52\times$ faster than PARSEC, $1.13\times$ than preliminary cuBLAS-MG, $3\times$ faster than Chameleon (Tile matrix layout) and $5\times$ faster than Slate or Chameleon (with LAPACK matrix layout). The maximum gain in performance arises for matrices less than 40 000, where the communications are still important compared to the computations.

Our previous paper on XKBlas [15] focused on the XKBlas API that made explicit call to make consistent the CPU memory and the gain in composing of kernels. Because of important capacity to overlap data transfers and computation, and the small size of its task based runtime system, we have decided to port our topology-aware communication and optimistic heuristic to favor communication between GPUs having faster NVLink links. This relative small development efforts give us access to a full set of BLAS routines to made comparisons against other BLAS libraries. For moderate size square matrices of dimension about 24 000, we mesure $\approx$54 TFlop/s on DGEMM with 8 NVidia V100 GPU (peak 62.4 TFlop/s), taking into account all the communications between CPUs an GPUs.

The remainder of the paper is organized as follows. Section II analyzes the background and related works. Section III describes XKBlas library and the topology-aware strategies. The experimental results of XKBlas against state-of-art implementations are presented in Section IV. Finally, we conclude the paper in Section V.

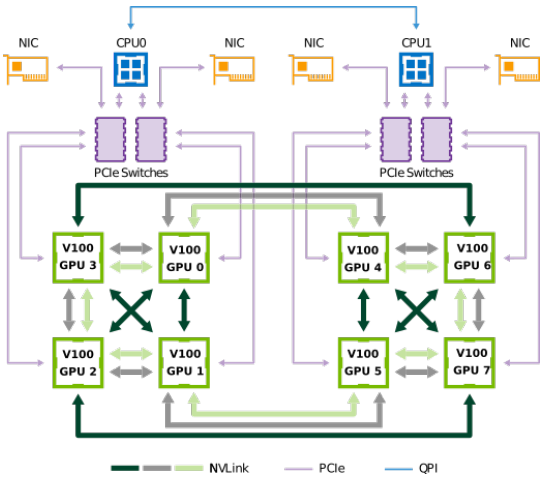

Fig. 1. Hybrid cube-mesh network topology between GPUs and CPUs on a NVIDIA DGX-1 machine.

D\D	0	1	2	3	4	5	6	7
0	744.05	48.37	48.39	96.49	96.45	17.11	17.74	17.97
1	48.38	750.48	96.50	48.38	16.98	96.44	17.32	16.97
2	48.34	96.28	750.48	96.47	17.62	16.93	48.39	17.75
3	96.26	48.34	96.28	750.48	17.58	17.22	17.60	48.39
4	96.46	16.98	17.65	17.53	746.89	48.39	48.40	96.49
5	16.94	96.42	16.88	17.21	48.39	745.47	96.51	48.40
6	17.65	16.90	48.40	17.51	48.34	96.47	750.48	96.47
7	17.80	16.91	17.77	48.39	96.28	48.38	96.28	747.61

xx : local GPU
xx : 2 NVLinks
xx : 1 NVLink
xx : PCIe

Fig. 2. Bandwidth (GB/s) measured between GPUs on a NVIDIA DGX-1 machine.

II. Background and Related Works

A. Multi-GPU BLAS libraries

There are several libraries that provide dense linear algebra algorithms for BLAS an LAPACK routines. Several works [2], [7], [9], [21], [22] assume matrix representation with *tile data layout*. Tile algorithms create tasks that operate on contiguous memory tiles in order to reduce cache penalty and to increase performance. But this representation comes at the price of rigidity in further decomposition of tiles that could not be made without copying or using another matrix representation as in PaRSEC [17]. Furthermore, when porting tile algorithms on multi-GPU, communications of sub matrices to GPU make them contiguous on GPU. In this context, tile representation on the host is less relevant. BLASX [14] and XKBlas [15] support only LAPACK data layout for matrix representation.

Several libraries [7], [9], [12], [14], [19], [21] offer LAPACK subroutines on the (legacy) LAPACK matrix representation. Few of them are designed to allow drop-in replacement such as cuBLAS-XT [19] (thanks to the NVBLAS wrapper) and BLASX [14]. But the latter is a stopped project for which

public source code only containing the general matrix-matrix multiplication GEMM routine.

NVIDIA announced cuBLAS-MG [20] at the end of 2019 as a state-of-the-art matrix-matrix multiplication library in which each matrix can be distributed over multiple devices in a 2D block cyclic strategy.

B. Overlapping communications and computations

The overlap of communications and computations is a common strategy in order to reduce the impact of communication latency between CPU and GPU on slow PCIe bus. For instance, the DGX-1 system has a complex heterogeneous network between processing units (Fig. 1). GPUs are linked together at 0 or 1 hops in DGX-1 high speed NVLink network, with routes at 96GB/s and other at 48GB/s. Moreover, the CPUs communicate with the GPUs through x16 PCIe Gen3 interfaces and each PCIe bus is shared by two GPUs.

One way is to exploit multiple CUDA streams with asynchronous communications with pinned memory as in StarPU [5] and XKaapi [6], [11], then more recently BLASX [14] and PaRSEC [17]. StarPU [5], BLASX and cuBLAS-XT enqueue input operands and kernels into the same stream. Then overlapping comes from the use of several streams.

Another strategy initially proposed in XKaapi [11] is to run each operation type over a separate stream (host-to-device copy, device-to-host copy or kernel execution) with multiple streams for kernel operations in order to let the GPU scheduler execute them concurrently if possible. StarPU, PaRSEC [17], cuBLAS-XT have adopted a similar strategy.

Slate [21] focuses on "exascale" challenge in linear algebra. The current version only relies on the block outer-products implemented on top of batched GEMM [21] as portability layer for accelerator. The asymptotic performance remains good [21] but this design decision is unable to fully exploit all the hardware, especially direct connections between GPUs thanks to the high speed NVLink network. Moreover, the limited bandwidth of PCIe bus between CPUs and GPUs of DGX-1 system drastically limits achievable performance as revealed by the experiments carried out in this paper.

