Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over PaRSEC

Thomas Herault*, Yves Robert*, George Bosilca* and Jack Dongarra*
* Innovative Computing Laboratory (ICL), University of Tennessee Knoxville, USA
† ENS Lyon, France

***Abstract*—This paper introduces a generic and flexible matrix-matrix multiplication algorithm $C = A \times B$ for state-of-the-art computing platforms. Typically, these platforms are distributed-memory machines whose nodes are equipped with several accelerators. To the best of our knowledge, SLATE [9] is the only library that provides a publicly available implementation on such platforms, and it is currently limited to problem instances where the C matrix can entirely fit in the memory of the GPU accelerators. Our algorithm relies on the classical tile-based outer-product algorithm, but enhances it with several control dependencies to increase data re-use and to optimize communication flow from/to the accelerators within each node. The algorithm is written with the PaRSEC runtime system, which allows for a fast and generic implementation, while achieving close-to-peak performance.**

***Index Terms*—Linear Algebra, Accelerator architectures, Runtime environment**

I. Introduction

As of today, the Summit and Sierra systems [16] are the fastest machines on the TOP500 list [23]. Both systems are distributed-memory platforms where each node is equipped with several high performance NVidia accelerators. For instance Summit nodes include 6 NVIDIA V100 GPUs, interconnected at the node level by multiple NVLinks. The forthcoming Frontier exascale system [16] is announced with four GPUs per node. On Summit, more than 97% of the overall compute performance is on the GPU side. The trend is the same for all state-of-the-art platforms equipped with multi-GPU accelerated nodes: these machines draw most of their computing power out of the accelerators; hence, it is crucial, for any efficient and scalable algorithm, to be able to extract the most performance out of the accelerators to achieve global efficiency. Several on-going projects aim at designing dense linear algebra kernels for these platforms, let alone to provide TOP500 performance and ranking.

Thus, it is critical that one of the most basic operations in dense linear algebra, the matrix-matrix multiplication, has an efficient implementation, whatever the size of the input matrices, on such architectures. To the best of our knowledge, the only publicly available library for dense linear algebra kernels on multi-GPU accelerated distributed memory platforms is SLATE [14], [9]. The current implementation only supports a limited number of operations in a multiple-accelerator setting and has size limitations: for instance the matrix product $C = AB$ prototype is limited to problem instances where the entire C matrix can reside in the memory of the GPU accelerators. On Summit with 6 GPU with 16 gigabytes of memory each, each node can store a double precision floating-point submatrix C (with 8-byte coefficients) of size $N \times N$, where $N \approx 40,000$ (leaving a quarter of the memory for A and B elements).

The main contribution of this work is the design of a generic and flexible matrix-matrix multiplication algorithm $C = A \times B$ for multi-GPU accelerated distributed-memory platforms, for matrices unrestricted by the size of the GPU memory. Our algorithm relies on the classical tile-based outer-product algorithm, but enhances it with several control dependences to increase data re-use and optimize communication flow from/to the accelerators within each node. The algorithm is written within the PaRSEC runtime system, which allows for a fast and generic implementation portable across a variety of architectures, while achieving a sustained performance close to the practical peak of the machine.

The rest of the paper is organized as follows. Section II overviews the main design principles of our algorithm. An analytical count of the number of inter-node and node-accelerator communications is given in Section III. Then Section IV discusses the main details of the prototype implementation, which is publicly available [4]. In Section V, we report preliminary performance results. Section VI briefly discusses related work. Finally, Section VII is devoted to concluding remarks and directions for future work.

II. Design principles

In this section, we outline the general layout of our matrix multiplication algorithm, which obeys simple design principles, and whose architecture is inspired by out-of-core implementations [22], [18], [13]. Table I shows the key notations.

We partition the original matrices into square tiles, which we distribute among the participating processes. A coarse grain view of the platform is a 2 dimensional grid of computing nodes, for which the standard 2D-cyclic layout of tiles is enforced. Let $A(M,K)$, $B(K,N)$, and $C(N,N)$ be the three matrices, regularly tiled into square tiles of size t^2, and assigned with a 2D-cyclic distribution of tiles onto a grid of processors of size $p \times q$. For simplicity, assume that t divides M, K and N and let $M_t = M/t$, $K_t = K/t$ and $N_t = N/t$ be the number of tiles in each dimension. We consider a processor grid of size $p \times q$, where p divides M_t and q divide N_t. The

TABLE I: Key Notations

Notation	Explanation
M, K, N	size of input matrices $A(M,K)$, $B(K,N)$, $C(M,N)$
t	tile size (tiles are square)
M_t, K_t, N_t	matrix sizes expressed in tiles
$p \times q$	size of processor grid
G	number of accelerators per node
$b \times c$	size of C blocks
d	depth of chunk
(x,y,z)	index of chunk, $0 \le x < X$, $0 \le y < Y$, $0 \le z < Z$
$X = \frac{M_t}{bp}$	number of C blocks across rows
$Y = \frac{N_t}{cq}$	number of C blocks across columns
$Z = \frac{K_t}{d}$	number of chunks per C block
ℓ	value of lookahead in terms of chunks

standard outer product algorithm [1], [24], [6], [5] goes as follows.

Let (u,v) denote the position of node number $qu+v$ on the grid, with $0 \le u < p$ and $0 \le v < q$. Node (u,v) initially hosts all the tiles $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$ whose indices satisfy to $i = u$ mod p and $j = v$ mod q, and is in charge of computing all these $C_{i,j}$ tiles. At step k of the algorithm (iteration k of the outer loop), tiles $A_{i,k}$ are broadcast horizontally: there are p parallel broadcasts, initiated by each node on column $k_q = k$ mod q on the grid: each processor of index (u,k_q) broadcasts its local N_t/p tiles $A_{i,k}$ across its grid row. Similarly, tiles $B_{k,j}$ are broadcast vertically, and there are q parallel broadcast across grid columns. Then all processors update their local $C_{i,j}$ matrices. In several implementations, the broadcasts at each step are organized as pipelined ring algorithms, but any broadcast tree can be used.

