HAL
open science

XKBlas: a High Performance Implementation of BLAS-3 Kernels on Multi-GPU Server

Thierry Gautier, Joao Vicente Ferreira Lima

To cite this version:

Thierry Gautier, Joao Vicente Ferreira Lima. XKBlas: a High Performance Implementation of BLAS-3 Kernels on Multi-GPU Server. PDP 2020 - 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Mar 2020, Västerås, Sweden. pp.1-8, 10.1109/PDP50117.2020.00008 . hal-03121583

HAL Id: hal-03121583

https://inria.hal.science/hal-03121583v1

Submitted on 28 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

XKBlas: a High Performance Implementation of BLAS-3 Kernels on Multi-GPU Server

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*—In the last ten years, 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. This paper presents XKBlas that aims to improve performance of BLAS-3 kernels on multi-GPU systems. At low level, we model computation as a set of tasks accessing data on different resources. At high level, the API design favors non-blocking calls as uniform concept to overlap latency, even by fine grain computation. Unit benchmark of BLAS-3 kernels showed that XKBlas outperformed most implementations including the overhead of dynamic task's creation and scheduling. XKBlas outperformed BLAS implementations such as cuBLAS-XT, PaRSEC, BLASX and Chameleon/StarPU.**

***Index Terms*—Multi-GPU, BLAS, Task Parallelism.**

I. Introduction

Dense linear algebra and operations on matrices are fundamental subprograms in scientific applications and deep learning. The design of BLAS library [1] makes easy the development of high performance applications. BLAS makes possible 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. For instance LAPACK relies on BLAS for performance portability. Several commercial (Intel MKL, AMD ACML, IBM ESSL) and open source (OpenBlas, ATLAS) implementations propose highly tuned algorithms. Hence, BLAS ensures *performance portability* of linear algebra routines. And, despite if acceptable criticism about the approach were formulated [2], [3], it became a standard building block in HPC.

Since the 70's, when BLAS was defined, its architecture varies a lot. Memory hierarchies in the 80's were captured by the definitions of BLAS level 2 and level 3 with higher arithmetic intensity. With the apparition of GPU in HPC a decade ago, GPU has continuously demonstrated its performance/energy ratio which makes it unavoidable for extreme scale computing. Nowadays NVIDIA V100 SMX2 has peak performance of 7.8 TFlops/s in double precision floating point number (DP). In comparison the high end 8180 Xeon platinum Skylake peak DP performance ranges between 1.5 TFlops/s and 2 TFlops/s depending of the real frequency due to turbo boost mode. Moreover, memory bandwidth has higher rate on GPU than CPU which is required for low arithmetic intensity applications. Hence, offloading computation to single or multi-GPUs has been an active research field [2], [4]–[14].

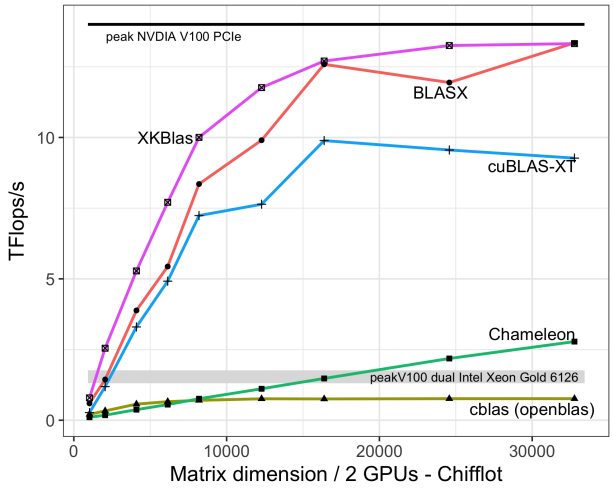

Fig. 1. Performance DGEMM on 2 V100-PCIe NVidia with several libraries that provide BLAS routines on LAPACK matrix layout.

To better exploit all performance capacity, programmers must deal with challenging problems concerning latency of communication between host and GPU(s), memory limitation of GPU and, finally, load balancing in heterogeneous architectures. Nevertheless, accelerating BLAS on multi-GPUs in legacy applications imposes a tradeoff between performance with heavy code refactoring and overhead of drop-in replacement libraries.