C. Multi-GPU software cache

A distributed caching mechanism is a well-known approach in order to hold copies of host data into disjoint address spaces such as GPUs. Several variations of a modified MOSI protocol have been proposed [5], [11], [14], [17] with impact on performance not really comparable due to the number of experimental variables involved (problem size, hardware, GPU memory size, GPU type and count). The notable protocols are BLASX that proposes a two-level cache mechanism to improve locality of data access to favor GPU-to-GPU communication, and XKaapi where the eviction strategy prioritizes read-only data first.

III. New topology-aware heuristics for XKBlas

XKBlas [15] library design was based on two important decisions regarding linear algebra algorithm and matrix layout. Its algorithms are based on asynchronous tiled algorithms from Chameleon [9] or PLASMA [22] that allows to describe computations using a dependent task model supported by XKaapi [18]. We only keep support for LAPACK matrix layout in XKBlas that is more robust to dynamic and recursive sub-partitions, differently from Chameleon or PLASMA. The XKBlas algorithms come with the tile version of the corresponding algorithms with the following differences:

- Tile representation is replaced by sub-matrix representation using LAPACK data layout (see below);
- Instructions to copy back a matrix block to the host have been suppressed because they introduce extra data transfer between device and host;
- LAPACK matrix data layout is required by legacy applications, thus the tile representation API has been discarded;
- Extended LAPACK API with asynchronous semantics is the only native XKBlas API for BLAS.

Therefore, the numerical algorithms of XKBlas have the same behavior of those from PLASMA or Chameleon.

A. Overview of data management

Our extensions in XKBlas only concern the XKaapi runtime system which is briefly recalled for the sake of clarity.

In XKBlas, the XKaapi runtime maintains information about the data distribution of matrices. Then the internal scheduling algorithm uses an owner-computes rule heuristic [11] to map tasks on resources. It controls the distributed execution of tasks, the schedule of communication while trying to overlap latency by kernel execution.

On CPU, each tile is represented as a memory region starting at address A with its description given by the tuple $(m, n, ld, wordsize)$. m, n are matrix dimensions, ld is the leading dimension and $wordsize$ the size in byte of an element of the matrix. The tuple is called a *memory view* of the matrix. The sub-matrices keep the same representation after a matrix decomposition.

Thanks to the `cudaMemcpy2D` set of primitives, a (sub)matrix A in XKBlas can be transferred between GPUs and CPUs. Once copied (to GPU), the memory view of (sub)matrix A is $(m, n, m, wordsize)$, *i.e.* the leading dimension always becomes the row dimension. The matrix has been compacted to a *dense tile form*. Since most computations are performed by the GPU, the compact tile form is used as effective parameters for GPU kernels.

All copies of tiles are tracked by the XKaapi software cache [11], [18]. Locality information was used by the several XKaapi data aware schedulers [11] in order to map a task on the GPU close to its data. When a GPU cache becomes full, the eviction strategy prioritizes read-only data first.

B. Topology-aware strategy on hierarchical memory systems

Figure 1 illustrates the hybrid cube-mesh interconnection network topology between all the GPUs on a NVIDIA DGX-1. Each GPU is interconnected by a link ranked in three groups: 2 NVLinks, 1 NVLink, and PCIe. The bandwidths between pairs

TABLE I
MAIN CHARACTERISTICS OF DGX-1 MULTI-GPU SYSTEM.

Name	CPU	GPU
Gemini	2 Xeon(R) E5-2698 v4 2.2GHz	8 NVIDIA Tesla V100-SXM2, 32GB CUDA-10.1

of GPUs are reported in Fig. 2. Some GPUs are connected together with 2 NVLinks (green background color in Fig. 2). This interconnects them with up to 100GB/s of bidirectional bandwidth (measured close to 96GB/s). Two GPUs can also be connected with each other by 1 NVLink (orange background color), with up to 50GB/s bidirectional bandwidth. Otherwise, they are interconnected using PCIe buses.

The main question in this topology is how to better exploit the interconnection network topology between all the GPUs. In XKBlas the runtime system (XKaapi) initiates data transfers of the tile from one of the resources handling a valid copy once the scheduler has decided to map a task. A matrix tile is frequently replicated over several GPUs at runtime, which is typically the case for tiles of the input matrix operands.

We modified the XKaapi runtime to better select the source GPU involved into a data transfer in order to favor a high speed NVLINK interconnect if available. For all possible GPUs that handle a valid copy of a tile, we prioritize the selection of a source following the decreasing order of performance link in respect to the destination GPU.

This information about performance link is reported by `nvidia-smi` tools on Linux by with specific function for CUDA SDK. So we extend XKBlas to store the performance of link during the library initialization by using the relative value of performance links as returned by a call to `cuDeviceGetP2PAttribute`.

Our topology-aware communication heuristic shares the same goal as the 2-level hierarchical cache mechanism in BLASX [14] but it is able to consider complex interconnect topology between GPUs such as in the NVIDIA DGX-1 (Fig 1). We have not evaluated our heuristic on IBM POWER 9 node with NVIDIA V100 GPUs, which is also possible to returns performance link information of the 50GB/s NVLink links between a CPU and a GPU.

C. Optimistic heuristic for device-to-device data transfers

With topology-aware heuristic, communications may favor high speed links between GPUs if a copy of the data to transfer is already present in one or several GPUs. But in most cases the host main memory is the source of most data transfers. This is especially true for all input operands of BLAS routines before the tiles are copied to GPUs. In this scenario and without a "clairvoyant" algorithm to distribute initial data, the advantage of having high-speed links between GPUs is significantly reduced.

Our heuristic opportunistically tries to take advantage of ongoing copies of tiles to GPUs at runtime instead of a distribution or a redistribution of tiles before the invocation of a BLAS routine, It considers that if a valid data is under transfer to a GPU but not yet fully received, it may be better to wait the end of this transfer before forwarding the data to the target GPU. This strategy aims to improve overall performance by a better usage of the high speed links between GPUs,

We have implemented this optimistic heuristic in XKBlas by extending the metadata stored by the multi-GPU XKaapi software cache with a state indicating that a data is *under transfer* to a specific GPU. When selecting a source GPU for making a copy, our implementation first searches if the data is already valid on a GPUs using the topology-aware heuristic. Otherwise, it returns the best GPU with data under transfer instead of falling back to the host as data source.