The outer loop is described as sequential, but in general there is no synchronization enforced across the nodes, and the progression of each node can be kept independent. Also, overlapping the communications of the next step(s) with the computations of the current step is a classical approach to ensure that nodes are kept active all the time. In fact, the nodes have become so powerful (being multi-GPU accelerated) that prefetching tiles of the A and B matrices is key to performance. Runtime task systems such as StarPU [2] or PaRSEC [3] are able to determine that all $A_{i,k}$ and $B_{k,j}$ tiles are read-only input data that are ready to be sent to the processor owning tile $C_{i,j}$ at the very beginning of the execution. This triggers all the broadcasts in the whole algorithm, meaning that each node ends up receiving (and storing) M_t/p rows of A and N_t/q columns of B. Such an eager communication scheme completely floods the communication network, leading to a drop in performance.

To avoid this congestion phenomenon, a simple solution is to partition the C matrix into blocks and to (logically) compute one block after the other. We use local blocks of size $b \times c$, which means that each processor is in charge of $b \times c$ tiles of C within a block. Globally, each block is of size $bp \times cq$. Assume that bp divides M_t and cq divides N_t for simplicity, and let $X = M_t/(bp)$ and $Y = N_t/(cq)$. Here, b and c are design-parameters, that will be tuned to enhance locality and re-use, as discussed in Section III below. Altogether, the blocks have indices (x,y) ranging as follows: $0 \le x < X$, $0 \le y < Y$. The blocked version writes as shown in Algorithm 1.

Algorithm 1: Blocked outer product algorithm.

```
for $y = 0$ to $Y-1$ in sequential:
    for $x = 0$ to $X-1$ in sequential:
        Compute block $(x,y)$ of $C$:
        for $k = 0$ to $K_t-1$ in sequential:
            forall $i = x(bc)$ to $(x+1)(bc)-1$,
              $j = y(cq)$ to $(y+1)(cq)-1$ in parallel do
                Task $GEMM(i,j,k)$: $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```

The end of each C block can be viewed as a synchronizing barrier: only those tiles of A and B that are needed for the current block are communicated across the network. This corresponds to bp rows of A and cq columns of B. The main idea is to choose b and c so that bK_t tiles of A and cK_t tiles of B would fit in the main memory of each node, in addition to the bc tiles of C. This leads to a total of $T(K_t) = (b+c)K_t + bc$ tiles that need to reside in the main memory of each node. Note that this global barrier is only logical. We actually implemented a lookahead version, as explained below.

Now consider the integration of accelerators. For large problems (with large K_t), $T(K_t)$ tiles will not fit in the memory of the accelerators. To ensure a good data-re-use, we further control the execution of each block by partitioning the internal k loop into chunks of length d, where d is the third parameter of the algorithm. Assume that d divides K_t and let $Z = K_t/d$. Inside block (x,y), chunks are labeled (x,y,z), where $0 \le z < Z$. The algorithm with chunks writes as shown in Algorithm 2.

Algorithm 2: Chunked blocked outer product algorithm.

