A Device-Side Execution Model for Multi-GPU Task Graphs

Ilyas Turimbetov
Koç University
Istanbul, Turkiye
iturimbetov18@ku.edu.tr

Mohamed Wahib
RIKEN Center for Computational Science
Tokyo, Japan
mohamed.attia@riken.jp

Didem Unat
Koç University
Istanbul, Turkiye
dunat@ku.edu.tr

Abstract

Executing task graphs on multi-GPU systems presents challenges typically managed by CPU-side runtimes, which handle memory management, track dependencies, and balance load. However, the interplay of runtime components, CPU-driven kernel initialization, and dynamic task graph construction creates significant overhead. For static graphs, recent advancements have enabled GPU-side execution, demonstrating substantial performance gains in single-GPU scenarios. However, multi-GPU execution still lags behind in both usability and performance. In particular, no GPU-side solution exists for executing task graphs on multiple nodes.

In this work, we introduce Mustard, a multi-GPU execution model that shifts execution of static task graphs entirely to the devices, drastically reducing overhead. Mustard offers a clean solution for executing CUDA graphs across multiple GPUs on multiple nodes without requiring modifications to GPU kernel code or the adoption of new runtime mechanisms or APIs. By transforming the task graph, Mustard enables precise tracking of task dependencies and load balancing directly on the GPU, eliminating the need for host CPU involvement. We evaluate our approach using generated graphs, as well as LU and Cholesky decomposition graphs. In a multi-node scenario with 64 GPUs, Mustard achieves an average 5.83× speedup over the linear algebra library SLATE. On a single node, compared to the best-performing baseline, Mustard delivers an average 1.66× speedup for LU and 1.29× for Cholesky.

CCS Concepts

• **Theory of computation** → **Parallel computing models**; • **Software and its engineering** → *Runtime environments*; *Scheduling*.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730426

ACM Reference Format:
Ilyas Turimbetov, Mohamed Wahib, and Didem Unat. 2025. A Device-Side Execution Model for Multi-GPU Task Graphs. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3721145.3730426

1 Introduction

GPUs have become integral to both the AI and HPC landscapes, gaining numerous hardware and software features that enhance their autonomy and control over program execution [12, 22]. Unlike traditional methods that depend on CPU involvement for communication, synchronization, and other operations, ongoing developments [38] aim to empower GPUs for reduced latency and improved flexibility. Notably, direct device-initiated data transfers are now feasible via the NVLink interconnect, guided by GPUDirect tools [30]. In-kernel synchronization is achievable for all participating thread blocks with the introduction of the Cooperative Groups. Additional efforts extend GPU capabilities by enabling system calls [39], networking [1, 28], and storage access [31, 32] directly from the GPU side, eliminating CPU involvement in many operations.

Despite the advancements in the GPU technology, multi-GPU programming is complex due to the need for intricate workload decomposition, scheduling, and orchestration. One commonly used way of representing the workload for multi-GPUs is *task graphs*. Vertices (i.e. nodes) of a task graph represent computations and edges indicate the dependencies between them. Typically, runtime systems [4, 9, 27] are used for scheduling of task graphs across multiple GPUs. Runtime systems schedule tasks on the CPU and then offload them to GPUs, making decisions based on factors like current workload, partitioning strategies, and performance models. Sophisticated schemes may lead to a more balanced workload distribution but can be prone to overheads from scheduling, kernel launches, inter-process communication, and other runtime system components [14, 18].

We introduce Mustard, a model for executing task graphs on multi-GPU systems, where the CPU is used solely for task graph initialization. Mustard is the first multi-node GPU-side execution model for partitioned static task graphs. It

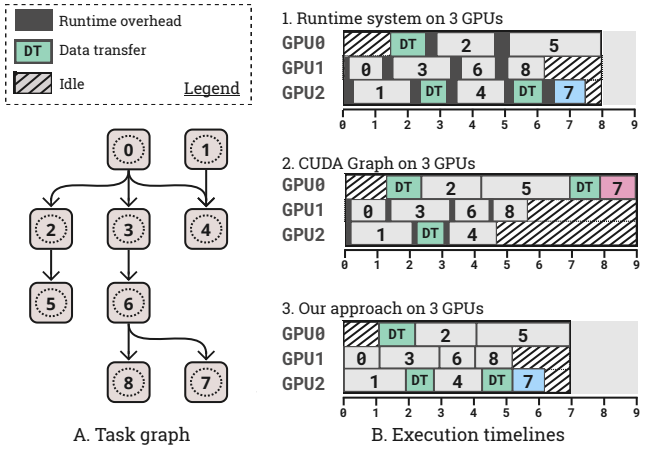

Figure 1: *Possible* execution timelines of various approaches. (1) Runtime system can use performance modeling or dynamic task allocation to allocate Task 7 on an idle GPU, but may suffer from overheads. (2) Multi-GPU CUDA Graph launch can remove the overheads, but cannot dynamically allocate tasks, resulting in Task 7 scheduled on GPU 0. (3) The proposed approach performs online scheduling with low overheads.

dynamically schedules graph vertices by tracking launched GPU resources, managing data transfers, balancing workloads across devices, and synchronously tracking task dependencies — all on the device side. Leveraging CUDA Graphs in its implementation reduces overhead while preserving ease of use for developers. Following a model introduced in this paper, a single-GPU CUDA Graph can be executed across multiple GPUs without requiring kernel code modifications.