The gain on NVIDIA DGX-1 is important because performance of PCIe buses between CPU and GPU are one of the main limiting factor to reach peak performance. Moreover, the heuristic avoids duplicate tile transfers from main memory to GPUs to reduce data traffic on PCIe bus. On Summit or Sierra supercomputer nodes, where GPUs have high speed NVLink interconnect between CPUs, it would be reasonable to assert that the gain will not be significant.

IV. EXPERIMENTAL RESULTS

Our experiments target a NVIDIA DGX-1 multi-GPU system described in Table I. It has 512GB of main memory and two Intel Xeon E5-2698 v4 processors with 20 cores each (40 cores total). The interconnect between CPU and GPU is PCIe (Gen3) and GPUs are interconnected together with NVLink-2. The DGX-1 is running GNU/Linux distribution with kernel 4.19.146.

We used the BLAS libraries: public version of BLASX [14]; StarPU 1.3.5 and Chameleon 1.0.0; cuBLASG-MG [20] version number 0.1.0; Slate [21] Git hash version `451f5ff`; cubBLAS-XT from CUDA Toolkit version 10.2; DPLASMA Git hash version `2cefa568`. XKBlas is a non commercial library available at https://gitlab.inria.fr/xkblas. The git hash of the library used to conduct these experiments is `27325d238`.

A. Methodology

We have make several comparisons in order to analyse our multi-GPU library compared to state-of-the-art BLAS implementations. Our experimental results compared performance on two scenarios: *data-on-host* and *data-on-device*. The goal of *data-on-host* set of experiments is to compare performance in the context where operands and results are mainly on the host memory. On the other hand, *data-on-device* scenario initially distributes matrices using a 2D-block cyclic distribution. The target applications are those where a drop-in replacement library for BLAS can accelerate computation on multi-GPU systems.

In Section IV-D we included the necessary time to transfer operands as well as the time to get the result back on the host. Our scenario has the advantage to give an end-to-end performance for practical use cases. Almost all the tested libraries have a set of benchmarks or testing codes to measure performance in terms of GFlop/s with the *data-on-host* constraints. The only exception concerns Chameleon

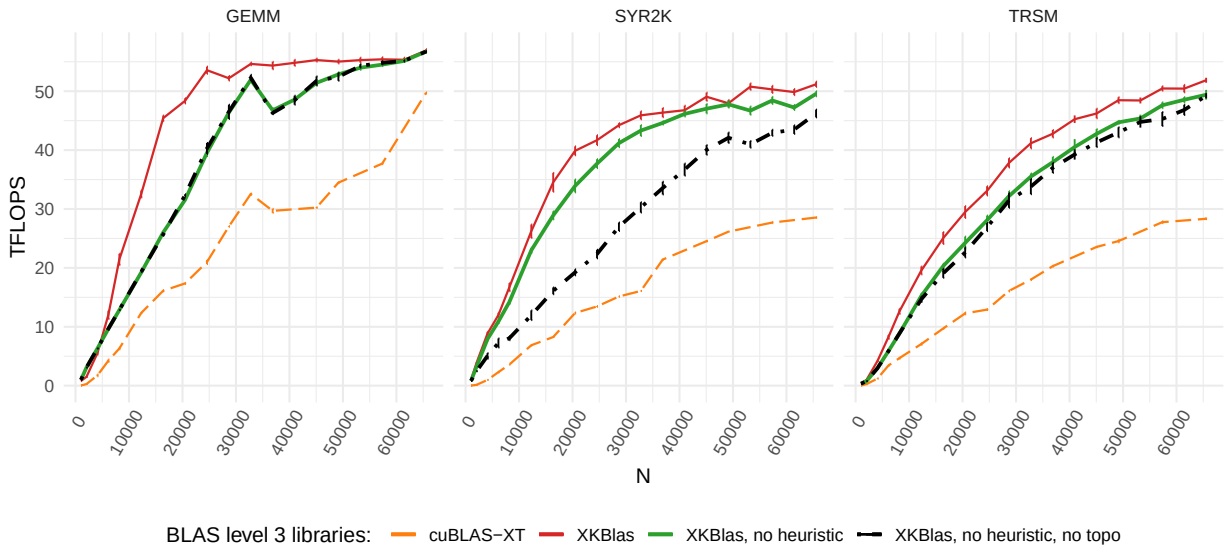

Fig. 3. Performance with device-to-device and topology-aware heuristics disabled on FP64 DP GEMM, SYR2K and TRSM. All runs followed data-on-host methodology. "XKBlas" means for XKBlas with both device-to-device and tologogy-aware heuristics enabled. "XKBlas no heuristic" report performance of "XKBlas" with only device-to-device heuristic (section IV-B) disabled. "XKBlas no heuristic, no topo" illustrates the level of performance without our optimistic heuristic to favor device-to-device transfers neither the topology-aware heuristic. cuBLAS-XT results are kept as reference.

with tile algorithms for which default testing codes report performance without guarantee that results are on the host before stopping the time counter[3]. We added the necessary synchronization instructions in our experiments to wait valid result on host before stopping the time counter.

Moreover we have changed the default configuration of workers on Chameleon/StarPU runtime [5] experiments. We increased the number of concurrent kernels per GPU[4] from 1 (default) to 2 since more workers degraded performance. We selected the DMDAS StarPU scheduling algorithm that seems to be well suited for linear algebra algorithms according to StarPU papers [5], [9]. Finally, we have made several runs before measuring performance in order to let StarPU build a performance model of each tasks involved during the computation.

Each measure reported in this Section was a mean of 8 runs with a warm-up run discarded. Except if it is explicitly described, we only report results with a tile size that maximizes performance among the experimented tile sizes $(1024, 2048, 4096)$ for each matrix dimension and library. We extended these sizes for cuBLAS-XT and Slate with up to 8192 and 16384 in order to maximize their performance. With this block size selection we were able to capture the best performance from the tested BLAS subroutines on all the range of matrix dimension. Block size tuning is outside of the scope of this paper.