```
for $y = 0$ to $Y-1$ in sequential:
    for $x = 0$ to $X-1$ in sequential:
        Compute block $(x,y)$ of $C$:
        for $z = 0$ to $Z-1$ in sequential:
            Compute chunk $(x,y,z)$:
            for $k = zd$ to $(z+1)d-1$ in sequential:
                Broadcast $d$ elements of $k$th row of $A$
                  and $k$th column of $B$
                forall $i = x(bc)$ to $(x+1)(bc)-1$,
                  $j = y(cq)$ to $(y+1)(cq)-1$ in
                  parallel do
                    Task $GEMM(i,j,k)$:
                      $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
```

Again, the execution of each chunk terminates by a barrier, local to the node, to prevent that too many elements of A and B to be loaded from main memory to GPU memory. This barrier controls the amount of tiles that are active on a GPU at a given time, but does not enforce synchronization between nodes. Now each chunk requires each node to hold bc tiles of C, and $(b+c)d$ tiles of A and B, for a total of $T(d) = bc + (b+c)d$ tiles. Figure 1 gives a visual representation of these values.

In the chunked version of the algorithm, the global barrier is enforced after each chunk (x,y,z), before beginning the

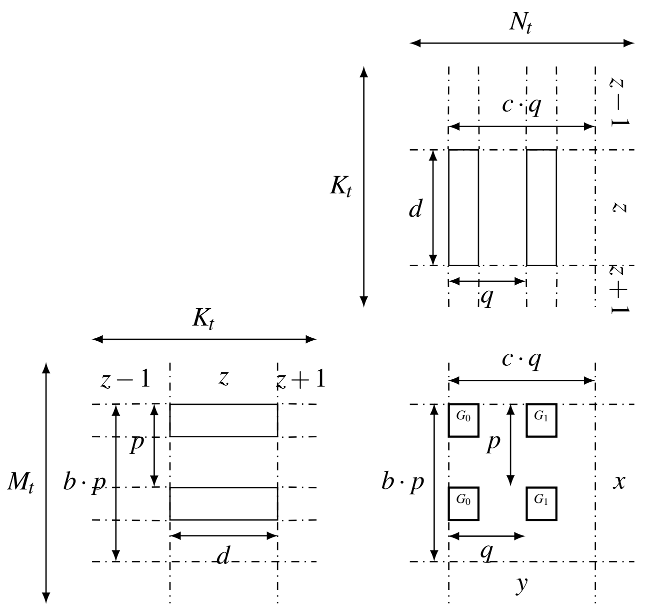


Fig. 1: Major variables used in Algorithm 2.

computations of the GEMMs that belong to the next chunk $succ(x,y,z)$. More precisely, each node (u,v) in the processor grid reaches a local barrier of index (x,y,z,u,v) at the end of chunk (x,y,z), and this local barrier introduces a control dependency to the global barrier of index (x,y,z), which in turns enables the inputs needed for the GEMMs of the next chunk $succ(x,y,z)$ for each node. In Algorithm 2, $succ(x,y,z)$ is computed as follows:

- if $z < Z-1$, then $succ(x,y,z) = (x,y,z+1)$: we proceed to the next fraction of computations for the current block of C;
- if $x < X-1$, $succ(x,y,Z-1) = (x+1,y,0)$: we start the next block of C, which involves the same columns of B but requires new rows of A;
- if $y < Y-1$, $succ(X-1,y,Z-1) = (0,y+1,0)$: we start the next block of C, which requires new columns of B (and new rows of A).

The lookahead version of Algorithm 2 is implemented as follows: at the end of chunk (x,y,z), each local barrier of index (x,y,z,u,v) points to the global barrrier of index $succ^{(\ell+1)}(x,y,z)$ instead of pointing to the global barrrier of index (x,y,z). Here, $succ^{(\ell+1)}(x,y,z)$ denotes the $\ell+1$-st successor of (x,y,z). The lookahead parameter ℓ is the fourth (and last) parameter of the algorithm; it is introduced to allow for prefetching the input data needed for the $\ell+1$ next chunks while computing GEMMs of the current chunk. We only prefetch input data, not the next block of C. Prefetching is more costly when the successors of (x,y,z) involve a different value of y, because B tiles of two different blocks will co-exist in memory. In the general case, prefetching with ℓ requires $\ell(b+c)d$ extra input tiles (from A or B) to fit in memory.

Finally, let G be the number of accelerators per node ($G=6$ for Summit). Assume that G divides c for simplicity. Inside each node, we allocate columns to accelerators in a wrap-around (cyclic) fashion, so that accelerator g of node (u,v) is in charge of computing columns $j = v + qg \bmod (qG)$ of C. Within a block of C, each accelerator is in charge of b rows and $\frac{c}{G}$ columns of C. Hence $T(d,G) = b\frac{c}{G} + (b+\frac{c}{G})d$ tiles must fit into the memory of each accelerator to be able to compute a full chunk without swapping.

III. Communication volume

In this section, we analytically compute an estimate of the number of tiles communicated across nodes on the network, and from main memory to accelerator memory within a node.

A. Problem size

Let Mem_{node} be the available memory per node and Mem_{GPU} be the available memory per accelerator (GPU). We express these quantities in double-precision words rather than bytes to ease the conversion into matrix sizes. On Summit, $Mem_{node} = 64 \cdot 10^9$ doubles and $Mem_{GPU} = 2 \cdot 10^9$ doubles.

First, what is the size of the largest problem that fits within a single node? Assume square matrices with $M = K = N$, there are $3N^2$ coefficients that must fit in the node memory, hence $3N^2 \leq Mem_{node}$. We find $N \approx 145,000$. Now, what is the size of the largest problem whose size would allow the entire C matrix to fit within the available memory of the G accelerators of a node? The $G=6$ accelerators can accommodate a block of C of size, say, $90K \times 90K$ (and we would for instance partition the columns across the GPus, allocating a rectangle of size $90K \times 15K$ per GPU). Such a C block would fill three-quarters of the memory of the $G=6$ GPUs, leaving some space to store few matching A and B tiles. With a square $p \times p$ grid of nodes and square matrices of size N, we need that $N \leq 90,000 \times p$ for the C matrix to entirely reside in the GPU memory of the Gp^2 available GPUs.

B. Communications

We discuss in terms of tiles of size t to clarify the discussion. Consider a $p \times q$ grid of processes and let M_t, K_t and N_t be the total number of tiles in each dimension.

1) Inter-node transfers: How many inter-node communications are triggered by the algorithm? There are $X \times Y$ blocks of C, each b rows and c columns on each processor. Hence $X = \frac{M_t}{bp}$ and $Y = \frac{N_t}{cq}$. Each block can be accounted for independently. Consider a given block owned by process P. For each block, we need to communicate b full rows of A and c full columns of B to process P. Although these communications are partitioned into chunks of size d, we can view them globally. Process P already owns the $1/q$-th fraction of each of these b rows of A and the $1/p$-th fraction of these c rows of B. This means that we send $bK_t(1-\frac{1}{q})$ tiles of A and $cK_t(1-\frac{1}{p})$ tiles of B onto process P. Note that these sends are usually implemented as part of broadcasts, but we focus on the volume of inter-process communication here. There is no inter-process communication involving C tiles. Altogether, process P receives $(b(1-\frac{1}{q}) + c(1-\frac{1}{p}))K_t$ tiles per block of C, and it has XY blocks, hence receives $Comm_{process} = (b(1-\frac{1}{q}) + c(1-\frac{1}{p}))K_tXY$ tiles. With pq processes, the grand total is $Comm_{total} = (b(1-\frac{1}{q}) +$