Let us illustrate the level of performance of various multi-GPU libraries providing API with same LAPACK matrix layout as BLAS (or CBLAS) library. Figure 1 reports the GFlops/s for GEMM matrix-matrix multiplication with the following libraries: up-to-date version[1] of the library Chameleon [15] on top of StarPU [5], BLASX [14], multi-core CBLAS from OpenBLAS-0.3.3 and cuBLAS-XT from CUDA-9.0. The GEMM, pure CPU, has performance of about 762 GFlops/s (43% of the CPU peak, 5% of the GPU peak) [2]. Chameleon with StarPU reaches at most 2.8 TFlops/s

[1]Git hash g1f14c6b25.

[2]DP GEMM. Dual socket server (Chifflot V) is an Intel Gold 6126 has peak performance between 1.3 TFlops/s and 1.76 TFlops/s with attached 2 NVIDIA V100 PCIe 16GB GPUs at 7TFlops/s DP per GPU.

(about 20% of GPU peak) due to mandatory conversion from LAPACK matrix layout to internal tile matrix representation. The cuBLAS-XT with 9.8 TFlops/s (70% of the peak) is able to process arbitrary large matrix but with overhead due to communication and grain size penalties (see section V-B in [14]). BLASX showed more than 13 TFlops/s (95% of the peak) and, to the knowledge to the author, was the fastest library with LAPACK matrix layout representation. It is possible to call BLASX routines directly in legacy applications for the small set of routines implemented. XKBlas, presented in this paper, attained similar performance (95% of the GPU peak) compared to BLASX but outperforms it on smaller matrix dimensions.

Obtaining performance on GEMM from a legacy application with LAPACK matrix layout is easy on large matrices because GPUs have good occupancy. But (1) it assumes the fact that memory used for matrices should have been already pinned, and accounting the pinning time in performance degrades it. (2) Real applications schedule several BLAS kernels with dependencies. Except XKBlas, proposed in this paper, all libraries of Fig. 1 with LAPACK matrix layout have synchronous semantics with strong guarantees about the CPU memory coherency after the operations. This point is a strong limiting factor: data on GPU after the end of BLAS routine may be transferred forth and back if new BLAS is scheduled. The lack of support to take into account composition of (BLAS) kernels is a performance penalty.

This paper presents XKBlas, a high-performance BLAS library to exploit multiple GPUs, based on the XKaapi runtime system [6], [11], [16]. The main contributions of XKBlas on multi-GPU BLAS for HPC legacy applications are two simple and powerful features:

1) *Composition of kernels.* XKBlas allows to describe a sequence of any BLAS kernels to be scheduled on resources. Each BLAS kernel is decomposed into tasks with data flow dependencies and the runtime manages dependencies between tasks of different kernels. At runtime, the data flow graph between computational tasks is unfold and scheduled thanks to the XKaapi runtime system.
2) *Explicit coherency operation.* XKBlas lets the user to express data transfers explicitly between resources. Memory of GPUs are viewed as cache, and data transfers are a meaning of forcing update to the main host memory.

We base our solution on re-using fine grain tasks from XKaapi that allows to integrate complex operations with dependencies among other operations as a XKaapi task. Task management and data flow dependencies have low overhead [11]. It is one of the reasons that XKBlas outperformed BLASX and cuBLAS-XT in Fig.1, which do not implement such dependent task model.

We evaluated XKBlas on two multi-GPU systems over cuBLAS-XT, PaRSEC and Chameleon/StarPU. XKBlas consistently outperformed them on all systems (4 NVIDIA P100 SMX2, 2 NVIDIA V100 PCIe, and 4 NVIDIA V100 SMX2). In comparison with NVIDIA cuBLAS-XT, XKBlas demonstrated better scalability with up to on average 188% performance gain and 450% less communication volume during all experiments.

The remainder of the paper is organized as follows. Section II analyzes the background and related works. Section III reviews the proposed XKBlas library and the XKaapi runtime system. We detail design and implementations of XKBlas and the interaction between XKBlas and XKaapi in Section IV. The experimental results of XKBlas against existing state-of-art implementations are presented in Section V. Finally, we conclude the paper in Section VI.

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], [17] assume matrix representation with *tile data layout*. Tile algorithms create tasks that operate on contiguous memory tile 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 copy or other matrix representation as in PaRSEC [18]. Furthermore, when porting tile algorithms on multi-GPU, the communications of sub matrices to GPU make them contiguous on GPU. In this context, tile representation on the host is irrelevant.

Several libraries [7], [9], [12], [14], [19] offer LAPACK subroutines on the (legacy) LAPACK matrix representation. Few of them are designed to be drop-in replacement (cuBLAS-XT [19] thanks to the NVBLAS wrapper and BLASX [14] but public source code only contains general matrix-matrix multiplication BLAS).