The contributions of this work are outlined as follows:

- The first device-side multi-GPU execution model for multi-node systems.
- Implementation of low-overhead dynamic scheduling of tasks and load balancing without CPU involvement.
- Elimination of the need for a performance model, heuristics, manual intervention, or familiarity with specific runtime APIs or DSLs.
- Demonstration of LU and Cholesky decomposition as example use cases, along with integration of cuBLAS and cuSOLVER into the proposed model.
- Performance evaluation against the StarPU runtime [4], SLATE [17], single- and multi-GPU cuSOLVER implementations, and single-GPU CUDA Graph. In all cases, Mustard outperforms StarPU and SLATE while providing performance that is better or comparable to NVIDIA's cuSOLVER. In the multi-node case, Mustard reaches performance close to the hardware's theoretical peak even on smaller matrices.

Mustard demonstrates that scheduling of static task graphs in multi-GPU systems can be done on the device, and defines to what extent the runtime components could be implemented on the GPU. We argue that removing the CPU from the critical path of the task graph scheduling and execution can achieve better performance, while reducing the aforementioned overheads. Finally, even though our implementation of Mustard is based on CUDA Graphs, its concepts are directly applicable to AMD hipGraph.

2 Motivation and Related Work

This work focuses on static task graphs. Although dynamic graphs offer more flexibility by allowing changes in graph structure, most scientific, linear algebra and graph analytics applications can be expressed in form of a static graph. Unlike dynamically constructed graphs, static graphs can be set up before execution and can be reused, further reducing runtime overheads.

Current multi-GPU scheduling approaches suffer from high overhead or require manual effort to orchestrate communication and partition the task graph. Namely, managing GPUs and launching kernels from the CPU induce high overhead. To mitigate some of these overheads, NVIDIA developed *CUDA Graphs* [29] (and similarly AMD developed hipGraph [2]), which facilitates the creation and execution of task graphs on GPUs. In this model, GPU kernels, data transfers, and other tasks are represented as vertices (nodes) of the graph. Once a GPU graph is created, it can be reused, reducing kernel launch overhead. While in a single-GPU setting, task graphs can be efficiently scheduled and executed by the CUDA (or RoCM) runtime alone, multi-GPU execution is more complex. It requires the programmer to allocate tasks and orchestrate data movement between devices. Although the issue of manual orchestration has been addressed in [3], CUDA Graphs-based approaches only work within a single node.

Figure 1 shows three execution timelines of different scheduling scenarios for the same task graph. While the top timeline shows one of the possible executions by a runtime system with overheads shown in dark gray, the second timeline shows a possible CUDA Graph execution designed by a programmer (who lacks runtime knowledge of task lengths), leading to suboptimal scheduling because of placing Task 7 on GPU0. The bottom timeline shows the proposed approach that combines the low overhead of CUDA Graphs with online scheduling, which results in the best overall runtime.

Although there are existing methods for multi-GPU execution of graph algorithms in multi-node scenarios, such as those in [8, 24, 36], these approaches focus on graph applications specifically and differ from task graph execution. In graph algorithms, a single fused kernel can often suffice,

Table 1: Overview of the existing task graph scheduling approaches. ✻new tasks can be submitted, but the task types cannot be changed. ✣requires additional steps for usage: SLATE relies on OpenMP tasking, multi-GPU CUDA Graphs need a partitioner, Juggler requires the program rewritten in a persistent kernel fashion.

Approach	Multi-GPU	Multi-node	CPU-free	Kernel launch overhead	Runtime overhead	Dynamic load balancing	No need for perf. model	Dynamic graphs	Static graphs	No custom API
StarPU [4]	✓	✓	✗	high	medium	✓	✗	✓	✗	✗
PaRSEC [9]	✓	✓	✗	high	medium	✓	✓	✓	✓	✗
Legion [5]	✓	✓	✗	high	high	✓	✓	✓	✗	✗
OpenMP (SLATE [17])	✓	✓	✗	high	medium	✓	✓	✓	✗	✣
CUDA Graphs	✗	✗	✓	small	small	✗	NA	✗	✓	✓
partitioned CUDA Graphs	✓	✗	✓	small	small	✗	NA	✗	✓	✣
Juggler (Megakernel) [7]	✗	✗	✓	none	small	NA	NA	✻	✗	✣
MUSTARD (ours)	✓	✓	✓	small	small	✓	✓	✗	✓	✓

requiring only data management for multi-node execution. In contrast, task graph scheduling involves the orchestration of entire kernels.