$c(1-\frac{1}{p}))K_tXYpq = \frac{b(1-\frac{1}{q})+c(1-\frac{1}{p})}{bc}M_tK_tN_t$. Rather than being communication-avoiding, our algorithm is communication-redundant. We voluntarily transmit the same data several times, namely Y times for an A tile and X times for a B tile; this the price to pay to control locality, data re-use, and allow the computation of very large products.

2) Intra-node transfers: Now how-many communications from the memory of each node to the memory of the accelerators? Each tile of C is read either zero time (for $C = AB$ or one time (for $C = C + AB$) and written back once. Again, consider one block of b rows and c columns of C onto one process. The tiles of B are partitioned across the accelerators, so each of them receives the $1/G$-th fraction of the needed cK_t tiles of B (we had $cK_t(1-\frac{1}{p})$ before with inter-node communications, but now we also need to send the tiles local to the process onto the accelerators). Furthermore, each accelerator receives bK_t tiles of A, be it from the main memory of the node or from other accelerators from the NVIDIA link.

Altogether, there are several cases, depending upon the problem size. Overall, the number of tile transfers $Comm_{GPU}$ to each GPU is given by the following equation (see [11]:

$$Comm_{GPU} = \begin{cases} (a) \frac{M_tK_t}{p} + \frac{K_tN_t}{qG} + \frac{M_tN_t}{pqG} \\ \quad \text{if } \frac{M_tK_t}{p} + \frac{K_tN_t}{qG} + \frac{bc}{G} \leq Mem_{GPU} \\ (b) \frac{N_t}{cq}\frac{M_tK_t}{p} + \frac{K_tN_t}{qG} + \frac{M_tN_t}{pqG} \\ \quad \text{if } bd + \frac{cK_t}{G} + \frac{bc}{G} \leq Mem_{GPU} \\ (c) \frac{N_t}{cq}\frac{M_tK_t}{p} + \frac{M_t}{bp}\frac{K_tN_t}{qG} + \frac{M_tN_t}{pqG} \\ \quad \text{if } bd + \frac{cd}{G} + \frac{bc}{G} \leq Mem_{GPU} \end{cases} \quad (1)$$

3) Optimal values for parameters b, c and d: In our implementation of Algorithm 2, we always aim at loading the largest possible block of C that will fit in the memory of the GPUs. This is because the larger the block, the more intensive the data re-use, as shown by numerous studies [22], [18], [13]. This is also confirmed by the number of transfers reported in case (c) of Equation (1): each tile of A is loaded $X = \frac{M_t}{bp}$ times, and we aim at minimizing X. Similarly, each tile of B is loaded $Y = \frac{N_t}{cq}$ times, and we aim at minimizing Y. Typically, we use $b = c$ for square matrices, because square blocks are more prone to data re-use than rectangles. We compute the values of b and c to ensure that $b \times \frac{c}{G}$ tiles of C will occupy, say, three quarters of the memory of each GPU. There are two sub-cases:

• *Case* (c_1)*:* The simplest case is when $b = \frac{M_t}{p}$ and $c = \frac{N_t}{q}$, i.e., when the entire C matrix fits in the memory of the accelerators. In that case, depending upon the amount of leftover memory, we will be able to: (i) either load A and B entirely , and hence only once (case (a)); or only $\frac{c}{G}$ full columns of B, and A tiles will cycle and be loaded several times; or both A and B will have to cycle, because we can only keep $bd + \frac{cd}{G} + \frac{bc}{G}$ tiles in memory (case (c)). Note that case (a) is for small problems only, and case (b) is unlikely to happen.

• *Case* (c_2)*:* This is the general case for large problems when we have to partition C into several blocks because the whole C would not fit into the GPU memories. In that case, $X > 1$, $Y > 1$, and A and B are loaded several times.

In both cases (c_1) and (c_2), once we have chosen b and c as large as possible, we proceed by chunks of depth d, hence we need additional space for bd tiles of A and cd tiles of B: we choose d as large as possible while enforcing the condition $bd + \frac{cd}{G} + \frac{bc}{G} \leq Mem_{GPU}$.

4) Lookahead parameter ℓ: Finally, we point out that using a lookahead further constrains the memory: with $\ell = 1$, we need space for $(b+c)d$ additional tiles in the general case, that of continuing the computations for the same block of C. Section V shows that $\ell = 1$ is enough to ensure good performance when there is a single block of C (case c_1)). However, when C is partitioned into several blocks, we also need to renew the C tiles. When moving to the next block of C, and these additional transfers cannot be fully overlapped with the computations of a single chunk, so we use $\ell = 2$ for case c_2).

IV. Implementation

In this section, we detail some implementation elements that are key to understand the performance of the algorithm.

A. Adaptation of the runtime system to the target architecture

The target architecture, featuring multiple accelerators per node, becomes easily unbalanced in favor of computations, compared to communications. For example, on Summit, with six GPUs per node, and two P9 sockets, the bandwidth between a GPU and the closest socket is 50GB/s, but data flowing from one GPU to the farther socket or to a GPU close to the other socket need to transit through the X-Bus that links the two P9 sockets. Since this bus has a maximum bandwidth of 64GB/s, it can become easily contended. Similarly, to pull or to push data from and to RAU needs to transit through at least one P9 bus, and may need to use the X-Bus between the two sockets. These architectural constraints encourage two steps in the implementation and deployment of the runtime system that supports the matrix multiplication algorithm: first, it is highly beneficial to reduce communications that transit through the X-Bus, and this can easily be achieved by deploying the runtime system with two processes per node (one process per socket). This way, each node of two sockets and 6 GPUs is presented to the algorithm and the runtime as two entities, each with a single socket and 3 GPUs. All data sent explicitly by the runtime system from one process to the other can transit through the X-Bus, but only these data will transit through it; hence there will be no contention created by eager scheduling policies that pull remote data through complex paths in the node. An added benefit of this deployment is that it doubles the number of progress threads for the communication subsystem of the runtime system, enabling it to reach the peak network bandwidth of the hardware, and reducing the contention on the progress queues of the underlying communication system.