B. Overlapping of communication by computation

The overlap of communication and computation is a common strategy in order to reduce the impact of communication latency between CPU and GPU on slow PCIe bus. One strategy 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 [18]. StarPU, BLASX and cuBLAS-XT enqueue input operands and kernel into a same stream and overlapping comes from the use of several streams.

Another strategy from XKaapi [11] runs each operation type over a separate stream (host-to-device copy, device-to-host copy or kernel execution) with multiple streams for kernel operation in order to let the GPU scheduler execute them concurrently if possible. PaRSEC [18] has adopted the same strategy.

C. Multi-GPU software cache

A distributed caching mechanism is well-known approach in order to hold copies of host data into disjoint address spaces such as GPUs. Several variations of modified MOSI protocol have been proposed [5], [11], [14], [18] with impact on performance not really comparable due to the number of

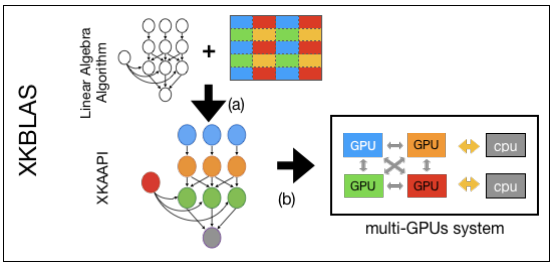

Fig. 2. Overview of XKBlas/XKaapi steps.

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 prioritize eviction of read only data first. We modified XKaapi to better select the source device involved into a data transfer in order to favor the closest GPU-to-GPU transfer if possible, and potentially through a high speed NVLINK interconnect.

III. XKBLAS OVERVIEW

Figure 2 overviews XKBlas steps from a dependency graph to execution on a multi-GPU system. XKBlas [20] links together two distinct parts: a set of tiled BLAS-3 linear algebra algorithms [9], [17], and the XKaapi runtime system for scheduling tasks with data dependencies over multi-GPUs [11]. The tiled version of dense linear algebra code [17] unrolls a data flow graph with a degree of parallelism that depends on matrix distributions (upper part of Fig. 2).

Then the scheduling algorithm uses the owner-computes rule heuristic to map a task on resources (Fig. 2(a)): a task is mapped to the resource owning its input matrix block. If no owner exists then the runtime makes a random choice among resources. Then the XKaapi runtime system distributes (Fig. 2(b)) the task and data among resources. It controls the distributed execution of tasks, schedules communication and tries to overlap latency with kernel execution.

A. Matrix representation and distribution

The BLAS-3 tiled algorithms of XKBlas are based on PLASMA [17] and Chameleon [9], which rely on tasks and data dependencies to unfold parallelism. Although both use tile matrix representation, XKBlas has the standard LAPACK matrix data layout (with colmajor storage) which allows to perform dynamic and recursive sub-partitions.

In the column major data layout from LAPACK, the memory region of a matrix starting at address A is entirely described 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 matrix. The tuple $(m, n, ld, wordsize)$ is called the memory view of the matrix. This representation does not change with decomposition operation: sub matrices have the same representation.

A matrix A in XKBlas is a pair of $(address, memory\ view)$ which can be copied or transferred between GPUs. Once copied, the memory view of (sub)matrix A is $(m, n, m, wordsize)$, *i.e.* the leading dimension always becomes the row dimension. We said the matrix has been compacted to a *tile form*.

The runtime maintains the *preferred master resource* of each matrix or submatrix in order to improve data mapping. For any matrix block A_{ij} that appears during computation, `preferred_map`(A_{ij}) returns the location of the resource that stores a valid copy of A_ij. Initially the returned value is `ANY` meaning that any resource is the preferred master resource. Once a task updating A_{ij} is completed on resource R_k, the runtime updates the `preferred_map` for A_{ij} to R_k.

The user may decide to set explicitly the preferred master resources for each block of a matrix or by calling `xkblas_map_1Dblock_cyclic` or `xkblas_map_2Dblock_cyclic` functions that only set the mapping information for all matrix blocks in 1D or 2D distribution over resources. XKBlas also proposes functions `xkblas_distribute_2Dblock_cyclic_async` or `xkblas_distribute_1Dblock_cyclic_async` to update the mapping information and migrate the blocks to the corresponding resources until the memory resources are full.

The mapping information is used by the scheduler to implement the owner-computes rule heuristic. Block cyclic and owner-computes rule are basic scheduling algorithms for mapping task to GPUs. cuBLAS-XT also relies on them.