Nevertheless, the time to page lock the memory was ignored in all experiments. We assume that applications have the capacity to amortize this cost by using the same memory multiple times. Thus, all the tested libraries register data memory to CUDA driver in order to accelerate transfers of (sub)matrices between CPUs and GPUs before measuring execution time.

[3]A simple Gantt chart from CUDA nvprof tool, with marker added using nvToolsExt, relevant that D2H data transfers arrive after the end of the computation event reported by the benchmark.

[4]By setting the environment variable STARPU_NWORKER_PER_CUDA.

B. Impact of our optimistic device-to-device data transfers

Section III-B and III-C present how XKBlas improves performance to better exploit of the connection topology between GPUs with two strategies: favoring communication between GPUs with the highest performance rank; avoiding CPU-to-GPU communications by optimistically deciding to wait for the arrival of a data replica on a GPU instead of copying from CPU.

Figure 3 reports the performance of GEMM, SYR2K and TRSM (triangular system solving) routines on 8 GPUs. We used CUBLAS-XT as a reference library, XKBlas with the two heuristics enabled and XKBlas with one or two heuristics disabled. *XKBlas* stands for the full extended XKBlas library with the two heuristics presented in previous sections. *XKBlas, no heuristic* is the XKBlas where optimistic heuristic to favor device-to-device communication is disabled. *XKBlas, no heuristic, no topo* is the configured as *XKBlas, no heuristic* but in addition we disabled the selection of GPUi with respect to the high performance group first as described in the topological-aware heuristic section III-B.

Table II resumes the maximum gain or loss in performance when a feature is disabled. We also added the performance gain with data-on-device case. On the highly regular GEMM algorithm, the performance degradation, with the optimistic heuristic disabled was up to 43% over default XKBlas baseline. Moreover, GEMM routine was not sensible to the

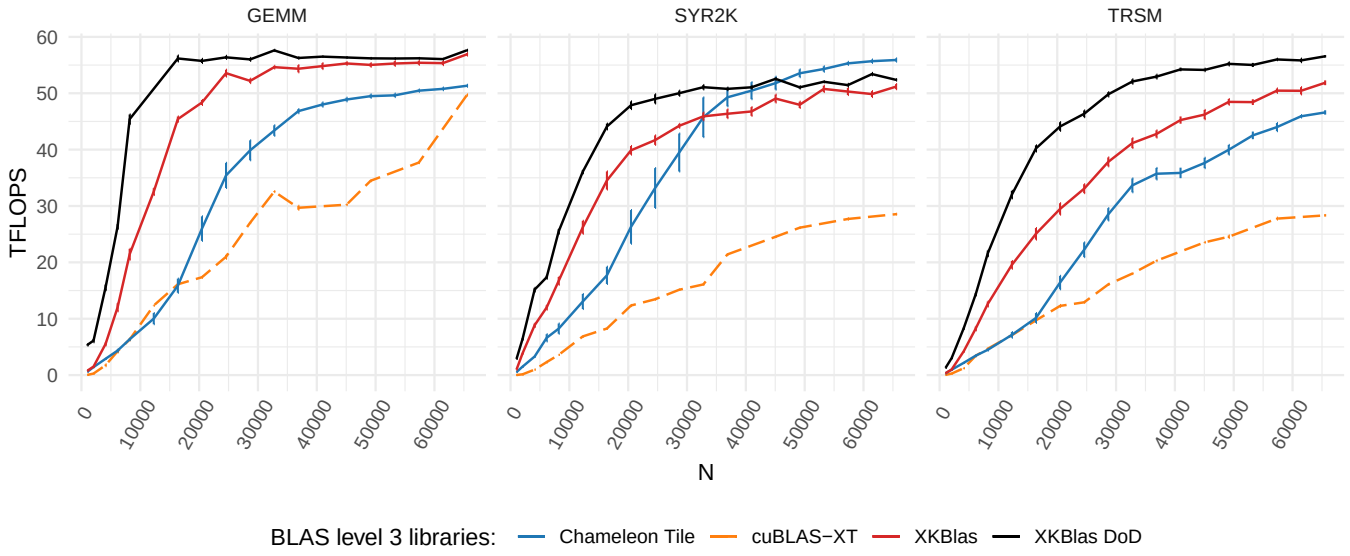

Fig. 4. Performance with data-on-device (black curve) on FP64 DP GEMM, SYR2K and TRSM. XKBlas runs with data-on-host, while XKBlas DoD runs with all data on the GPU devices. XKBlas, Chameleon Tile and cuBLAS-XT results presented in Figure 5 were kept as references.

TABLE II
MAXIMUM LOSS/GAIN OF PERFORMANCE FOR DIFFERENT XKBLAS VARIANTS WITH RESPECT TO THE BASELINE XKBLAS ON MATRIX DIMENSION GREATER THAN OR EQUAL TO 16384.

Kernel	data-on-device	no heuristic	no heuristic, no topo
DGEMM	+111.7%	−43.5%	−43%
DSYR2K	+71.1%	−19.4%	−53.5%
DTRSM	+52.6%	−29.6%	−29.3%

topology-aware strategy with the highest performance rank. TRSM had less performance degradation up to 30% with the two features disabled.

On SYRK2, performance degraded 20% if the heuristic is disabled. However, disabling the topology-aware feature decreased the performance up to 54% over fully-enabled XKBlas library. The origin of this issue is due to an imbalance in the communication between GPUs introduced by the XKBlas scheduling algorithm. It seems that some GPUs require more time to send or receive data than the others over an 1 NVLink path. In that case, if the runtime does not try to systematically get the peer GPU with high performance rank, then it impacts performance significantly.

C. Performance with data-on-device

Two dimensional block-cyclic distribution scheme is a standard data mapping to resources when porting dense linear algebra routines on distributed memory architecture. This is the data layout in the ScaLAPACK library. Viewing the GPUs as a distributed memory architecture, we are able to distribute matrices following a block-cyclic distribution. This is the purpose of the routine `xkblas_distribute_2Dblock_cyclic_async` in XKBlas.