The second step taken to increase the performance of the runtime system over this architecture is to enable direct Device-to-Device communications. In the PaRSEC programming paradigm we used, communications are implicit: they are deduced by the runtime system, from the data flow itself, and implemented in the background, while other tasks progress. PaRSEC manages these transfers by keeping a trace of the data movements through a set of meta-data, called the *data copies*. A data copy is a particular instance of a user data, that can reside on a given device. Multiple data copies that represent the same or different versions of the same user data on one or multiple devices are connected under a same set, called a *user data*. The data flow engine passes data copies between tasks, and instantiates each copy on the target device when it decides to run a task on it. By default, all initial data copies reside in the main RAM, when they are initially generated by the user, or received from the network during the distributed progress of the data flow execution.

We extended the PaRSEC runtime to implement an opportunistic strategy: when the runtime system detects that a new data copy needs to be instantiated on a given GPU (typically it did not find a data copy with the appropriate version number on the target device, either because that copy was never uploaded, or because it was reclaimed to allow for another computation), it first searches on the other devices of the same type if another data copy with the appropriate version exists. If such a copy is found, its usage count is updated to prevent the alternative source device to release it, and a device to device transfer using the NVLink capabilities of CUDA is scheduled. Once the copy is instantiated on the target device, the usage count of the copy on the source device is updated, potentially triggering its release in the LRU cache, as was already implemented in the runtime system.

B. Adaptation of the programming language

In order to guarantee that the input parameters b, c, and d will allow maximum reuse and minimal data movement, not only must the implementation guarantee that only tasks that pertain to specific data can execute at a given time, but also that the distribution of work between the accelerators remains fixed. The first point is ensured by the additional control flow that is embedded in the algorithm; the second point, however, needed some extension of the Programming Language. Indeed, work assignment between the different computing devices of a same process is usually opportunistic in PaRSEC: work stealing is the default behavior of all computing devices, including the GPU managers.

PaRSEC, however, follows a last-writer heuristic for GPU work-scheduling in order to minimize the data movement: if a given data has been accessed read-write recently, and its corresponding most recent copy is residing on a given GPU, that device is the only one that can execute another task that accesses the data in read-write mode, until an explicit update of the RAM data copy is requested by the user. We leveraged this policy to statically assign the device that can work on a given block of C, by extending the programming language to allow for the explicit creation of a data copy generated by a task onto a given device. Thus, the GEMM implementation is modified so that each new chain of updates of a given tile starts on a specific device, computed according to the algorithms above. Then, as the algorithm leaves that tile of C resident onto the same GPU until all updates have been applied, all the work on that tile is guaranteed to be assigned to the same accelerator.

V. Performance evaluation

Performance measurements are conducted on Summit, which has over 200 Petaflops of double precision theoretical performance [16] hosted at Oak-Ridge National Laboratory. It consists of 4,600 IBM AC922 compute nodes, each containing two POWER9 CPUs and six Nvidia Volta V100 GPUs. The POWER9 CPUs have 22 cores running at 3.07 GHz, and 42 cores per node are made available to the application. Dual NVLink 2.0 connections between CPUs and GPUs provides a 25GB/s transfer rate in each direction on each NVLink, yielding an aggregate bidirectional bandwidth of 100GB/s.

The program evaluated below implements Algorithm 2 over the PaRSEC runtime system [3], using the Parameterized Task Graph (PTG) DSL featuring the extensions described in Section IV. The algorithm implementation, the driver program and the extensions are all available online in a fork repository [4]. The PaRSEC runtime, the GEMM operation and the driver program were all compiled in optimized (Release) mode, using XLC 16.1.1-2, CUDA 9.2.148, Spectrum MPI 10.3.0.0 available on the Summit programming environment. The BLAS3 GEMM kernel was the one provided in the cuBLAS library provided with CUDA.

We measured the practical peak of the GEMM kernel in this version of cuBLAS and this hardware at 7.2TFLOP/s per GPU. To obtain this value, we ran a single GEMM operation on large matrices that were pre-initialized in the GPU memory, repeated the operation 10 times, and took the fastest run measured.

All performance evaluation results presented below are obtained by measuring the time of executing the parallel double precision real matrix matrix multiply (PDGEMM) with all data residing in the main memory of the nodes (and nothing on the GPU memory). The operation is complete only when the resulting C matrix is back in the main memory of the node, where it started. Thus, the cost of data movement from CPU to GPU memory is included in our measurements, but the experiment reflects a more traditional usage of the PDGEMM routine, where the data was made available by a previous operation in main memory. Each point is measured 5 to 10 times, and all figures showing performance present a Tukey box plot at the mark. On most figures, the measured variability is so small that the box plot is hidden by the mark or the line placed at the mean value, highlighting the stability of the distributed algorithm.