Next, we provide background information on CPU-managed runtimes, CUDA Graphs, and other existing solutions. We discuss their pros and cons before introducing our approach. Table 1 presents a comparative overview of the solutions and their examples from literature.

2.1 CPU-managed Runtimes

The existing approaches to directed acyclic graph (DAG) execution on GPUs usually rely on CPU orchestration. For dynamic scheduling of tasks they use CPU-side queues for each device, offloading tasks to GPUs upon availability and/or request. While StarPU [4] and XKaapi [18] can take advantage of a performance model, PaRSEC [9] and Legion [5] aim to leverage knowledge about the data access patterns to improve parallelism and data reuse. All of the mentioned runtimes can leverage work stealing for dynamic load balancing, although in Legion such decisions have to be triggered by the user. These CPU-managed runtimes are capable of employing continuous dynamic scheduling, leading to a balanced allocation and reduced time-to-solution as shown in Figure 1B-1. However, in addition to the programming burden of learning new APIs, the overhead of kernel launch and other runtime components can be high. As previously shown [21], StarPU is very efficient in terms of scheduling both parallel and serial task chains on the CPU. Despite this fact, Section 5.1.2 shows that on the GPU it can be up to 13x slower compared to CUDA Graphs.

Moreover, for balanced task allocation runtime systems may rely on performance models or heuristics considering metrics such as the potential running time of each task or size of the accessed data tiles, the current computational load of devices, communication overhead, or a combination of these factors. Additionally, the mentioned runtimes rely on dynamic graph construction with the exception of PaRSEC, which allows for both static and dynamic graph construction.

2.2 CUDA Graphs

Starting from CUDA 10, a new task graph execution model named *CUDA Graphs* was made available, allowing for low-overhead *static* scheduling of task graphs. This model enables pre-scheduling of kernels and formulating dependencies between them in form of a task graph, which can be executed entirely on the GPU without engaging the host. In a single-GPU setting scheduling of the graph's kernels, as well as the individual blocks and warps is done automatically. In a multi-GPU setting, however, load balancing and management of inter-GPU communications between multiple devices remains contingent on the programmer's decisions *and is only possible within a single node*. For a CUDA Graph to be run on a multi-GPU system, when the graph is being constructed, the `cudaSetDevice` function should be called to assign a newly added vertex in the graph to its intended device. Additionally, `cudaMemcpy` vertices need to be manually inserted to perform device-to-device data transfers when needed to satisfy inter-device data dependencies.

As shown in Figure 1B with the middle timeline, the overhead of dependency tracking and kernel launches in this scenario is minimized thanks to CUDA Graphs, but multi-GPU execution is limited to a single node and requires additional static task-to-device allocation and data transfers. As Figure 1B depicts with Task 7 in pink scheduled on GPU 0, in such cases optimal scheduling may be difficult to achieve, since the execution time of graph nodes is unclear in advance and there is no runtime inter-GPU load balancing mechanism in CUDA Graphs. If Task 7 could be scheduled on GPU 2 instead, an optimal low-overhead execution timeline can be achieved.

2.3 Persistent Megakernels

Previous research on CPU-free task graph execution [7] uses *persistent megakernels* to address the limitation of GPUs not being able to launch kernels. The approach involves launching a single large persistent kernel across all available SMs, containing code required for each task type that can be executed, split by the conditional operators. The GPU then uses warps or thread blocks as workers, distributing tasks through mechanisms like queues. While some applications benefit from this method [6] and efficient caching techniques [41], more complex applications with diverse tasks, dependencies, and multiple GPUs still rely on CPU-side scheduling [11]. Some examples of the megakernel approach include Softshell [33], Whippletree [34], and Juggler [7]. However, to our knowledge, no multi-GPU megakernel implementation exists. In addition, they inherited limitations of persistent kernels, such as high register pressure [12], thread divergence [35], and lack of hardware scheduler support [19].

3 Mustard

Mustard aims to address the current limitations of multi-GPU scheduling of static graphs by introducing an inter-node execution mechanism and demonstrate the viability of moving runtime components to the device. We suggest device-side solutions for dependency tracking, synchronization and data transfers, elaborated in Section 3.1. While the scope of this work involves only static graphs, we propose mechanisms for dynamic scheduling of static graphs in Section 3.2, which involves runtime decision making for work distribution, paving the way for fully GPU-side runtime systems.

3.1 Task Graph Execution with Static Scheduling

In a static scheduling scenario, partitioning of the graph between GPUs is known in advance. This scenario does not require full-fledged runtime systems with dynamic task placement and load balancing features. The main challenge of static scheduling is dependency tracking. Communication has to be managed too, but it is less challenging, since in the static scenario the participating devices for data transfers are known in advance. While both the dependency tracking and communication can be handled by CUDA Graphs in a single-node case, or can be managed by the CPU, such solutions are prone to overheads mentioned in Section 2. Additionally, Mustard helps reducing memory footprint of CUDA Graphs achieved by splitting the task graph between the devices.