B. Tiled algorithms

XKBlas currently implements the following algorithms: GEMM, SYMM, TRSM, TRMM, SYRK, SYR2K, HEMM, HERK and HER2K for classical precision: S (single precision floating point number), D (double precision), C (complex, single precision) and Z (complex, double precision).

XKBlas algorithms comes with the tile version of corresponding algorithms in Chameleon [9] with the following modification:

- Tile representation are replaced by sub matrix representation using LAPACK data layout;
- 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 so the tile representation API has been discarded;
- Extended LAPACK API with asynchronous semantics is the only internal XKBlas API for BLAS.

We extend matrix-matrix multiplication by providing a XGEMMT BLAS level 3 routine, similar to Intel MKL, that computes the general matrix product $A \times B$ but only updates the upper or lower triangular part of the result matrix. This is useful when the result is known to be symmetric, such as some computations in MUMPS [21] or WSMP [22]. The algorithm only creates tasks to update the triangular part. On the diagonal, due to the lack of similar routines, we call cuBLAS XGEMM.

C. Composition of BLAS routines

Integration of GPU in the sparse linear solver WSMP [22] goes beyond BLAS acceleration. Authors have proposed in ACCEL_WSMP API to capture some compositions of BLAS calls during their sparse Cholesky factorisation in order to provide optimized version, *e.g.* with DTRSMSYRK which composes one DTRSM followed by one DSYRK. In [23] this is the Schur complement update that is subject to optimization by on composition, aggregation to increase arithmetic intensity of GEMM and pipeline.

In XKBlas any sequence of BLAS calls could be composed in order to let the runtime better schedule tasks and manage communication. Moreover, XKBlas proposes a set of API methods to force cache invalidation which is coherent with the rest of the API. We also developed a multi-GPUs GEMMT that updates triangular part of matrix-matrix product which is extensively used in numerical solver such as MUMPS [21] when input problem is symmetric.

IV. XKBLAS INTERNALS

This section describes the implementation details of XKBlas such as its runtime system XKaapi, API calls for cache asynchronous execution, and composition of BLAS kernels.

A. Inherited task model and scheduling

XKaapi is a *macro data flow* runtime that creates, at runtime, tasks and themselves may spawn child tasks [6], [11]. Each task must describe the access mode to the memory for each of its formal parameter (read, write, etc).

XKaapi implements a non-preemptive (work stealing) scheduler, i.e. once started a task runs to completion. In order to avoid data hazards, there are constraints on the modes of access between the formal parameters and the effective parameters [24]. The "sequential" semantics comes from Athapascan-1 [24]: a value read by a task is produced by the last previous task with write access following the sequential order of execution.

At runtime, XKaapi computes the true dependencies between tasks and builds a data flow graph that represents the future of the computation. The runtime is in charge of transfers of input data (read, read-write access mode) and makes data allocation before starting tasks. XKaapi manages memory resources as a distributed shared memory.

In XKaapi each GPU has its own queue of ready tasks. A thread, bound to a CPU core closed to the GPU it manages, tries to execute at most w-tasks, where w is the window size (generally two in XKaapi) of concurrency on GPUs or it pulls for completion of asynchronous communication. The task execution relies on K-Stream [11] to handle communication and kernel execution.

When a task finishes, it may activate successor tasks that are pushed to the queue of a GPU that stored the output tile matrix. Among the available scheduling algorithms in XKaapi [16], we have only kept the owner compute rule strategy in XKBlas to map a task on the GPU that stores one of its output. If none of the task outputs is already mapped to GPUs, the library push the task to a random selected GPU.

B. Distributed Cache and Coherency operator

To execute a task, the scheduler creates a local copy of each parameter and transfers input data from a valid copy located on a GPU or main memory. The runtime keeps track of address spaces, data copies and their state. The decision about the source address space is always to select the closest with respect to the GPU topology or at random if several resources have a valid copy at the same distance.

After a synchronisation point, all tasks previously spawned, and their descendants, are completed. Nevertheless, host memory remains non-coherent with previous results of computation: new tasks could be spawned and the runtime will transfer data among resource, but the user could not directly read the host memory to access the result. He must explicitly ask the runtime to make coherent the host memory with respect to all the updated copies on the GPUs. XKblas has a function to update the upper or lower (or both) triangular part of a matrix on the host called `xkblas_memory_coherent_async`. Its implementation simply spawns a task for each block of matrix $A_{i,j}$ concerned with triangular part. The task takes a read access mode on the block and initiates the asynchronous copy back to the host. The operation is asynchronous and completion is guaranteed after the next synchronisation point. Note that a complementary function allows to request only one tile to be coherent.

