Comparing and Contrasting User and Runtime Directed Data Placement Strategies for Owner-Compute, Multi-Accelerator Distributed Task Based Scheduling

Aurelien Bouteiller[1], Qinglei Cao[2], Joseph Schuchart[3], and Thomas Herault[4,1]

[1] The University of Tennessee, Knoxville
[2] Saint Louis University
[3] Stony Brook University
[4] Centre Inria de l'Université de Bordeaux

Abstract. Given GPU accelerators' high arithmetic capacity, reducing data motion and optimizing locality are critical to achieving high performance. The task-based programming paradigm, as employed in the PaRSEC micro-task runtime system, enables the decoupling of data distribution and computation mapping to resources from the algorithm's base expression. In this paper, we leverage this capability to explore the performance impact of several data placement strategies–some automatic and runtime-directed, and some user-directed–for the owner-compute scheduling model in the context of split-memory accelerators. We implement three different strategies for data and task mapping: a randomized first-touch policy that assigns data randomly to an accelerator, a load-balancing strategy that assigns data to the accelerator with the lowest load, and we compare it to a user-directed strategy that minimizes cross-accelerator traffic by placing tasks according to a cross-memory bandwidth minimizing strategy. We carry the evaluation on a variety of multi-GPU accelerated systems , including the Frontier system, and demonstrate that runtime-directed automatic data placement can improve locality compared to naive strategies, but also highlight that the capability of easily having modifiable user-directed data placement is of crucial importance to achieve peak performance.

Keywords: Task-based runtime · Matrix computations · Cholesky factorization · accelerator.

1 Introduction

The current dominance of accelerators in leadership class High Performance Computing systems has motivated the emergence of the task-based programming style. This programming model enables the dynamic execution and mapping of the computation on computing resources and a greater asynchronous execution of tasks, hence enabling the execution to reach a higher portion of

the computational peak. Another important aspect is the overlap between computation and the motion of data between nodes and memory hierarchies. The task-based programming paradigm, as employed in the PaRSEC micro-task runtime system [1, 2], enables the decoupling of the data distribution and mapping of computation to resources from the base expression of the algorithm [3]. As a consequence, it becomes easy to modify these mappings and explore the performance impact between a number of strategies, some automatic and runtime-directed, and some user-directed. In this paper, we focus on the comparison and contrast of different data placement strategies for the owner-compute scheduling model in the context of split-memory accelerators–that is, when host memory and accelerator memory are separate domains, or when accessing host memory through Unified Virtual Memory (UVM) incurs a significant cost [4].

We implement in multiple applications (LLT Cholesky factorization, matrix multiplication [5]) three different strategies for data and task mapping: a randomized first-touch policy that assigns data randomly to an accelerator, a load-balancing strategy that assigns data to the accelerator with the lowest load, and we compare it to a user-directed strategy that minimizes cross-accelerator traffic by placing tasks according to a cross-memory bandwidth minimizing strategy. We also compare these strategies to using UVM to let the hardware position data on-demand on the site of computation as a baseline. An important consideration that we take into account is the positioning of data received from the network in distributed systems that are capable of depositing message payload directly in accelerator memory, this is critical for example on the Frontier system where network interfaces have closer affinity with accelerator memory banks than host memory banks [6].

The evaluation is carried out on a variety of multi-GPU accelerated systems (using the PaRSEC capabilities to schedule tasks with CUDA, HIP, OneAPI [2]), including the Summit and Frontier systems. We finally discuss how these strategies, including user-directed strategies, can be implemented in a manner that maintain the separation of concerns between expressing the correctness of the algorithm and the mapping of tasks on resources.

The rest of this paper is organized as follows: in Section 2 we provide more details about the problem, and how it relates to micro-task scheduling; in Section 3 we present the data placement strategies that we have implemented in the PaRSEC runtime, and at the application level; in Section 4 we present the application benchmarks, and present the experimental evaluation in 5. Lastly, we present related works in Section 6 before we conclude.

2 Background

2.1 The PaRSEC Runtime System

PaRSEC [1,2] is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed, multicore, heterogeneous architectures. PaRSEC target applications are expressed as a Direct Acyclic Graph of tasks

with labeled edges designating data dependencies. In technical terms, the runtime handles the architecture-aware part, assigns computation loads to the cores and accelerators, orchestrates the movement of data between memory domains (communication between nodes, staging data to accelerators) in a manner that overlaps communication and computation, and uses a dynamic fully-distributed scheduler. The runtime uses the DAG representation of the algorithmic data dependencies to identify required data motion, ensure appropriate memory consistency, and prevent incorrect concurrent accesses.