The execution of CUDA Graphs across multiple nodes introduces additional complexities. While CUDA Graphs allow manual allocation of graph vertices, this is limited to devices within the same host. To overcome these limitations and reduce overhead, we propose a device-side method that allows for multi-GPU dependency tracking, and is applicable to multi-node systems.

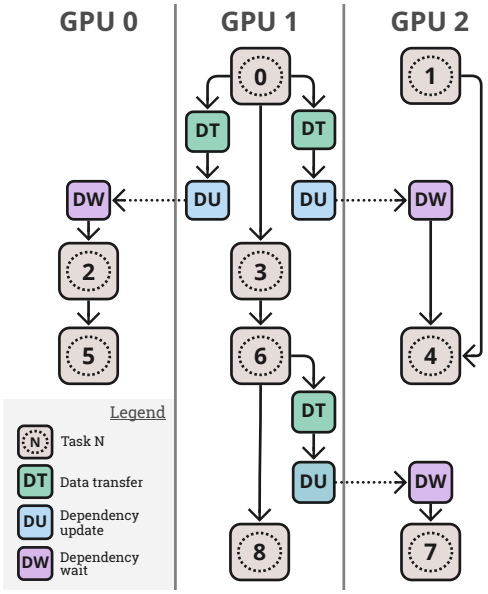

Figure 2: Enriched static graph: dependency tracking and data transfers on the GPU side in a `cudaGraph` format. The graph and resulting timeline follow Figure. 1

3.1.1 *Device-Side Synchronization and Graph Enrichment.*

Dependency updates and data transfers require a synchronized effort in multi-GPU systems. Consider, for instance, a graph vertex with a compute kernel spanning all thread blocks. To update the dependencies upon its completion, it is crucial to ensure that all threads in all blocks have concluded the kernel execution. However, kernel-level synchronization is impossible without CUDA Cooperative Groups, requiring `cudaLaunchCooperativeKernel` wrapper for kernel launch, with a limitation of the maximum assignable thread count. Calling `cudaDeviceSynchronize()` is not an option either, since it must be invoked on the host side. While static scheduling requires GPU-wide synchronization, dynamic runtime decisions rely on synchronization even more, since it is also required for occupancy tracking, distribution and scheduling of tasks, as will be discussed in Section 3.2.

To tackle the challenges mentioned above, Mustard uses a concept we refer to as *graph enrichment*. The idea behind graph enrichment is to incorporate all the functionality required for graph scheduling into the graph itself. This way, dependency updates, occupancy tracking and data transfers can occur within the added vertices of the enriched graph. Figure 2 shows an example of an enriched version of the graph presented in Figure 1.

3.1.2 Dependency Tracking. While the local dependencies can be tracked efficiently by CUDA Graphs, global dependencies are either prone to overhead or, in case of multi-node execution, nonviable. With the graph enrichment method at hand, dependency tracking can be done in separate vertices of the graph. This way, whenever there is a remote dependency, i.e. a child vertex is on a different GPU, it can be decremented by inserting a new vertex between the parent and a child. Since the child will be executed on a remote GPU, it needs to be able to poll from the same memory address that is being decremented by the parent. To ensure proper ordering of decrements, atomic memory updates are required. For atomic updates to span between compute nodes, we leverage NVSHMEM atomics.

Figure 2 shows the injection of dependency update vertices labeled as DU, which have a matching dependency wait (DW) vertex on a remote GPU. Note that even if there may be multiple incoming DU vertices, only one DW vertex is needed on the receiver side.

3.1.3 Data Transfers for Static Scheduling. The source and destination devices are known for all the data transfers when the workload is statically scheduled. As such, using the source and destination IDs, device-side communication can be issued whenever there is a data dependency between tasks allocated to different GPUs. The data transfers can then be added to the graph as DT vertices, as per Figure 2. While, within a node, Mustard uses `cudaMemcpy` calls, in a multi-node case, it uses NVSHMEM instead to perform the data transfers.

3.1.4 Example with a Multi-node LU Decomposition with CUDA Graphs. To better illustrate the approach, Figure 3 provides a simple example. It shows a block LU decomposition graph with 3x3 tiles on a system with 2 GPUs, each on a separate compute node. The graph is partitioned according to the round-robin columnwise data distribution scheme, same partitioning scheme that is used by cuSolverMg and by Mustard in Section 5. The leftmost part shows the first iteration of the algorithm, with the second iteration behind on the data distribution scheme in muted colors. The middle and right parts depict the graph partitions created by Mustard for GPU0 and GPU1, correspondingly.