C. Asynchronous API design

All XKBlas operators have a non blocking semantic: the caller should explicitly issue a synchronisation call to ensure that previous asynchronous operations have completed. XKBlas low-level routines define non-blocking function calls for computation, data annotation and memory coherency.

XKBlas high-level methods have CBLAS and Fortran BLAS entry point with blocking semantics. Typically the XGEMM function is implemented on top of non-blocking API using the following sketches:

```
void dgemm( char * transa, char * transb,
  int * m, int * n, int * k,
  double* alpha, double* A, int * lda,
  double * B, int * ldb,
  double * beta, double * C, int * ldc)
{
  // asynchronous tile dgemm
  xkblas_dgemm_async(transa, transb,
    *m,*n,*k,alpha,A,*lda,B,*ldb,beta,C,*ldc);
  // requests to make coherent full C matrix
  xkblas_memory_coherent_async(
    UPPER|LOWER,
    *m,*n,C,*ldc,sizeof(double)
  );
  xkblas_sync();
}
```

The `xkblas_dgemm_async` call returns immediately after pushing a request to make coherent full C matrix (upper and lower parts). Visibility of the request is ensured after the

next synchronisation point `xkblas_sync` that waits until completion of all previous spawned tasks and their children.

D. CUDA memory register call

To be able to overlap transfers between device and host with kernel execution, transferred data must have been registered to CUDA using `cudaHostRegister`. The purpose is to page lock the memory in order to let DMA to transfer data. Nevertheless, the cost of this function is significant especially when it is difficult to reuse already pinned memory. Moreover, we have noticed that the total time spent in this `cudaHostRegister` call is multiplied by 2 when using 4 GPUs instead of 1 GPU.

The XKBlas API provides an asynchronous function called

```
xkblas_register_memory_async
```

to overlap the memory pinning by other operations (computation, matrix assembly, ...). The host registration is done by a specific thread using *HostRegisterPortable* flag and the result is visible to all CUDA contexts as ensured by the CUDA specification: "memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation". This function introduces an implicit synchronisation with future computational kernels which are launched only when all requests to page-lock data accessed by task are completed.

E. Composition of BLAS kernels

In XKBlas all kernels are invoked through asynchronous function calls such as `xkblas_dgemm_async`. The feature is 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 [22], [23] such as MUMPS [21].

Moreover, if two tasks generated by the same thread and the first one writes while the second reads the same tile, they would create a true dependency 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 synchronisation between tasks among different function calls. The absence of synchronous semantics that force synchronization between calls permits to keep busy the GPUs. Other BLAS library runtime (Chameleon/StarPU, Magma, PaRSEC) expose asynchronous tile API for asynchronous function calls in order to favor composition but they impose implicit coherency on synchronisation point. cuBLAS-XT has synchronous invocation of BLAS kernel.

Instead of a set of implicit data transfers after kernel completion, 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 `xkblas_memory_coherent_async` is a typical composition that allows to overlap computation with data transfers back to the CPU. At runtime, because of the data dependency, the transfer to CPU memory is executed as soon as tile results are computed. In Chameleon/StarPU, to initiate as soon as possible such transfer of result to CPU, each tiled algorithm unrolls a data flow graph with annotation to flush back when the computed tile is done. The drawback of this approach is extra data transfers between host and GPUs a priori to the real composition of tiled algorithms. Nevertheless, in XKBlas the user has an explicit control over the usage of communication links between CPU and the GPUs which are a shared and limited resources.

TABLE I
MAIN CHARACTERISTICS OF MULTI GPU SYSTEMS.

Name	CPU	GPU
Chifflot	2 Xeon(R) Gold 6126 2.6GHz	2 NVIDIA Tesla V100-PCIE, 32GB CUDA-9.0
Blaise	2 Xeon E5-2699 v4 2.2GHz	4 NVIDIA Tesla P100-SXM2, 16GB CUDA-10.0

V. EXPERIMENTAL RESULTS

Experiments have been made on two multi-GPU systems described in Table I. The interconnect is PCIe (Gen3) on Chifflot, and NVLINK-1 on Blaise. We used the public version of BLASX [14], version 1.2.6 of StarPU and Chameleon with up-to-date git hash g1f14c6b25.