These graphs are constructed from a high-level expression of parallelism, accessible through multiple Domain Specific Languages (DSLs) that are provided with the PaRSEC runtime. Some of these DSLs embrace the traditional inspector-executor model [7], while others express the parallelism in a graph-oriented expression that avoids the cost of executing skeleton code [8, 9]. These expressions have been used with great success to express regular and irregular algorithms, like linear algebra with `DPLASMA` [5], Sparse tensor contraction used in Chemistry with NwChemEx [10,11], sparse, mixed-precision operators used in geospatial modeling and 3D unstructured mesh deformation applications [12–17], Genomics [18], and climate modeling [19] (Gordon Bell Climate award winner). Additional details on PaRSEC enabled applications are provided in [2].

A critical aspect of the micro-task model in general, and of the PaRSEC dataflow algorithmic representation in particular is the clear separation of concerns between the expression of parallelism, on one hand, and the expression of the data placement and scheduling policy on the other. For example, in [3, 20] we demonstrated that the same algorithms can operate on data using the well-known Scalapack [21] data format, or the novel tile-based format used in more modern linear algebra libraries [5, 22]; Meanwhile, in [15, 17] we demonstrated how the runtime can overlap the transformation of an initial, user-provided data distribution to a different, more optimal distribution used during computation.

2.2 The Data Placement Problem

While the micro-task dataflow model enables the unique capability of optimizing data placement and distribution independently of the algorithmic expression, it also raises the question of how to create an efficient placement when it is not hardcoded in the algorithm. In traditional SPMD/MPI programming (and its variants, like MPI+CUDA/HIP), the data distribution is inherently hardcoded early in the algorithm implementation design. Arbitrary placement of data is, in theory, possible, but would result in an untractable irregular decomposition of the work that is hard to reason about for the programmer, and in practice rarely used.

In contrast, in a micro-task dataflow algorithm, the data placement can be substituted without changing the inherent expression of the algorithm. This is especially true of owner-compute task distributions, that schedule tasks to devices according to the existing locality of data, that is, preferring to locate the task execution so that write accesses to data are device-local. However, the problem of finding an optimal, or even efficient data placement remains both

critical and difficult. Thus, micro-task scheduling must follow an efficient data access pattern to achieve high efficiency, and the dimension of the search space for possible placements is larger and more complex due to irregular, runtime dynamic placements being practical options that can be deployed in practice.

3 Multi-Accelerator Data Placement Strategies

To evaluate the impact of data placement on the execution of micro-tasks, we implemented multiple data placement strategies in the PaRSEC runtime. The placement is selected when the data a data-dependent task is ready for execution on an accelerator. As previously described, we consider the case of the owner-compute task scheduling model, that is, the selection of data placement has a strong impact on the scheduling of tasks using that data. In particular, when a write-to data is selected for upload into a given accelerator, the tasks that write to said data will execute on that same accelerator until the data needs to be staged out (for lack of memory on the accelerator), or transferred to another accelerator or host cpu (due to a read access by a task scheduled on that device). When a write-to data is not already available on an accelerator device type, the data placement strategy will select an appropriate accelerator to upload the data to, with consequences on the scheduling of the task that is immediately using that data, but also on future successor tasks that may write, or read from that data. Tasks that write to that data will be scheduled on the same accelerator, while tasks that read from that data will prefer being scheduled on that accelerator if possible. Hence, data placement has an indirect effect on the load balancing of tasks.

3.1 Host to Device Placement Strategies

Data can have multiple copies, existing on multiple devices. The runtime tracks read and write access to copies, and makes sure that coherency is maintained. For example, a host CPU access to data that has been modified on a device will trigger the repatriation of that data to the host memory. Conversely, a data block may have multiple copies on multiple devices that are in read access simultaneously.

Load-based data placement strategy: In this strategy, when a data needs to be staged in, the target accelerator is selected using a load-based heuristic. When a task is scheduled onto a device, its Estimated Time of Completion (ETC) is computed given the existing load on the device, and the task duration estimate (hooks are provided for the user to customize irregular task duration estimates). If the task ETC is lower on a given device, write-to task data is staged onto that device, and the load on the device is updated accordingly. When the task completes, the device load is reduced accordingly.