Figure 4 reports performance of DGEMM, DSYR2K and DTRSM with XKBlas "data-on-device" (DoD) where the matrices were initially distributed using a 2D-block cyclic distribution. All the matrices can be stored on the 8 x 32GB of the V100 GPU memory on these experiments. We assume a (4,2)-grid of GPUs and we select the following parameters to fix the distribution: the tile size used by the algorithms is $\lceil \frac{N}{4ngpus} \rceil$ to ensure enough parallel slackness, and the block-cyclic block sizes of the distribution is set to (1,1), *i.e.* two adjacent blocks of matrices are mapped to different GPUs. We keep performance results reported in Figure 5 of XKBlas and CUBLAS-XT as reference that includes time to transfer operands and results.

The overall results were consistent with our hypothesis. When data are on device, all transfers occur between GPUs at the speed of NVLink interconnect without transfers between the CPUs. XKBlas attained performance results of ≈50TFlops even for square matrices of about 10 000. The gap between data-on-device and data-on-host experiments of XKBlas decreases when increasing the matrix dimension N. The arithmetic intensity is in $O(N)$ for the three kernels, so it implies the ratio communication/computation asymptotically tends to 0. The arithmetic operations become a bottleneck for any communication amount from the CPU or from the GPU.

Chameleon Tile outperforms XKBlas DoD on SYR2K, even if input and output data are stored in the GPU memories. Despite the gain avoiding data transfer with CPUs, when the matrix size becomes bigger than 45 000 (20,000 in previous data on host experiments, see Figure 5) Chameleon Tile SYR2K obtains higher levels of performance. We will investigate furthermore this work imbalance problem in Section IV-E.

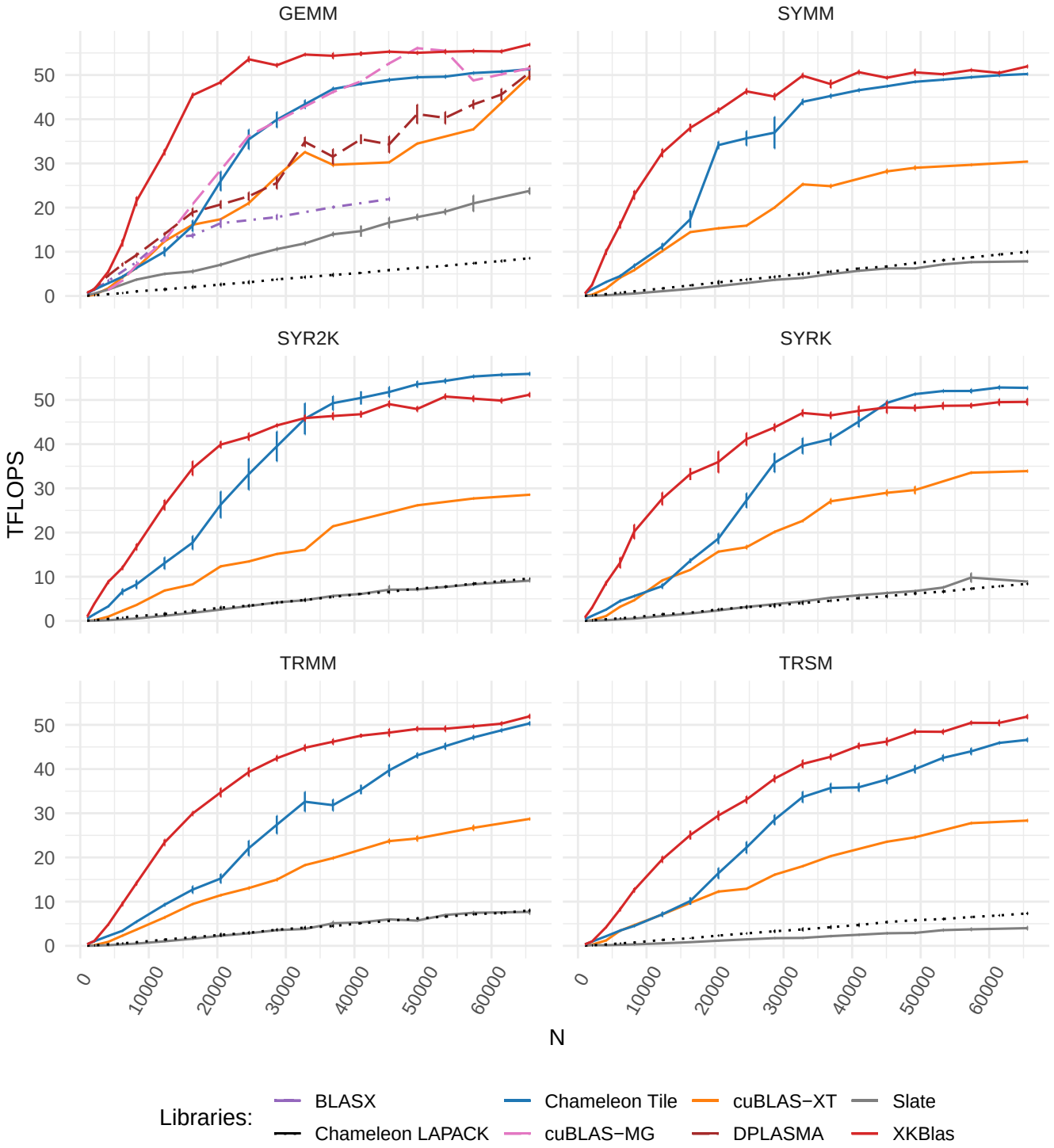

Fig. 5. Performance of libraries on DGX-1 with 8 GPUs for 6 BLAS subroutines. Chameleon LAPACK (black dotted line type) reports performance when input and output matrices are stored using LAPACK layout, while Chameleon Tile uses tile representation. BLASX DGEMM reports memory allocation errors when running with bigger matrices than 45 000. Error bars display the 95% confident interval of each point.