All the local dependencies in the graph are handled by CUDA Graphs, while internode dependencies are tracked by Mustard. Whenever the destination of an outgoing dependency belongs to another GPU, a dependency update (*DU*) vertex is inserted as shown in the middle part of the figure with GPU1. Meanwhile, the incoming dependencies from peer GPUs are preceded with a dependency wait (*DW*) vertex as shown with the right part of the figure with GPU0. The *DU* vertex contains an atomic NVSHMEM update, which is caught inside a busy-waiting *DW* vertex. Each *DW* only occupies a single GPU thread, ensuring conservation of compute resources.

Algorithm 1 Graph Enrichment for Dynamic Scheduling

```
1: function MUSTARD(Graph g)
2:     S = GETSUBGRAPHS(g)                          ▷ partition into subgraphs S
3:     D = ADDDEPENDENCIES(S)                       ▷ creates dependency array D
4:     ADDCOMMUNICATIONVERTICES(S)                  ▷ handles data transfers
5:     O = ADDOCCUPANCYUPDATES(S)                   ▷ O stands for current load info
6:     q = INITIALIZEQUEUE(D, device)
7:     for device = 0 to GPU_COUNT do
8:         INITIALIZESUBGRAPHS(S, device)
9:         SCHEDULER(q, S, O, device)               ▷ runs on the GPU
10:    end for
11: end function
```

3.2 Task Graph Execution with Dynamic Scheduling

Apart from multi-node execution of static graphs, our multi-GPU execution model attempts to address the limitations of intra-node scheduling. Particularly, it has been impossible to dynamically schedule work on a multi-GPU node without CPU involvement. Mustard proposes a model for device-side task distribution with load balancing and minimized overheads. The runtime overheads of queuing, dependency tracking and kernel launch are reduced by merging the vertices into subgraphs, letting the hardware scheduler to handle the intra-device scheduling of the subgraph vertices. While the graph remains static, decisions regarding task distribution within the system are dynamically made based on the current resource usage. This approach results in a more informed runtime that facilitates load balancing across devices.

To our knowledge, apart from the methods mentioned in Table 1, the only work attempting to move the runtime functionality completely to the device side that by Choi et al. [13]. However, it is incapable of scheduling various tasks required by the execution of DAGs and does not maintain resource usage information. To have a more complete runtime functionality for running multi-GPU task graphs entirely on the device side these challenges must be addressed: (i) Enabling device-side kernel launches, (ii) Achieving device-side synchronization, (iii) Tracking task dependencies to ensure correct execution, (iv) Managing memory and data transfers between devices, (v) Monitoring device occupancy for load balancing, and (vi) Efficiently distributing tasks among devices. In this section, we will explain how Mustard tackles these challenges and outline the design of its components.

3.2.1 Graph Enrichment for Dynamic Scheduling. As discussed in Section 3.1.1 for synchronization we employ *task graph enrichment*, which uses the implicit barriers between

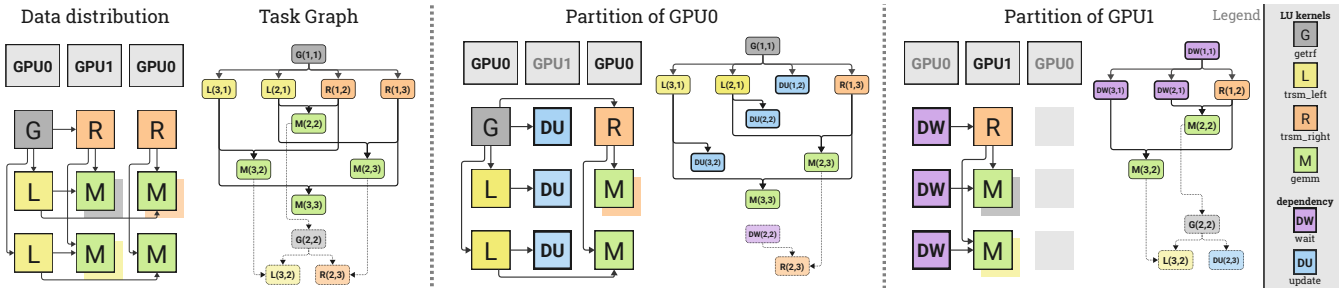

Figure 3: Example of partitioned LU decomposition graph with 3x3 tiling and 2 GPUs on 2 separate compute nodes. Round-robin column-wise partitioning of data is used. Additional vertices are inserted into the graph for inter-node dependency resolution, making it feasible for multi-node execution.

vertices and adds functionality required for task graph execution into the graph itself. Before scheduling the task graph, the initialization phase creates the enriched graph through a series of steps as shown in Algorithm 1. Line 2 identifies the serial or tightly connected sections of the graph by partitioning it into subgraphs. Line 3 initializes the outgoing dependency counts for each subgraph and adds dependency update vertices whenever there is an outgoing dependency to a task in another subgraph (Section 3.1.2). The data pointers are also recorded inside these vertices. Data is read within communication vertices created on Line 4, enabling inter-device communication (Section 3.2.3). Line 5 adds vertices to the graph to update occupancy information on each device (Section 3.2.4). Line 6 initializes the task queue (Section 3.2.5), and finally, Line 8 initializes all subgraphs on each device. The enriched graph can then be executed by a simple GPU-side scheduler, as discussed in Section 3.2.6.