The load balancing strategy can come into conflict with the data reuse strategy. In order to minimize data motion, it is ideal to prefer scheduling a task on

an accelerator where most of its input data are already resident. This however can result in systematically preferring (at the extreme) a single device, as input data are already present on that device, while neglecting to load-balance to other devices. For example, if a task has two read data A and B, and writes to a third data C, if A and B are already resident on a given accelerator, it is beneficial to upload C to that same accelerator to minimize data motion overall. However, considering the owner compute scheduling model, that also means that the task will execute on that accelerator, regardless of the load imbalance that could be created. Hence, the load-based placement strategy has a *skew* parameter. When the load imbalance is below the skew threshold (for example the ETC is less than 20% more than the optimal load balance), the write data is staged to the accelerator with preexisting read data (henceforth also scheduling the task to that accelerator). If the load imbalance is above the threshold, preexisting read data are ignored, and the write data is staged to the accelerator that offers the best load balance (a consequence is that read data will have to be migrated from one accelerator to another).

The expected advantage of this strategy is that it enforces a fair load balance between the devices, while at the same time minimizing data motion when load balance is not too skewed. We compare the effect of the skew parameter in the experimental section.

User-directed placement strategy: In this strategy, when a data needs to be staged in, the target accelerator is selected by calling a user-provided function. The user provided function takes as parameter the task identifying parameters (e.g., coordinates of the data block accesses during the task) and computes either an absolute mapping onto an accelerator, or an inherited mapping from the input data. Individual data blocks in a data collection can be tagged with a memory affinity that expresses an user directed preference for that data when it comes to selecting which accelerators should have that data into residence.

Randomized data placement strategy: In this strategy, when a data needs to be staged in, the target accelerator is picked at random. This strategy is designed as a comparison point with other strategies but is expected to yield a good load balance, possibly at the expense of sub-optimal data reuse.

No placement strategy (UVM): In this strategy, data placement (and data motioning to accelerators in general) is deferred to the CUDA runtime (or similarly, RoCM, for simplicity this discussion will consider CUDA, without loss of generality). Again, this strategy is designed as a comparison point with other strategies and is expected to be inferior in most cases. The runtime does not schedule the `cudaMemcpyAsync` calls to pre-position data before the execution of tasks on a given accelerator. Instead, the memory is allocated with the CUDA host allocator, thus enabling coherent access to memory between devices and host. Host memory pointers are passed directly into the accelerator kernels executing on the device (e.g., CuBLAS calls), and the UVM system is in charge of resolving cross-device accesses in a manner of its choosing. We implemented two

different strategies: in the first strategy, memory is allocated in host memory with `cudaHostAlloc`, using the `cudaHostAllocPortable, cudaHostAllocWriteCombine` flags. In the second strategy, we provide additional hints to the UVM mechanism by allocating the memory in managed mode (`cudaMallocManaged`), and we take advantage of the fine-grain data tracking performed in PaRSEC to issue timely `cudaMemAdvise` to the device that will execute the dependent task. The owner-compute scheduling employs the same heuristic as if the data had been placed using the user-directed placement strategy, so as to have a scheduling that is as close as possible and comparable to the explicit data placement strategy.

3.2 Network to Device Placement Strategies

Data arriving from the network must have a buffer to be deposited into. Again, selecting in what device memory that buffer is located can have a major impact on the data motion cost and load balancing of the owner-compute scheduling. In particular, on the Frontier system, network interfaces are not attached to the host CPU PCI-E bus, but instead are directly attached to the accelerator devices, and depositing all reception into host memory causes significant unnecessary cross-traffic.

Host-memory placement strategy: In this strategy, all data received from the network are received into the host memory. When the dependent tasks are scheduled, the normal host-to-device placement strategy is employed (in the experiments we use the combination with user-directed node-local placement).

User-directed accelerator-memory placement strategy: In this strategy, the runtime uses the same user-provided functions described for the node-local placement to select the target accelerator from which the memory buffer is allocated. The buffer is allocated in the accelerator memory on-demand by the communication engine and is passed directly to the communication library (e.g., an MPI library that has GDRcopy enabled). When the internode communication completes, the data is already staged into the target accelerator, and the dependent task can immediately use that data without further host-to-device copies. In many cases, the write-to data is already local, and this strategy has the effect of collocating read-only data received from the network with the write-to data.

Random Accelerator-memory placement strategy: This strategy is similar but uses a random selection function to pick the target accelerator where the reception buffer is allocated. Again this strategy is designed as a comparison point; it is simple but does not preserve collocation of data between read-only data received from the network and write-to data that are already node local.

4 Applications

To evaluate the data placement strategies described above, we use the `DPLASMA` Linear Algebra package. We focus our analysis on matrix-matrix multiply (GEMM),

and Cholesky factorization (POTRF). The implementation of these algorithms in dataflow form employ the Parameterized Task Graph input language to express dependencies between tasks. More details on the implementation can be found in [2,5]. The internode parallel partitioning is expressed as a parameterized function, but the intra-node data distribution and scheduling is dynamic.