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

Figure 3 reports average performance results of DGEMM, DSYMM, DSYRK, DSYR2K, DTRMM and DTRSM increasing the matrix dimension. We setup benchmarks over Blaise (Table I) using double precision routines. Matrices are initialized with random numbers. We evaluated XKBlas performance against cuBLAS-XT and BLASX that have LAPACK matrix data layout. Note that even if the authors of BLASX reported performance with almost all BLAS L3 kernels in [14], the public available code only contains GEMM routines.

Each point was a mean of 8 runs. For each matrix dimension and runtime system, we only report the performance corresponding to the tile size that maximizes it among the experimented tile sizes $(1024, 2048, 4096)$. Time to transfer data to GPUs and results to the host was included. We excluded the time to page lock the memory because applications should have the capacity to amortize its cost when using the same memory multiple times.

XKBlas outperformed other libraries for all configurations. Note that the performance of XKBlas using 1 GPU was better than cuBLAS-XT with 2 GPUs. Performance of XKBlas over 2 GPUs was similar or better than the performance of cuBLAS-XT with 4 GPUs.

B. Composition of BLAS TRSM+GEMM

We built a benchmark composing TRSM and GEMM BLAS kernels as it appears on the MUMPS application [21]. In [22] the authors report such fundamental

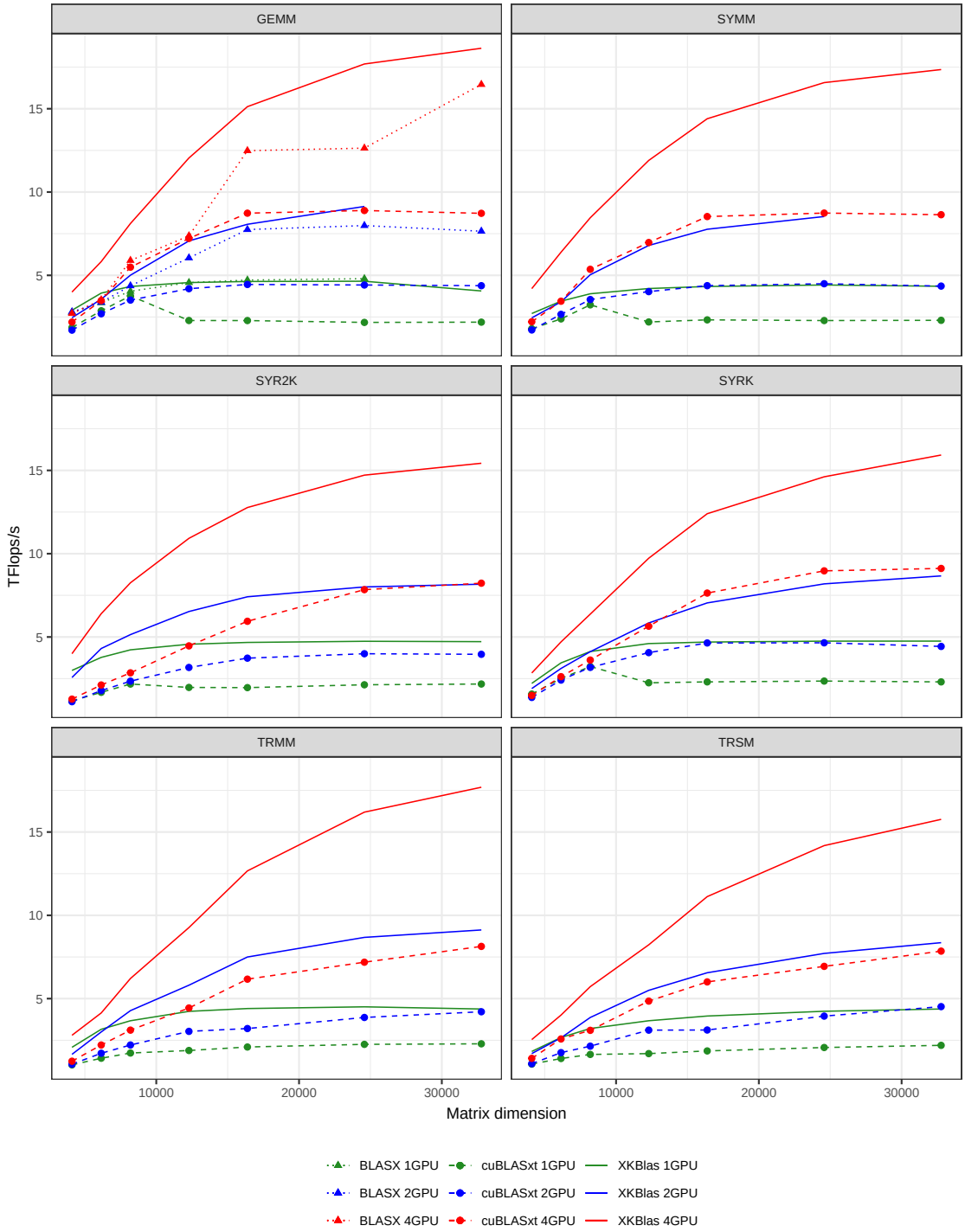

Fig. 3. Performance of BLAS L3 kernels on 4 P100-SXM2 NVIDIA GPUs (Blaise) for several BLAS double precision routines with LAPACK matrix layout. Memory was already pinned before timings.