Figure 4 shows an example of enriched task graph, which has additional vertices inserted into the task graph from Figure 1. The vertices updating dependencies, data transfers and tracking device occupancy are added. Moreover, tasks are merged into separate schedulable units called *subgraphs*, labeled from A to E.

3.2.2 Device-side Kernel Launch. The main obstacle for GPU-side task scheduling is the lack of kernel launch capability on the device. Existing methods of device-side kernel initiation do not exhibit the performance and flexibility of CPU-side kernel launch. For instance, a CUDA feature called *dynamic parallelism* allows the programmer to split a portion of the parent kernel's resources to execute a child kernel. However this method suffers from multiple sources of overhead [10], proved to be suitable only for applications that follow the divide-and-conquer paradigm [23] and was later deprecated, making its way back only with CUDA version 12. Another potential solution could be avoiding kernel launches completely by adopting the persistent kernel model [22] and following the megakernel approach. This would leave the resource allocation and scheduling decisions between blocks and threads

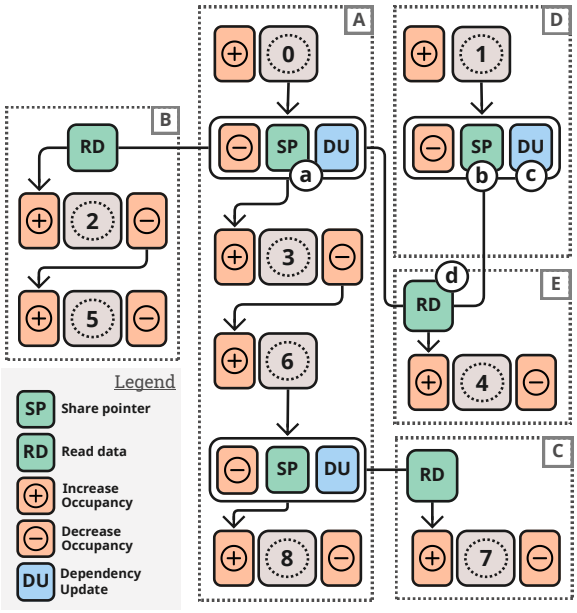

Figure 4: Enriched graph: dependency and occupancy tracking on the GPU side in a `cudaGraph` format.

inside each device to be managed by the runtime, and would forbid the use of a hardware scheduler. Maintaining the inter-device runtime components as well would exacerbate the register pressure issue of persistent (mega)kernels.

Our approach uses the *device-side CUDA Graph launch* feature as a solution for launching device-side kernels and online scheduling of kernels. This feature allows one to launch a previously initialized `cudaGraph` from an already running graph node (vertex). We can split graph vertices into separate `cudaGraphs` and schedule them from a long-running device-side kernel, minimizing the kernel launch overheads.

However, device-side graph launch comes with a limitation, which requires the number of graphs submitted to a device to be no more than 120. In cases with less than 120 vertices, every vertex can be considered a separate *subgraph*.

Otherwise, an acyclic partitioning such as dagP [20] is required.

After partitioning, subgraphs serve as work items to be scheduled, as shown in Figure 4. The task graph vertices shown in Figure 1 are coarsened into subgraphs labeled as A, B, C, D and E. The cut edges represent a dependency update and potential inter-device communication, if the parent and child subgraphs are placed on different devices. The internal dependencies inside a subgraph are handled by CUDA Graphs and ensure minimal overhead.

3.2.3 Data Transfers and Memory Management for Dynamic Scheduling. Unlike the execution method that partitions the graph and schedules the memory allocations and data transfers before starting the DAG execution, data transfers in dynamic scheduling scenario are more difficult. Both the memory allocation and device-to-device data transfer graph nodes in CUDA require the IDs of the participating devices, making it impossible to schedule such data transfers dynamically without reinitialization of the task graph by the host. Yet, dynamic device-side data transfers are now possible thanks to device-initiated communication [1, 15, 30]. Using a preallocated buffer, devices can share the data between them inside the compute kernel graph nodes. This approach, however, poses an additional challenge of buffer management.

Memory allocation in the dynamic setting can be addressed in the same way by adding `__device__ cudaMalloc` calls into compute kernels. Memory allocated in such a manner, however, can not be shared with the peer devices, making it unsuitable for multi-GPU execution. Moreover, memory allocation nodes can not be launched using the device-side graph launch feature. As a solution, *Ouroboros* tool [40] provides low-overhead device-side memory allocation. Due to the fact that it uses `cudaMalloc` API to reserve all the available memory in the initialization stage, the memory pointers created by Ouroboros can be shared with other devices, making it a good fit for the multi-GPU scenario. The memory pointers allocated using Ouroboros malloc calls can be announced globally and used without the need to relaunch device kernels. For example on Figure 4, the pointers are announced in Step ⓐ and ⓑ. Since both Task 1 and Task 4 are executed by GPU 2, it already has access to the data generated by Task 1. However, Task 0 is executed by GPU 1, so GPU 2 has to read the data ⓓ from a remote address that has been announced in step ⓐ.