User-directed Data Placement Function User-directed data placement is expressed through a function that marks the input data with a preferred device. This is similar to how normal memory advise is typically performed in NUMA systems: individual blocks of the input matrices are tagged with their preferred device, and the runtime then takes that preference into account when selecting onto which device to position data during their first access on a device-enabled task.

For the Cholesky factorization, tiles of the triangular matrix are distributed following a 2D-cyclic distribution [23] between the nodes. Local tiles for each node are then distributed between the different GPUs of the node are then distributed according to a second level of 2D-cyclic distributions: consider a given rank (p, q) in the 2D grid (P, Q) with $G = G_P \times G_Q$ GPUs; Tile $A(m, n)$ of the matrix A is set to be located on rank $(m \mod P, n \mod Q)$, and on GPU $G_p * G_Q + G_q$ where $G_p = (m/P) \mod G_P$, and $G_q = n/Q \mod G_Q$.

For GEMM, the tile that is accessed read-write by the single task type GEMM is also first distributed between the ranks following a 2D-cyclic distribution. Local tiles for each rank are then distributed between the different GPUs by taking the tiles of a given tile-column and distributing them between the GPUs in a round-robin fashion (1D-cyclic distribution between the tile-rows). Consider a given rank (p, q) in the 2D grid (P, Q) with G GPUs; Tile $A(m, n)$ of the matrix A is set to be located on rank $(m \mod P, n \mod Q)$, and on GPU g where $g = m \mod G$.

5 Experimental Results

5.1 Experimental Systems

We perform our experiments on three different systems that cover a range of hardware capabilities. The Guyot system, hosted at the university of Tennessee, is a single-node multi-accelerator system comprising 2x64 cores AMD EPYC 7742 Processors with 2TB of memory (4 NUMA domains), and Eight NVIDIA A100-SXM4 80GB (2 devices per NUMA); the operating system is Rocky Linux release 9.4 with kernel 6.1.114-1.el9.elrepo, and CUDA 11.8. The Summit system, hosted at Oak Ridge National Laboratory is a supercomputer with 4,608 compute nodes, each compute node sports 2 IBM POWER9 processors, with 512GB of memory, 6 NVIDIA Volta V100 SXM2 16GB, the network interconnect is Mellanox EDR 100G Infiniband. The Frontier system, also hosted at Oak Ridge National Laboratory is the current number two supercomputer on the Top 500. Frontier compute node consists of a 64-core AMD EPYC 7A53 CPU with 512 GB of DDR4 memory in 4 NUMA domains. Each node also contains 4 AMD

MI250X, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node. The programmer can think of the 8 GCDs as 8 separate GPUs, each having 64 GB of high-bandwidth memory (HBM2E) [6]. Each MI250X has an attached HPE Slingshot-11 network interface card (200Gbps), for an overall node bandwidth of 1000Gbps. We use launcher parameters (e.g., `-gpu-bind=closest` with Slurm) to select GPUs that are close to the host CPU cores NUMA domain, as well as close mapping to network interfaces. Huge pages and TLB are set to 2MB.

5.2 Comparison between placement strategies on a single node

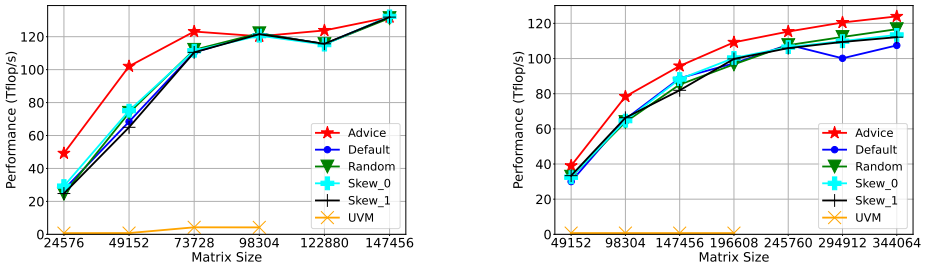

Fig. 1: Performance comparison between on-node, multi-accelerator data placement strategies on a single node (Guyot, 8x A100-SXM4-80G). Left: DGEMM; Right: DPOTRF.

Figure 1 presents the performance of GEMM (left) and PORTRF (right) on the Guyot system with 8 A100-SXM4 GPUs managed in a single PaRSEC process. We present problem scaling, that is, the size of the input matrix is increased meanwhile the number of hardware resources remains constant. Despite 1) using recent NVIDIA hardware and CUDA SDK, and 2) using the knowledge extracted from the PaRSEC runtime to issue timely `cudaMemAdvise` operations to hint at the proper memory placement, the UVM strategy achieves a very low fraction of peak, and fails to scale with the problem scale (results for large problem size were not collected due to the excessive runtime.)