composition in WSMP application. The XKBlas version was a sequence of calls: `xkblas_dtrsm_async` followed by `xkblas_dgemm_async`. The Chameleon/StarPU version was based on asynchronous API for TRSM and GEMM. The cuBLAS-XT relied on synchronous calls to `cublasXtDtrsm` followed by `cublasXtDgemm`. It is not possible to experiment with BLASX because the public version of the routines does not include multi-GPUs version of TRSM. For all programs, all data initially reside on the host memory and we force results back to the host. In asynchronous XKBlas it was made by calling `xkblas_memory_coherent_async`.

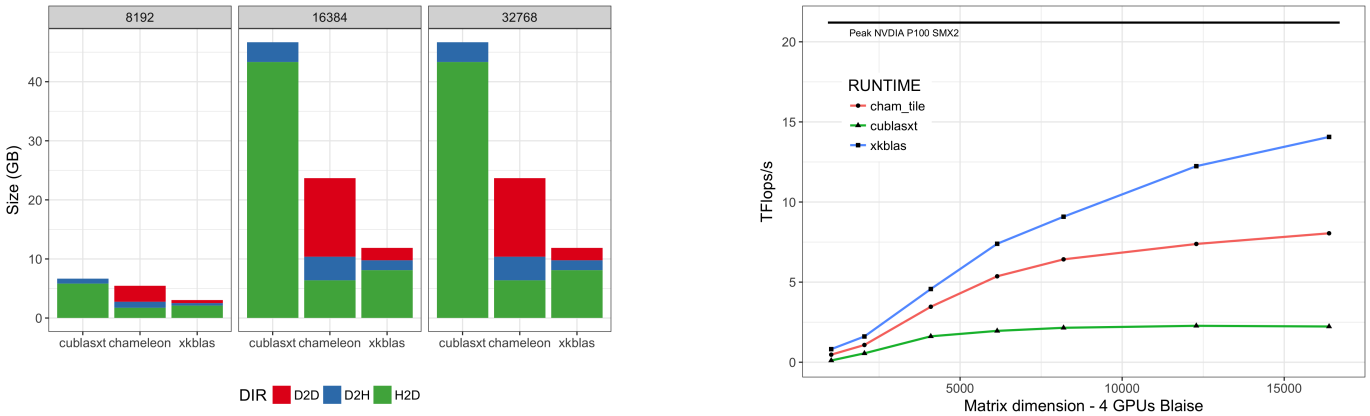

Fig. 4. Left: Cumulative size of data transfers on 4 GPUs during the composition of TRSM + GEMM between host and GPUs on Blaise and 3 size of matrices. Runs with cuBLAS-XT, Chameleon/StarPU (tile version) and XKBlas. Right: corresponding performance for varying matrix dimension on for GPUs.

Figure 4 reports performance and cumulative size in GByte of data transferred from host to GPUs (green, label H2D), GPUs to host (blue, label D2H) and GPU to GPU (red, label D2D) on three set of matrices and on four GPUs. First, cuBLAS-XT showed the highest communication volume. Moreover, it never uses the fast device to device NVIDIA NVLINK communication link (label D2D), which impacted its performance.

XKBlas had the lowest communication volume compared to others. As presented in Section IV-E, XKBlas allows to reduce the volume of communication between GPUs to host (blue D2H part in the bar chart) by decoupling completion of kernel to communication of results. The programmer is responsible for calling the necessary function to make the host memory coherent.

We note that on bigger dimensions ($\geq 10k$) XKBlas outperformed Chameleon on this composition because of the higher communication footprint of Chameleon, which was about 2 times greater than XKBlas. Chameleon performed better on smaller matrices.