3.2.4 Occupancy Tracking and Load Balancing. Another requirement for dynamic multi-GPU execution is to maintain the resource availability information of the devices, which allows to make load balancing decisions between the devices. Traditionally, such information is being kept track of on the CPU side. The host can keep track of the memory allocations, the amount of launched threads and blocks, and other more detailed resource usage data.

Mustard aims to maintain information about the amount of memory being allocated and the number of CUDA threads and blocks in use. Memory usage can be kept track of inside the Ouroboros `memcpy` kernels. To keep track of the threads and blocks launched, a single thread has to atomically update a corresponding value. Since these updates need to be done just before a compute kernel is launched and decremented again after it finishes, we use the same method as the one employed for dependency tracking: adding single-thread kernels to the subgraphs. As a result, each kernel is enclosed by two dependency update kernels, represented by vertices with increment and decrement signs, as illustrated in Figure 4. In order to reduce the overhead of multiple additional kernels, occupancy and dependency update kernels are fused together.

When the occupancy information is available, it allows us to maintain load balancing between the devices. Since there are no locality mechanisms involved and there is a single task pool, maintaining the occupancy of each device below the maximum device occupancy (108 blocks of 1024 threads each for NVIDIA A100, for example) is sufficient.

3.2.5 Task Distribution. For distribution of tasks between the GPUs in the system we employ a device-side queue. The task ID is retrieved from the queue and the pointers for the required task data are obtained, so only two variables need to be transferred. All the other data is broadcasted to each device in advance. The tasks need to be synchronously enqueued and dequeued to ensure consistency. While efficient single-GPU parallel queues have been around for a while [26, 35, 37], queueing for multi-GPU systems has been done through the CPU, with an option of multiple threads managing queues for every device [4, 5, 9]. This limitation comes mostly from the fact that such queues usually rely on CUDA atomic memory reads and writes, which are consistent only within a device's own memory. Only recently, NVSHMEM bridged the gap by introducing multi-GPU atomics. The queue still has to reside on a single device, so that there is a point of synchronization between the GPUs. Alternatively, the queue can be partitioned between the devices, but the total number of nodes in a graph is typically not large enough to require it and the limitation of the number of device-side graph node launches makes this even less of a concern.

We reimplemented the Broker queue [26] with NVSHMEM atomics, introducing GPU-side queues that are consistent even in multi-GPU systems. Such queues allow us to share the workload between the devices in a fast manner without the CPU involvement. In Mustard, queue updates happen in dependency update kernels, shown in Figure 4 ⓒ. When the atomic decrement returns 0, the task is enqueued. The

Algorithm 2 GPU-side Scheduler (CUDA kernel)

```
1: function SCHEDULER(
   Queue q, Subgraph[] S, Occupancy[] O, int deviceID)
2:     while q.itemsDequeued != S.size do          ▷ while tasks left
3:         if !O[deviceID].isBusy() then           ▷ if this GPU is not busy
4:             int sID = q.dequeue()               ▷ claim a task
5:             cudaGraphLaunch((S[sID])            ▷ device-side launch
6:         end if
7:     end while
8: end function
```

dependency updates are done similarly to static approach in Section 3.1, but with no need to use busy-waiting vertices. Instead, a busy-waiting scheduler launches tasks from a shared queue.

3.2.6 Scheduler. Algorithm 2 shows the simple GPU-side scheduler employed by Mustard. It takes the task queue, subgraphs, occupancy and device ID as input and claims available jobs from the queue and executes them using device-side CUDA Graph launch. To better illustrate how the scheduler works, let us consider the example of the task graph from Figure 1, assuming the initialization is completed and subgraphs are formed as depicted on Figure 4. Subgraph vertices will be executed with no need for runtime control, as they are fully managed by CUDA Graphs. With the dependency update vertices inserted, whenever a task is completed it will automatically decrement the dependency value of the child subgraph and enqueue it if the value is 0. This way, after Task 0 is completed, it enqueues subgraph B, and decreases the dependency value of subgraph E. Later, when Task 1 finishes, it further decrements the dependency value of E to 0 and enqueues it. This shows how the granularity of dependency tracking is based on individual tasks rather then entire subgraph: subgraph B does not have to wait for completion of subgraph A, and becomes ready upon completion of Task 0.

4 Limitations

Although Mustard resolves the most challenges a device-side runtime poses, there are certain limitations remaining due to hardware or CUDA limitations. The most important and restraining one is the 120-subgraph limitation of device-side launch. While it can be overcome by merging serial or highly interdependent vertices, the parallelism of the merged tasks may reduce. Moreover, this limitation severely hinders scaling, especially to multi-node execution scenarios.