Now considering only strategies with explicit data motioning performed by the PaRSEC runtime (i.e., using `cudaMemcpyAsync`), we compare three variants of the load-balancing heuristic with a randomized placement of data onto GPUs. The Default strategy favors data locality by dispatching additional data required by a task to the same accelerator that already holds some of its input data until the load imbalance exceeds 20%, in which case it will favor the accelerator with the lowest load. When skew=1, the accelerator with the lowest load is always selected. When skew=0, the accelerator that maximizes data reuse is always selected. The randomized strategy picks an accelerator at random. In this case, the automated, runtime heuristic strategies all perform similarly.

Still, the user-directed strategy performs better than the automated strategies. In the POTRF case, the performance advantage is moderate (15-20%) and across the problem size range; due to the better reuse of data the user-directed strategy provides. In the GEMM case, the performance advantage is very significant for small problem sizes with performance close to doubling but subsides as the input matrices grow larger, and the overall balance between computation and data motion diminishes.

5.3 Comparison between placement strategies on multiple nodes

Figure 2 presents the results of the same data placement strategies when considering the case of a multi-node execution on the Summit system. The node counts vary from 4 to 256 nodes with 1 MPI process per node (with the PaRSEC runtime dynamically scheduling between the 6 V100 GPUs in each process). MPI is employed within the PaRSEC runtime to send and receive data over the network; data that is received in host-allocated memory buffers, and the selection of the accelerator memory to upload to, as well as the actual host-to-device data transfer, are deferred to the time when the dependent task is ready for execution.

Again, we notice a notable advantage for the user-directed accelerator data placement compared to automated heuristics across the board. In the GEMM case, the advantage is maintained when scaling either the number of nodes or the problem size. In the POTRF case, the advantage grows significantly when the problem size is large: the automated strategies place write data at random (or random+load balance) on the accelerator, resulting in read-only data received from the network being replicated on all accelerators (as they are accessed by tasks co-located with write data positioned on all accelerators).

Another interesting consideration is the results for the skew=0 case in PORTF, which significantly underperform. This heuristic forces the strict placement of data without regard for load balance, which in this case results in placing most data on the same accelerator, and causing a subsequent compute load imbalance when tasks are dispatched.

5.4 Comparison for placement strategies for inter-node reception buffers

In Figure 3 we consider the effect of placing receive buffer for data incoming from the network. In the left figure (strong scaling, problem size is fixed at N=131,072), we increase the number of nodes from 4 to 32 (PxQ=4x4 to PxQ=16x8, 4 MPI processes per node, 2 GCDs per process). Performance for the host reception buffers underperforms and does not scale. The two in-accelerator memory placement strategies we consider are as follows: advise is using the user-defined memory placement to select where to place the reception buffer, and random picks an accelerator at random; both strategies achieve similar performance, and increase performance by an order of magnitude compared to the host-placement strategy. In the weak scaling case (right figure) problem size increase commensurate with the number of nodes, that is PxQ=4,