D. Comparison with state-of-the-art multi-GPU libraries

Figure 5 shows performance results in TFlop/s of DGEMM, DSYMM, DSYR2K, DSYRK, DTRMM and DTRSM increasing the matrix dimension over the 8 GPUs of our NVIDIA DGX-1. Time to transfer inputs to GPUs and the output results to the host are included. Chameleon experiments are made

with two versions of the BLAS subroutines: the "Chameleon Tile" version allocates matrices using internal tile data layout. The "Chameleon LAPACK" allocates operands and result using the LAPACK data layout, then the implementation of BLAS subroutines automatically converts matrices to/from tile data layout before and after computing the result. Some routines have missing points because the tested libraries do not include all routines: cuBLAS-MG only implements GEMM in its current version; BLASX public code only contains GEMM routines even if the authors reported performance with almost all BLAS L3 kernels [14]; DPLASMA implementation exploits GPUs with GEMM only.

XKBlas outperformed all other libraries for almost all routines with peak performance of 56.9 TFlop/s on FP64 DP GEMM ($\approx 91.2\%$ from theoretical peak of GPUs, i.e. 8×7.8 TFlop/s for SMX2 V100). On GEMM subroutine, with small matrix dimension of about 10,000, XKBlas was more than three times faster than the other best libraries (cuBLAS-MG, DPLASMA or Chameleon Tile). The less performant library was Chameleon using LAPACK data layout due to the penalty, on the host, to convert operands and result to/from tile matrix representation [9].

Slate did not scale on our experiments with 8 GPUs of DGX-1 system. Its design targets supercomputers and focuses on distributed memory computers interconnected by high speed network. Slate organizes portability to accelerators through the block outer-product pattern common to several algorithms [21] which can be based on batched GEMM. However, the implementation of the batched GEMM was unable to exploit the capability of 8 GPUs to directly exchange data through the high speed NVLink network. Consequently, all data transfers between CPUs and GPUs pass through the 4 PCIe 16x Gen3 buses at 16GB/s each, which limits performance on DGX-1 system.

The low performance of Chameleon LAPACK was due to the software design for handling matrix stored in LAPACK layout. With the same layout XKBlas was able to achieve a much higher level of performance because it does not need copies between matrix data layout during computation.

Chameleon Tile attained significant performance results on bigger matrix sizes. The differences between XKBlas versus Chameleon Tile decreases with the dimension of matrices because the relative weight of communication reduces with respect to arithmetics. Because the BLAS algorithms in XKBlas and Chameleon are the same, the performance differences between XKBlas (with our heuristics) and Chameleon (LAPACK or Tile) were only due to: unnecessary copies in case of Chameleon LAPACK; the runtime systems Chameleon/StarPU or XKBlas/XKaapi; our heuristics.

Furthermore, Chameleon outperformed XKBlas on 2 subroutines (SYR2K and SYRK) over the 6 routines when the matrix dimension were greater than $20\,000$ (resp. $45\,000$). One reason may be that Chameleon/StarPU scheduler DMDAS [5] is more efficient on these problems SYR2K/SYRK to avoid communication and load imbalance. The XKBlas scheduler relies on the XKaapi work stealing, with locality heuristic [11], that seems to generate work and communication load imbalance.

The performance improvement made by our heuristics presented in section IV-B demonstrate a significant performance gain of XKblas against the other libraries, especially for smaller matrices. This high reactivity comes from the dynamic nature of the heuristic that reduces communication volume between the host and the GPUs.

Among all the experimented libraries, the only available libraries that offers the 9 standard BLAS subroutines[5] supporting the LAPACK matrix data layout are cuBLAS-XT, Chameleon LAPACK and XKBlas. Moreover, cubBLAS-XT with NVBLAS and XKBlas provide dynamic libraries to trap Fortran and C calls from BLAS subroutines and offload them to GPUs. Regarding this scenario, as a drop-in replacement library, XKBlas had up to 300% more performance than cuBLAS-XT and 500% more performance than Chameleon LAPACK version.

E. Execution trace analysis

We analysed the execution traces of GEMM and SYR2K in order to understand performance results obtained. We did not include Chameleon/LAPACK since its performance was lower in our performance experiments (Figure 5). Each trace only included GPU operations and API calls were discarded. We used the `nvprof` utility to collect the traces.

Figure 6 shows cumulative execution times (left) and normalized ratio over total execution (right) of GEMM for a matrix dimension of 32768. It clearly demonstrates that XKBlas was more efficient on data transfers than other libraries with $\approx 25.4\%$ of total execution. Other libraries spent more time in data transfers during execution. Chameleon Tiled had slightly better efficiency wih $\approx 41.2\%$ on data transfers.

Figure 7 illustrates the execution trace of SYR2K for each GPU with matrix dimension of 49152. It seems that Chameleon/StarPU scheduler was able to balance workload over GPUs efficiently. On the other hand, XKBlas showed load imbalance on either communication and execution over GPUs. We constate that CUBLAS-XT had a similar behavior on GEMM than on SYR2K where it spent most of execution time in data transfers.

F. Composition of BLAS calls

In XKBlas all kernels are invoked through asynchronous function calls such as `xkblas_dgemm_async`. If a thread executes a sequence of calls to XKBlas subroutines, then each call inherits from previous data distribution among GPU stored in the XKaapi cache when work on tiles have been dispatched among the GPUs. This fact sketches our the simple proposal for composition of BLAS kernels in XKBlas. Composition is noted to be one of the key point for reaching high performance in sparse direct solver [23], [24] such MUMPS [25].