A. Single node runs

First, we consider single node runs in order to find the optimal tile size for the kernel implementation and the available hardware. Figure 2 shows the performance *per GPU*, of a square GEMM of size $M = K = N = t\lceil \frac{70{,}000}{t} \rceil$ (or equivalently $M_t = K_t = N_t = \lceil \frac{70{,}000}{t} \rceil$), for different values of the tile size t on the x-coordinate, and for 1 to 6 GPUs. At this size, each matrix represents 36 GBytes of memory, and the algorithm

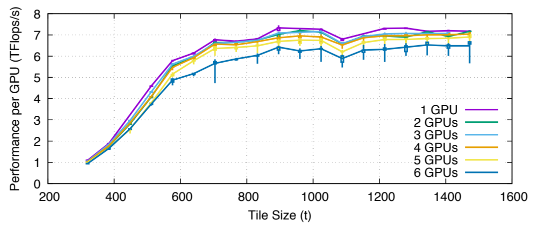


Fig. 2: Performance per GPU of double precision GEMM

has to cycle A and B, with a stationary C (case c_1)). When running with 1 to 3 GPUs, even the matrix C is too big, and it must be cycled by the algorithm (case c_2)). The parameters b,c and d are chosen as described in Section III-B3: $b=c$, $b \times c$ is a divisor of $\frac{M_t}{G}$, and $b \times \frac{c}{G}$ occupies at most three quarters of the GPU memory. Then d is chosen as a divisor of K_t such that $bd + \frac{cd}{G} + \frac{bc}{G}$ tiles fit in the GPU. This run uses a single process to control up to 6 GPUs, incurring potential NUMA effects and overload of the X-bus.

As expected, performance grows with the tile size up to a plateau. This is consistent with the traditional roofline model [25]: until a tile size of $t = 1,024$, the cost of memory transfers dominates the execution time, and there is not enough data reuse on the accelerators to keep them working at maximal efficiency. As soon as a tile size of 1,024 is reached, the arithmetic intensity of the operation is high enough to mask all RAM to GPU memory communication costs, and the performance plateau.

The performance per GPU remains close to the practical peak (¿95% for tile sizes bigger than 1,024), for 1 to 3 GPUs, showing excellent strong scalability at this problem size. When adding more GPUs, from 4 to 6 (maximum available on the hardware), the performance per GPU drops slightly but remains high at 85%. The issues due to X-bus usage and longer times to upload or download memory between the GPUs and the RAM depending on the NUMA bank and the target GPU also translate in a higher variability of the measurements: at 6 GPUs, the first quartile of the runs can get up to 17% slower than the mean value. This performance drop and variability increase is justified by the hardware, and motivates that the other experiments allocate two PaRSEC processes per node. Based on this evaluation, we also select a tile size $t = 1,024$ for all subsequent experiments.

B. Distributed runs

We evaluate the implementation on square grids of processes. Since two processes are assigned the same node, the grid of nodes is $p \times \frac{p}{2}$: two consecutive processes on the process grid are sharing the same tile-rows of the three matrices. Figure 3a shows the performance measured for different problem sizes, using different process grids, and different values of the lookahead.

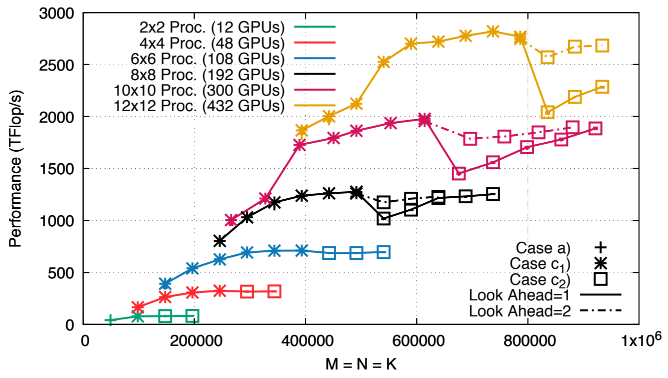
The problem size is represented with the x-coordinate, and the colors of the lines define the process grid size, from 2×2 (12 GPUs) to 12×12 (432 GPUs). Mean values for the measurements are represented with different markers: a *plus* represents the case *a*) above, when the data fit on the GPU memory. A single run in the 2×2 process grid experiments falls in that category. Then, a *star* represents the case (c_1): C is distributed amongst the GPUs and remains static, with parts of A and B cycling multiple times from RAM to GPUs, in order to complete the product. Last, *squares* represent the case (c_2): C itself is too large to fit on the GPU memory, and needs to be cycled with A and B. Last, a plain line links the runs made with a lookahead $\ell = 1$, while some runs with a lookahead $\ell = 2$ are linked together with a dashed line.

In all the runs, the parameters b,c, and d are selected according to the strategy described in Section III-B3: first, we aim at leaving C static on the GPUs, until it is not possible anymore, in which case C is split into even blocks of size $b \times c$ with $c = b$, and then d is used to fill the GPU memory with even chunks of A and B.

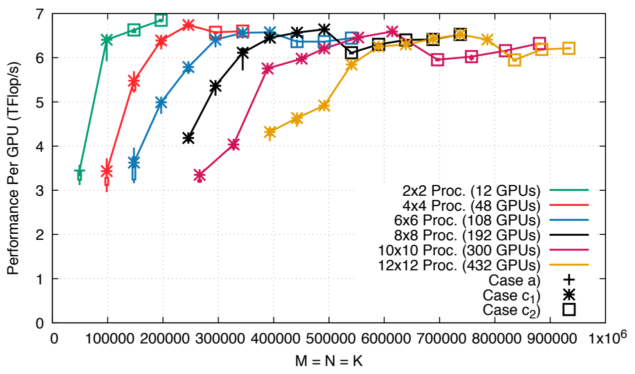
With the problem size increasing, and up to the point where it reaches the case (c_2), the measured performance is consistent with the roofline model: performance grows with the problem size, until it reaches a plateau. Up to a process grid of 6×6, this plateau is maintained, even when the task system transits from the case (c_1) to the case (c_2). Almost no performance degradation is measured during that transition. For the larger runs, however, a steep performance drop is observed when this transition happens. As the scale of the system increases, that drop increases. Then, the performance grows again until it reaches the same plateau.

As illustrated in Figure 3a, that performance drop is due to a small lookahead parameter. With a lookahead $\ell = 1$ (plain lines), only the data necessary for the execution of the next local chunk is prefetched by the runtime system (the input tasks artificially depend upon the execution of the global barrier). When operating on a static C, each new local block of tasks requires to load tiles of A or B from the network The lookahead of 1 is sufficient to allow this load to happen in parallel with the computation. However, when the algorithm reaches the step where the current block of C must be switched with the next one, it needs to upload to the GPU the new block of C, together with all the corresponding tiles of A and B. This rush of data is too high for the network to sustain it within the execution of a single local block, and GPUs become idle during each transition from one C block to another. As the problem size continues to increase, that number of transitions remains the same for a large set of problem and grid sizes, while the overall duration of the computation increases, so the performance increases again. With a lookahead $\ell = 2$, this drop of performance is absorbed by the system much sooner: the idling itself is reduced, by allowing more time to overlap the communication of future tiles with the current computation. We conducted experiments with a lookahead of 3, 4 and 5, without measuring additional performance gains. A lookahead of $\ell = 8$ exceeds the memory capacity of the machine.