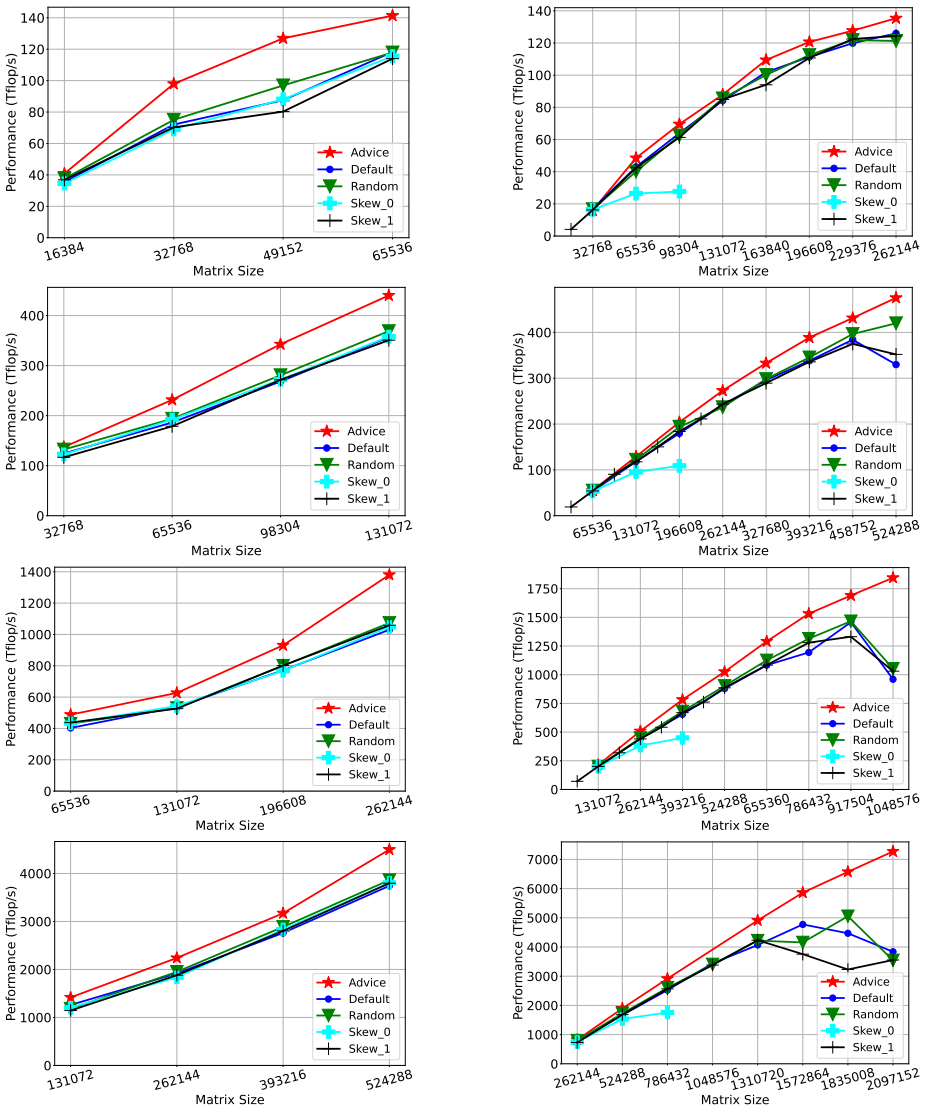

Fig. 2: Performance comparison between on-node, multi-accelerator data placement strategies on multiple nodes (Summit, 6x V100-SXM2-16GB per node). Top: 4 nodes; upper: 16 nodes; lower: 64 nodes; bottom: 256 nodes. In each row, left: DGEMM; right: DPOTRF.

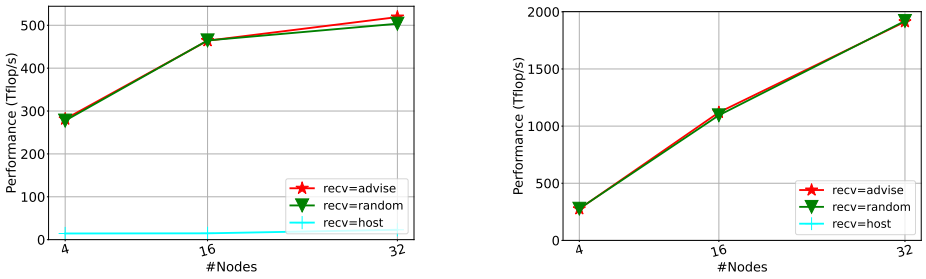

Fig. 3: DPOTRF Performance comparison for off-node inbound data placement multi-accelerator strategies (Frontier, 8x MI250X GCD 80GB per node, 4 processes per node, 2 GPU per process); left: strong scaling N=131072; right: weak scaling

N=131,072; PxQ=8x8 N=262,144; PxQ=16x8 N=327,680). Both in-accelerator memory placement strategies achieve similar performance and scale linearly.

6 Related Work

Efficient data placement strategies are crucial for optimizing performance in multi-accelerator distributed task-based scheduling. Prior research has explored various aspects of data placement, spanning geographically distributed cloud environments, task-based workflows, heterogeneous memory systems, and load-balancing strategies.

Li et al. [24] propose an optimal data placement strategy for geographically distributed cloud environments, considering capacity limitations and load balancing. While their work focuses on reducing data transmission costs in a distributed cloud setting, similar challenges arise in multi-accelerator task-based scheduling, where data movement can become a bottleneck.

In the context of scientific workflows, Sun et al. [25] introduce an in-staging data placement technique to facilitate asynchronous coupling of task-based workflows. Their approach aims to minimize data transfer overheads by staging intermediate data closer to computation resources, an insight that can be extended to task-based runtime systems that schedule computations across multiple accelerators.

More recently, Klinkenberg et al. [26] examine phase-based data placement optimization in heterogeneous memory architectures. Their work highlights the importance of leveraging different memory technologies, such as high-bandwidth memory (HBM) and non-volatile memory (NVM), to optimize performance. Similar principles apply to multi-GPU scheduling, where memory-aware placement can significantly impact execution efficiency.