Figure 8 shows performance results of a composition benchmark with routines TRSM + GEMM using Chameleon Tile

[5]The 6 routines of the figure 5 with addition of Hermitian version of SYMM, SYR2K and SYRK

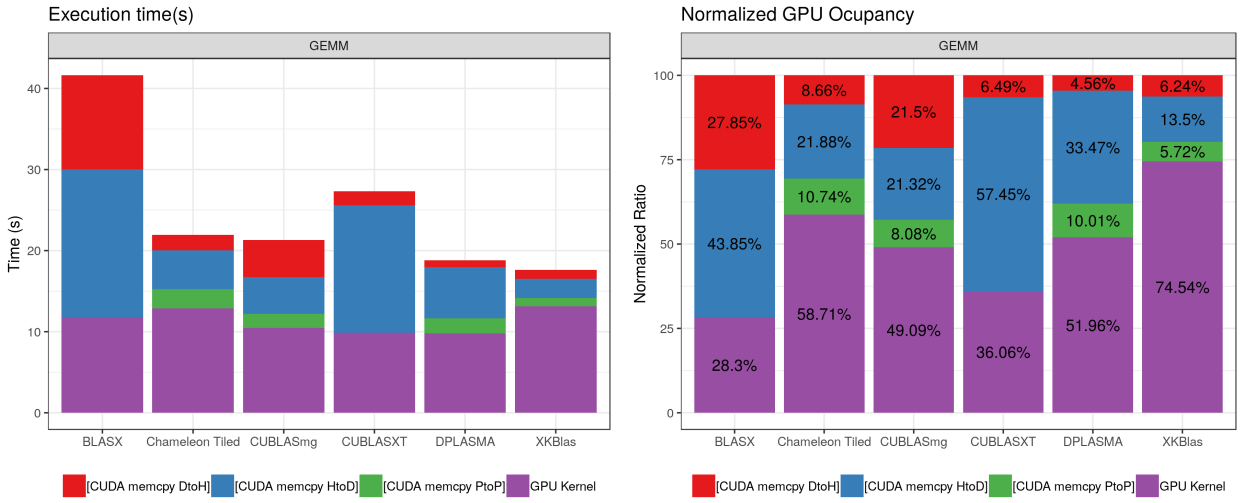

Fig. 6. Detailed execution of GEMM FP64 with cumulative execution time (left) and normalized ratio over total execution (right) on 8 GPUs of the DGX-1 System.

and XKBlas libraries. Each routine call was asynchronous over the input matrices. XKBlas reached significant performance of 56.6 TFlop/s that was close to the peak performance of GEMM routine (56.9 TFlop/s). It seems that XKBlas routines were able to compose both routine calls and did not add synchronization points between routine calls. Chameleon attained 36.6 TFlop/s which was under its GEMM peak of 51.3 TFlop/s.

Figure 9 illustrates a Gantt chart of one run of Chameleon and XKBlas composition benchmark. Clearly, XKBlas was able to compose both routine calls while Chameleon had synchronization gaps.

Moreover, if a first task writes a tile while a second task reads the same tile, they would correctly be dependent as needed thanks to the asynchronous semantics of the XKaapi runtime. Thus any sequence of user function calls generating tasks would allow to define point-to-point synchronization between tasks among different function calls. The absence of synchronous semantics that force synchronization between

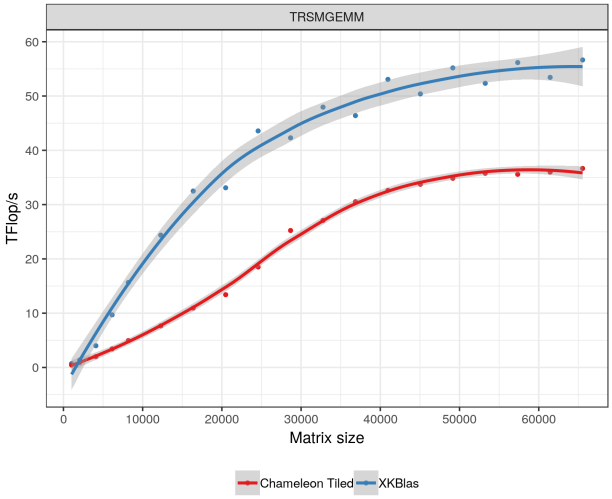

Fig. 8. Chameleon Tile and XKBlas performance of the composition of TRSM+GEMM FP64 with matrix dimension 32768 and block size 2048 over 8 GPUs. The graphic shows each point as a mean of 8 runs and a LOESS smoothing line with 95% confidence interval around the smooth.

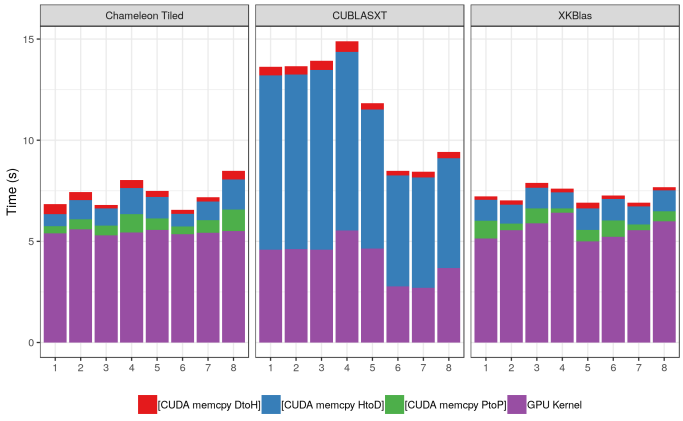

Fig. 7. Execution trace of SYR2K FP64 with matrix dimension 49152 by GPU on DGX-1 System.

calls permits to keep busy the GPUs. Other BLAS libraries, such as Chameleon, MAGMA, DPLASMA expose asynchronous tile API for asynchronous function calls in order to favor composition but they impose the CPU memory to be consistent on a synchronization point. cuBLAS-XT has synchronous invocation of BLAS kernel with data transferred back and forth after each call to BLAS.

In place of systematic and implicit data transfer associated with completion of kernel, XKBlas adopts a lazy approach where the user should describe which matrix or subparts of matrix has to be made coherent on the CPU. This is the key point for efficient composition of BLAS subroutines to avoid unnecessary data transfers.