C. Performance of Tile versus LAPACK layout

Figure 5 reports experiments with runtime systems that have LAPACK matrix layout representation. It has been argued that tile matrix representation, where each tile is contiguous in memory, is more friendly to cache and TLB [9], [17]. PaRSEC [18] has hybrid matrix representation: at the coarse level, matrix is stored with tile matrix representation. Each coarse tile may be decomposed into fine tiles with LAPACK matrix layout.

In XKBlas, a GPU always holds a tile as contiguous memory region. Data transfer between GPU and host relies on `cuMemcpy2DAsync` that allows to change online from/to LAPACK matrix layout to/from contiguous memory region. A relevant question would be: What is the real gain of tile representation when most of the dense linear algebra computations are performed on GPUs on contiguous memory region?

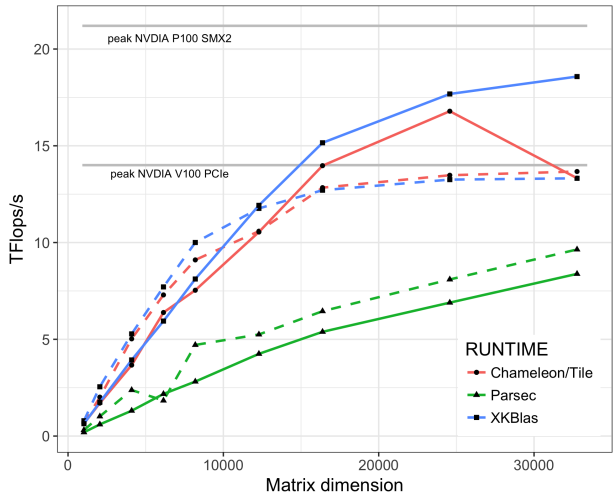

Fig. 5. DGEMM comparison of XKBlas over Chameleon/StarPU and PaRSEC with tile matrix representation. Solid lines show performance on Blaise with 4 GPUs; dashed lines on Chifflot with 2 GPUs.

We partially give an answer with Figure 5 which reports experimental results of DGEMM with tile matrix representation of Chameleon [15] and PaRSEC [18]. In all configurations, the time encompass data transfer of the operands and transfer of the result matrix to the host. Memory was already pinned before timings. On Chifflot (V100 dashed lines) there was no notable difference of using LAPACK matrix layout of XKBlas versus the tile matrix representation of Chameleon. On Blaise (P100, solid lines) XKBlas outperformed Chameleon and its tile representation. We suspect that most of the performance gap was due to StarPU under Chameleon rather than the tile representation. At matrix size of 32768, Chameleon/StarPU has systematically reported a bad performance without a clear explanation. PaRSEC was the less performant runtime system and it seems to suffer from high overhead in management of large number of small tiles.

We argue that forcing tile matrix representation in complex runtime systems for dense linear algebra on multi-GPUs is counterproductive on addressing the problem of accelerating

legacy application. Tile representation, which aims to promote strong code modifications, seems not competitive with libraries such as XKBlas or even BLASX.

VI. Conclusion

In this paper we presented XKBlas, a high-performance BLAS-3 library to exploit multi-GPUs based on the XKaapi runtime system. The main contributions of XKBlas are based on two powerful concepts: asynchronous function calls to compose BLAS kernels; and explicit operator to make coherent CPU that allows to move less data during a composition of BLAS. Scheduling is based on owner compute rule with auxiliary operators to give hints to the scheduler. Moreover, our experimental results showed that XKBlas outperformed BLASX and cuBLAS-XT in almost all configurations or it reaches equivalent performance results.

XKBlas has the potential to be widely used by the scientific community on legacy applications relying on dense linear algebra operations. This paper focus the API and performance of GEMM kernel but the library has complete set of BLAS level 3 kernel. XKBlas currently develops heuristics for a better use of huge memory capacity of multi-GPUs in order to increase the bandwidth and reduce latency in GPU-to-GPU communication.

Acknowledge

This work has been partially supported by the projects: UFSM/FATEC through project number 041250-9.07.0025 (100548); "GREEN-CLOUD: Computação em Cloud com Computação Sustentavel" (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

Research leading to these results has in part been carried out on an UFRGS/Brazil server that was supported by Petrobras project 2016/00133-9.

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] "Chameleon library," 2016. [Online]. Available: https://gitlab.inria.fr/solverstack/chameleon

[16] 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.

[17] 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.

[18] 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.

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

[20] "Omitted for blind review," 2019.

[21] 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.

[22] 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.

[23] 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.

[24] 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 the 1998 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 88–.