Several studies have tackled the challenge of load balancing and memory-aware scheduling in task-based environments. Lifflander et al. [27] present a communication- and memory-aware load-balancing model that considers both

computation and data movement costs. Gonthier et al. [28] explore memory-aware scheduling for multi-GPU task execution, introducing novel eviction policies and task allocation strategies to minimize data movement overheads. These efforts align with the need for efficient data distribution mechanisms in task-based runtime systems targeting accelerators.

Task-based runtime systems have also been leveraged to improve data redistribution and scheduling efficiency. Cao et al. [3] propose a flexible data redistribution approach within the PaRSEC runtime, demonstrating its effectiveness in handling irregular data distributions. Similarly, Haidar et al. [29], Cojean et al. [30], and Cao et al. [16] present different works focusing on Cholesky factorization.

General Matrix-Matrix Multiply (GEMM) is a fundamental operation in many scientific and machine-learning applications. Efficient implementations of GEMM are crucial for optimizing performance on modern accelerator architectures. Prior research [31, 32] has explored various optimizations, including tiling, memory hierarchy utilization, and parallel execution strategies to maximize throughput and minimize data movement overheads.

7 Conclusion and Future Work

It comes as no surprise that reducing data motion is critical to achieving high performance. In this paper we present a number of strategies, some automated and some user-directed, that strive to select an efficient data placement for data to be uploaded to accelerators when executing owner-compute tasks in a production quality micro-task runtime system. Our results confirm that inappropriate placement can cause significant performance reduction and that relying on unified virtual memory is ineffective. The performance difference between fully user-directed placement and runtime automated placement remains significant, and highlights the importance of having hooks for user to be able to express an optimized placement in micro-task runtimes. Among automated heuristics, some strategies that appear natural, like always favoring maximum data reuse, show they can be detrimental, as they may force the execution of tasks to become imbalanced. Overall, automated strategies that preserve both data reuse and load balancing perform better and can achieve in many cases lower, but competetitive performance when compared with user-directed placement at a reduced engineering cost. Last, we evaluated the effect of placing reception buffers on modern systems like Frontier, where network interface cards are attached directly to accelerators, resulting in host-memory receive buffers having an unacceptable level of overhead. On such systems, placing received data onto the correct accelerators can uplift performance by up to an order of magnitude.

In future works, we would like to investigate more varied application patterns, including irregular application patterns where the optimal data placement may be harder for users to express (e.g., tensor contraction [11], mixed-precision algorithms [16]).

Acknowledgments. This research was supported in part by internal awards from Saint Louis University (Grant-0001651 and PROJ-000498) and the National Science Fundation (NSF awards #1931384, #1931387). For computer time, this research used the compute node at Innovative Computing Laboratory of the University of Tennessee, Knoxville, and the Summit and Frontier supercomputer at Oak Ridge National Laboratory (Project ID: CSC574).

References

1. G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra. PaRSEC: A Programming Paradigm Exploiting Heterogeneity for Enhancing Scalability. *Computing in Science and Engineering*, 99:1, 2013.
2. A. Bouteiller, T. Herault, Q. Cao, J. Schuchart, and G. Bosilca. PaRSEC: Scalability, flexibility, and hybrid architecture support for task-based applications in ECP. *IJHPCA*, 39(1):147–166, January 2025.
3. Q. Cao, G. Bosilca, W. Wu, D. Zhong, A. Bouteiller, and J. Dongarra. Flexible data redistribution in a task-based runtime system. In *2020 IEEE CLUSTER*, pages 221–225, Sep. 2020.
4. T. Allen and R. Ge. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *Proceedings of Supercomputing*, SC '21. ACM, 2021.
5. G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IPDPS Workshops*. IEEE, 2011.
6. S. Atchley, C. Zimmer, J. Lange, D. Bernholdt, et al. Frontier: Exploring exascale. In *Proceedings of Supercomputing*, SC '23. ACM, 2023.
7. R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, 2017.
8. A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. PTG: An Abstraction for Unhindered Parallelism. In *WOLFHPC*, pages 21–30, 2014.
9. G. Bosilca, R. J. Harrison, T. Herault, M. M. Javanmard, P. Nookala, and E. F. Valeev. The Template Task Graph (TTG)-an Emerging Practical Dataflow Programming Paradigm for Scientific Simulation at Extreme Scale. In *International Workshop on Extreme Scale Programming Models and Middleware*. IEEE, 2020.
10. T. Herault, Y. Robert, G. Bosilca, and J. Dongarra. Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over PaRSEC. In *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 33–41. IEEE, 2019.
11. T. Herault, Y. Robert, G. Bosilca, Robert J Harrison, Cannada L. A., Edward V.F., and J. Dongarra. Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 537–546. IEEE, 2021.
12. Q. Cao, Y. Pei, T. Herault, K. Akbudak, A. Mikhalev, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra. Performance analysis of tile low-rank cholesky factorization using PaRSEC instrumentation tools. In *ProTools*, pages 25–32, 2019.
13. Q. Cao, Y. Pei, K. Akbudak, A. Mikhalev, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra. Extreme-scale task-based cholesky factorization toward climate and weather prediction applications. In *PASC*, pages 1–11, 2020.