The sequence of one BLAS kernel, that generates computational tasks, followed by a call to

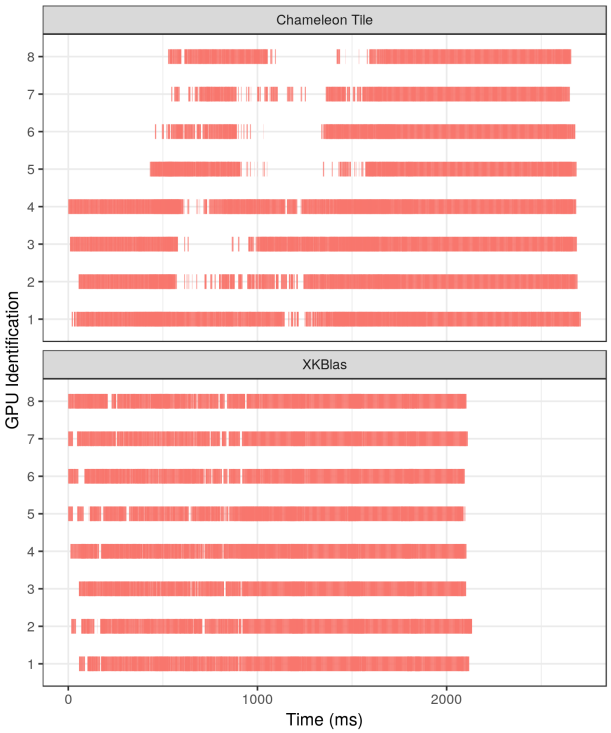

Fig. 9. Chameleon Tile and XKBlas Gantt chart of the composition of TRSM+GEMM FP64 with matrix dimension 32768 and block size 2048 over 8 GPUs.

`xkblas_memory_coherent_async` is a typical composition that allows to mix together computation with data transfers back to the CPU. At runtime, a task making CPU memory coherent is executed as soon as tile results are computed because of the dependency. In Chameleon/StarPU, to initiate as soon as possible such transfer, each tiled algorithm unrolls a data-flow graph with annotations to flush back when a computed tile is done. Nevertheless, the user in XKBlas has an explicit control over the usage of communication links between CPU and the GPUs which are a shared and limited resources.

V. Conclusion

In this paper we presented two heuristics to increase performance of the XKBlas library. The heuristics make use of the topological information of the high speed NVLink interconnect with the dynamic information of the location of data replicas. They were added inside XKBlas and only impact the data management part of the underlaying XKaapi library. Experimental results have demonstrated significant performance gains for XKBlas.

The main contributions are a topology-aware strategy to transfer data from the GPU with the most performant link, and an optimistic heuristic that gives priority to device-to-device instead of host-to-device transfers. In addition, we present a complete comparison of several software stacks that target BLAS routines. Our experimental results showed that XKBlas scales on multi-GPU systems and outperformed other libraries in all the cases except for the SYRK and SYR2K computations where Chameleon was able to outperform XKBlas if the dimension of matrices becomes larger. The problem was due to the XKaapi scheduling algorithm that introduced work imbalance in the computation and communication between GPUs on these algorithms.

Our heuristics make no assumption about the scheduling algorithm. They are interfaced between the scheduler that takes decision to execute a task on a resource and the runtime part in charge of data input transfers. We believe that our heuristics are not specific to XKBlas and would be portable across other libraries based on dependent task execution.

The portability of our performance results on other architectures is the next step. We were very interested in validating the performance results on other highly integrated multi-GPU systems such as the IBM Power9 with NVIDIA V100.

XKBlas has the potential to be widely used by the scientific community on legacy applications relying on dense linear algebra operations. It supports natively LAPACK matrix layout without sacrifice on performances and it is able to obtain high performances even on small problem instances. For this two properties XKBlas is one of the supported multi-GPU libraries in the MUMPS [25], [26] software, a sparse linear solver. Preliminary experimental results confirm the performance of XKBlas presented in this article.

Acknowledgment

This work has been partially supported by the projects: “GREEN-CLOUD: Computação em Cloud com Computação Sustentavel” (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

Experiments on DGX-1 were carried out using the Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

References

[1] “An updated set of basic linear algebra subprograms (blas),” *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002.

[2] F. D. Igual, E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, R. A. van de Geijn, and F. G. V. Zee, “The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations,” *Journal of Parallel and Distributed Computing*, vol. 72, no. 9, pp. 1134 – 1143, 2012, accelerators for High-Performance Computing.

[3] F. G. Van Zee and R. A. van de Geijn, “Blis: A framework for rapidly instantiating blas functionality,” *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015.

[4] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, “Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks,” in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP ’08. New York, NY, USA: ACM, 2008, pp. 123–132.

[5] 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.

[6] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, “Multi-gpu and multi-cpu parallelization for interactive physics simulations,” in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, ser. Euro-Par’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 235–246.

[7] 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, Jun. 2010.

[8] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, April 19-23 2010, pp. 1–8.

[9] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2. [Online]. Available: https://hal.inria.fr/inria-00547847

[10] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 557–568.

[11] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1299–1308.

[12] A. Charara, H. Ltaief, and D. Keyes, "Kblas: An optimized library for dense matrix-vector and matrix-matrix operations on gpu accelerators," 2016. [Online]. Available: https://ecrc.kaust.edu.sa/Pages/Res-kblas.aspx

[13] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra, "Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 491–500.

[14] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, "Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 20:1–20:11.

[15] T. Gautier and J. F. Lima, "Xkblas: a high performance implementation of blas-3 kernels on multi-gpu server," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2020, pp. 1–8.

[16] "Xkblas," 2019. [Online]. Available: https://gitlab.inria.fr/xkblas/versions

[17] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 156–165.

[18] J. V. Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard, "Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures," *Parallel Computing*, vol. 44, pp. 37 – 52, 2015.

[19] NVIDIA, "cublasxt," 2016. [Online]. Available: developer.nvidia.com/cublasxt

[20] ——, "cublas-mg, early access program," 2019. [Online]. Available: https://developer.nvidia.com/CUDAMathLibraryEA

[21] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "Slate: Design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

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

[23] A. Gupta, N. Gimelshein, S. Koric, and S. Rennich, "Effective minimally-invasive gpu acceleration of distributed sparse matrix factorization," in *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 672–683.

[24] P. Sao, R. Vuduc, and X. S. Li, "A distributed cpu-gpu sparse direct solver," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 487–498.

[25] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIAM J. Matrix Anal. Appl.*, vol. 23, no. 1, pp. 15–41, Jan. 2001.

[26] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, "Performance and scalability of the block low-rank multifrontal factorization on multicore architectures," *ACM Transactions on Mathematical Software*, 2018.