Figure 3b represents the same data, but reports the performance *per GPU*, and keeping only the runs with the best lookahead for each measurement. The figure shows more clearly that the task system is capable of reaching close to



(a) Absolute Performance



(b) Performance per GPU

Fig. 3: Performance of double precision matrix-matrix product (PDGEMM) on Summit

peak performance, and of maintaining this performance when the problem size does not fit in the accumulated GPU memory, which is a unique feature at the time of this writing.

To validate the communication model of Section III, we collected the amount of GPU communications during all previous experiments. We measure independently how many bytes are transferred from the RAM to each GPU (H_2D transfer), from any other local GPU to each GPU (D_2D transfer), and from each GPU to the RAM (D_2H transfer). We then compare the amount of data loaded per each GPU ($H_2D + D_2H$), with the communication model, and represent this in Figures 4 and 5. Figure 4 shows the number of tiles loaded, and the number of tiles loaded from RAM only, as well as the number of tiles that should be loaded according to the algorithm analysis, while Figure 5 shows the same information as a ratio to the model prediction. The x-coordinate for these two figures is any run presented above, so they are sorted in an arbitrary order.

There were three case (a) measured, and the rest are cases (c_1) and (c_2), evenly distributed. In all cases, approximately 95% of the number of tiles predicted to be loaded is indeed loaded by a GPU. The number of actual loads is slightly smaller than predicted by the model. This is due to the cache policy of PaRSEC when managing the GPU memory: when a tile is loaded onto the GPU, it remains there unless the space is needed. When it is time to allocate a space for a tile, the PaRSEC runtime needs to eject an old one that is not currently in used. To do so, it maintains a LRU of the currently not-in-use tiles, and ejects the least recently used.

The parameters b,c, and d are selected to use as much memory as possible, but also to distribute the load evenly between the blocks. Consequently, there are always a few hundred tiles of GPU memory that are not in the active set of a given local block. The PaRSEC runtime takes advantage of this slack in memory management to slightly increase the data reuse, compared to the algorithm model, and this explains the 5% difference.

More importantly, this figure shows that about 50% of all loads are device to device: only half of the memory loads are issued to the RAM, and the other half targets another GPU that already loaded the required tile. This is also a consequence of the strong synchronization implemented in the algorithm: as all GPUs work on chunks of updates that are at most 1 away from each other, the probability that they require the same data is high. The opportunistic approach that replaces a RAM access by a device-to-device access hits half the time, reducing by as much half the load on the bus to the RAM.

VI. Related work

The design of matrix product algorithms for high-performance computing platforms has received considerable attention in the recent years. On the theoretical side, several authors have aimed at minimizing the number of communications for rectangular matrices of arbitrary sizes, since the seminal paper of Hong and Kung on the I/O pebble game [12]. Due to lack of space, we refer to a recent report [15] which provides a good overview and multiple references. Cache-oblivious algorithms are surveyed in [10], [20].

Out-of-core algorithms for matrix product have been developed to optimize the number of transfers between hard disks and RAM. The pioneering work of Toledo [22], [13] suggested to load three equal-size square blocks of A, B and C into main-memory, while a refined analysis [18] suggests to load the largest possible block of C, one slice of B and to cycle tiles of A. The chunked algorithm is an extension of this approach to multi-GPU accelerated platforms, where the chunk is needed to increase granularity and properly feed the GPUs.

On the practical side, many libraries provide an implementation of matrix-product for distributed-memory machines [19], [17], [8], [7], [9]. Only SLATE [9] is capable of dealing with multi-GPU accelerated nodes, and currently suffers from the limitation that the whole C matrix must fit into the (cumulated) memory of the accelerators. In other words, there must be a single block of C, this is case (c_1) of Section III-B3. Oour implementation with PaRSEC does not have any limitation.

VII. Conclusion

This work has introduced a simple and flexible matrix-multiplication algorithm for multi-GPU accelerated distributed-memory platforms. We designed a prototype implementation that achieves a sophisticated management of transfers from node memory to GPU memory, thereby

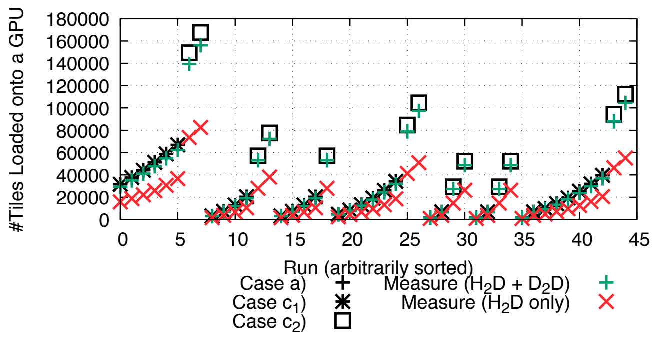


Fig. 4: Number of tiles loaded onto a GPU

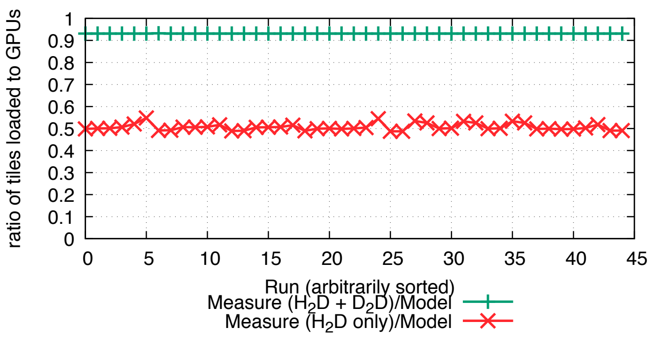


Fig. 5: Comparison with the model

guaranteeing optimal data re-use. The GPUs are kept fully active by using a partitioned version of the computations into chunks whose size is large enough to launch many GEMMs in parallel, while allowing all input data to fit into GPU memory. Chunk data transfers are orchestrated so as to prevent swapping, but with some overlap to avoid starvation and unnecessary synchronization. Altogether, we report preliminary performance results that squeeze 85% of the peak performance of the platforms, and this even for larger instances that do not fit into the cumulated memory of the platform GPUs. This very good performance is achieved within a short time-frame, owing to the flexibility and extended capabilities of the PaRSEC task runtime system. It would be straightforward to implement the algorithm onto a different GPU-accelerated distributed-memory platform.

Future work will be devoted to extending the algorithm to handle the case of matrices with irregular tiles. More precisely, in the TESSE framework [21], we have to multiply matrices whose tiles can have very different sizes across rows and columns.

Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. It used resources of the Oak Ridge Leadership Computing Facility at ORNL, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development*, 38:673–682, 1994.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *CCPE*, 23(2):187–198, 2011.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *IEEE Computing in Science Engineering*, 15(6):36–45, 2013.

[4] G. Bosilca, A. Bouteiller, T. Herault, et al. Publicly available repository of the code. https://bitbucket.org/herault/parsec%2dgemm%2dgpu.

[5] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scient. Prog.*, 5:173–184, 1996.

[6] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 107–114, 1996.

[7] Distributed Parallel Linear Algebra Software for Multicore Architectures. DPLASMA. http://icl.utk.edu/dplasma.

[8] Elemental: C++ library for distributed-memory linear algebra and optimization. Elemental. https://github.com/elemental/Elemental.

[9] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *SC'2019*. ACM Press, 2019.

[10] K. Goto and R. A. v. d. Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Software*, 34(3):12:1–12:25, 2008.

[11] T. Herault, Y. Robert, G. Bosilca, and J. Dongarra. RR-9289: Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over PARSEC. Technical Report hal-02282529, INRIA, 2019.

[12] J.-W. Hong and H. Kung. I/O complexity: the red-blue pebble game. In *STOC '81: Proceedings of the 13th ACM symposium on Theory of Computing*, pages 326–333. ACM Press, 1981.

[13] D. Ironya, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distributed Computing*, 64(9):1017–1026, 2004.

[14] J. Kurzak, M. Gates, A. Charara, A. YarKhan, I. Yamazaki, and J. Dongarra. Linear systems solvers for distributed-memory machines with gpu accelerators. In *Euro-Par 2019*, pages 495–506, 2019.

[15] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. *arXiv e-prints*, page arXiv:1908.09606, Aug 2019.

[16] Oak Ridge National Laboratory. Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/.

[17] Parallel Linear Algebra PACKage. PLAPACK. http://www.cs.utexas.edu/users/plapack.

[18] J.-F. Pineau, Y. Robert, F. Vivien, and J. Dongarra. Matrix product on heterogeneous master-worker platforms. In *ACM SIGPLAN PPoPP'2008*, pages 53–62. ACM Press, 2008.

[19] Scalable Linear Algebra PACKage. http://www.netlib.org/scalapack.

[20] M. D. Schatz, R. A. van de Geijn, and J. Poulson. Parallel matrix multiplication: A systematic journey. *SIAM J. Scientific Computing*, 38(6):C748–C781, 2016.

[21] Task-Based Environment for Scientific Simulation at Extreme Scale. TESSE. https://www.nsf.gov/awardsearch/showAward?AWD%5FID=1450300&HistoricalAwards=false.

[22] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[23] Top500. Top 500 Supercomputer Sites, June 2019. https://www.top500.org/lists/2019/06/.

[24] R. A. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical report, University of Texas at Austin, Austin, TX, USA, 1995.

[25] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Comm. ACM*, 52:65–76, 2009.

Appendix: Artifact Description

This appendix follows the form available to the Paper Artifact Description, as provided by the Author Kit, under the SC Reproducibility Initiative.

Are there computational artifacts such as datasets, software, or hardware associated with this paper?

Yes

Summarize the experiments reported in the paper and how they were run:

The experimental method is described in details within the paper. Performance measurements were conducted on Summit, hosted at Oak-Ridge National Laboratory.

The program evaluated implements Algorithm 2 over the PaRSEC runtime system [3], using the Parameterized Task Graph (PTG) DSL featuring the extensions described in Section IV of the paper. The algorithm implementation, the driver program and the extensions are all available online in a fork repository at https://bitbucket.org/herault/parsec-dgemm-gpu, at commit 0fc319d07ad632005409004b407cd058891c0016. The PaRSEC runtime, the GEMM operation and the driver program were all compiled in optimized (Release) mode, using XLC 16.1.1-2, CUDA 9.2.148, Spectrum MPI 10.3.0.0 available on the Summit programming environment. The BLAS3 GEMM kernel was the one provided in the cuBLAS library provided with CUDA.

All performance evaluation results presented below are obtained by measuring the time of executing the parallel double precision real matrix matrix multiply (PDGEMM) with all data ready in the main memory of the nodes (and nothing on the GPU memory). The operation is complete only when the resulting C matrix is back in the main memory of the node, where it started. Each point is measured 5 to 10 times, and all figures showing performance present a Tukey box plot at the mark.

Software Artifact Availability:

All author-created software artifacts are maintained in a public repository under an OSI- approved license.

Hardware Artifact Availability:

There are no author-created hardware artifacts.

Data Artifact Availability:

There are no author-created data artifacts.

Proprietary Artifacts:

None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:

https://bitbucket.org/herault/parsec-gemm-gpu

Relevant hardware details, e.g., system names, makes, models, and key components such as CPUs, accelerators, and filesystems:

At the time of submission, Summit consists of 4,600 IBM AC922 compute nodes, each containing two POWER9 CPUs and six Nvidia Volta V100 GPUs. The POWER9 CPUs have 22 cores running at 3.07 GHz, and 42 cores per node are made available to the application. Dual NVLink 2.0 connections between CPUs and GPUs provides a 25GB/s transfer rate in each direction on each NVLink, yielding an aggregate bidirectional bandwidth of 100GB/s.

Operating systems and versions:

Linux 4.14.0-115.8.1.el7a.ppc64le

Compilers and versions:

xl/16.1.1-2, spectrum-mpi/10.3.0.0-20190419

Applications and versions:

N/A

Libraries and versions:

cuda/9.2.148, essl/6.1.0-2

Key algorithms:

Algorithm 2 of this paper

Input datasets and versions:

N/A

Modifications made for the paper:

no modification of the hardware was done for this paper.