14. Q. Cao, Y. Pei, K. Akbudak, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra. Leveraging PaRSEC runtime support to tackle challenging 3D data-sparse matrix problems. In *IPDPS*, pages 79–89. IEEE, 2021.
15. Q. Cao, S. Abdulah, R. Alomairy, Y. Pei, P. Nag, G. Bosilca, J. Dongarra, M. G. Genton, D. Keyes, H. Ltaief, et al. Reshaping geostatistical modeling and prediction for extreme-scale environmental applications. In *Supercomputing*, 2022.
16. Q. Cao, R. Alomairy, Y. Pei, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra. A framework to exploit data sparsity in tile low-rank Cholesky factorization. In *IPDPS*, pages 414–424, 2022.
17. Q. Cao, S. Abdulah, H. Ltaief, M. G. Genton, D. Keyes, et al. Reducing data motion and energy consumption of geospatial modeling applications using automated precision conversion. In *CLUSTER*, pages 330–342, 2023.
18. H. Ltaief, R. Alomairy, Q. Cao, J. Ren, L. Slim, T. Kurth, B. Dorschner, S. Bougouffa, R. Abdelkhalak, and D. Keyes. Toward capturing genetic epistasis from multivariate genome-wide association studies using mixed-precision kernel ridge regression. In *Supercomptuing*, pages 1–12, 2024.
19. S. Abdulah, A. Baker, G. Bosilca, QinQ.glei Cao, S. Castruccio, M. Genton, D. Keyes, Z. Khalid, H. Ltaief, Y. Song, G. Stenchikov, and Y. Sun. Boosting earth system model outputs and saving petabytes in their storage using exascale climate emulators. In *Supercomputing*, pages 1–12, 2024.
20. Q. Cao, G. Bosilca, N. Losada, W. Wu, D. Zhong, and J. Dongarra. Evaluating data redistribution in PaRSEC. *IPDPS*, 33(8):1856–1872, 2021.
21. L.S. Blackford, J. Choi, A. Cleary, E.F. D'Azevedo, J.W. Demmel, I.S. Dhillon, J.J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D.W. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.
22. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Par. Comp.*, 35(1):38–53, 2009.
23. L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, 1997.
24. Chunlin L., Qianqian C., and Youlong L. Optimal data placement strategy considering capacity limitation and load balancing in geographically distributed cloud. *FGCS*, 127:142–159, 2022.
25. Q. Sun, M. Romanus, T. Jin, H. Yu, P. Bremer, S. Petruzza, S. Klasky, and M. Parashar. In-staging data placement for asynchronous coupling of task-based scientific workflows. In *ESPM2*, pages 2–9, 2016.
26. J. Klinkenberg, C. Foyer, P. Clouzet, B. Goglin, E. Jeannot, C. Terboven, and A. Kozhokanova. Phase-based data placement optimization in heterogeneous memory. In *CLUSTER*, pages 382–393, 2024.
27. J. Lifflander, P. Pebay, N. Slattengren, P. Pebay, R. Pfeiffer, J. Kotulski, and S. McGovern. A communication- and memory-aware model for load balancing tasks. `https://arxiv.org/abs/2404.16793`, 2024.
28. M. Gonthier, L. Marchal, and S. Thibault. Memory-aware scheduling of tasks sharing data on multiple GPUs with dynamic runtime systems. In *IPDPS*, pages 694–704, May 2022.
29. A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra. High-performance cholesky factorization for GPU-only execution. In *GPGPU-10*, page 42–52, 2017.
30. T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P.A. Wacrenier. Resource aggregation for task-based cholesky factorization on top of modern architectures. *Parallel Computing*, 83:73–92, 2019.
31. G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, et al. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. SC '19, 2019.

32. P. Anastasiadis, N. Papadopoulou, N. Koziris, and G. Goumas. Uncut-GEMMs: Communication-aware matrix multiplication on multi-GPU nodes. In *CLUSTER*, pages 143–154, Sep. 2024.