There is also a memory-related set of limitations. `cudaGraph`, when scaled to millions of vertices, can occupy the entire memory of a device. This, however, becomes an issue only if the previous limitation is resolved. Apart from that, maintaining the ownership of data and migrating it between the devices, which may require awareness of the system topology and prediction of the future execution schedule is not handled by Mustard. Instead, we focus on scenarios where the data can be replicated on all devices or can be partitioned between them.

Additionally, while occupancy tracking is able to handle memory usage and number of threads and thread blocks in use, more detailed metrics, such as register or cache usage are inaccessible from the device side. Additionally, libraries such as cuSOLVER and cuBLAS choose thread block and thread count automatically, limiting the efficiency of occupancy tracking.

In Mustard, multi-node execution is only possible *without dynamic load balancing*. The intended use case scenarios of Mustard cover 2 main limitations of current GPU-side task graph execution methods: (i) *Multi-node execution* for statically partitioned graphs and (ii) *Dynamic load balancing* for applications whose data fits into a single GPU's memory.

5 Evaluation

Experiments were conducted on a machine with multiple nodes, each containing 8 NVIDIA A100 GPUs connected to a dual-socket AMD EPYC 7742 CPU through PCIe Gen4. Each socket has 64 physical cores, and hyperthreading was disabled. The GPUs are connected through NVLink 3.0 interconnect. Measurements are taken five times and averages are reported. We compare Mustard against four related works:

- **StarPU** [4] is a well-known CPU-side runtime system
- **SLATE** [17], is a dense linear algebra library targeting large-scale HPC systems
- **Single- and multi-GPU cuSOLVER** implementations by developed and maintained by NVIDIA
- single-GPU CUDA Graph implementation.

Next, we compare the overhead of Mustard on generated graphs and then its performance on real task graphs using LU and Cholesky solvers.

5.1 Overhead Studies

We analyze the sources of overhead of Mustard, in static and dynamic scheduling configurations. To assess partitioning efficiency and its associated overheads, we conduct graph partitioning experiments. Additionally, we test generated graphs with varying computational intensity to demonstrate Mustard's efficiency across different scenarios.

5.1.1 Static scheduling overhead. We use a graph generator as a controlled experiment where varying parameters such as number of vertices or average degree help to understand sources of overhead. Table 2 shows static partitioning overhead of Mustard compared to multi-GPU CUDA Graph on a

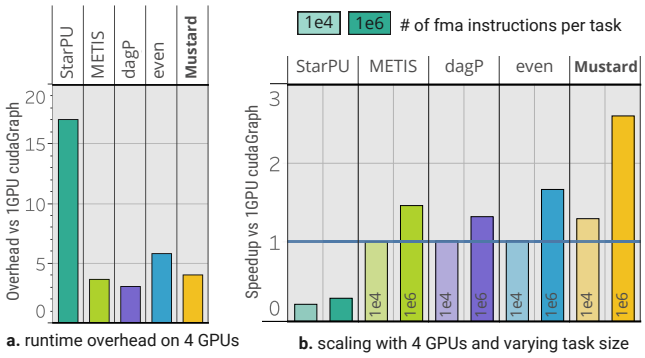

Figure 5: (a) Overhead against single-GPU `cudaGraph` (lower is better). (b) Speedup of various approaches against `cudaGraph` on graphs with increasing task length, given by number of FMA instructions performed. *METIS, dagP* and *even* use static partitioning, with *even* referring to evenly split DAG levels between GPUs. (Higher is better)

Table 2: Overhead comparison of static graph execution. The values show the time it takes to complete execution of a randomly generated task graph with empty vertices, in ms. Improvement shows the reduction in overhead of Mustard over multi-GPU CUDA Graph.

	number of vertices			
method	2^8	2^{10}	2^{12}	2^{14}
Single-GPU CUDA Graph	0.204	0.877	3.703	18.639
Multi-GPU CUDA Graph	25.026	57.214	145.966	453.530
Mustard	**0.501**	**2.641**	**17.273**	**269.057**
Improvement	50.48x	21.63x	8.45x	1.69x

4-GPU system. Both methods use the same partitioning with evenly split graph levels. Mustard exhibits significantly lower overheads, especially in smaller graphs. The diminishing improvement with increasing graph size is largely due to the increased number of dependency wait vertices occupying a high percentage of warps.

5.1.2 Runtime overhead. In this experiment we use the same graph generator to analyze the runtime overheads of dynamic scheduling. Overhead results for graphs with multiple dependencies can be seen on Figure 5a. Methods shown on the figure refer to statically partitioned CUDA Graph executions on 4GPUs with various partitioning methods (METIS [25], dagP (directed acyclic graph partitioner) [20], and *even*, evenly split DAG levels between GPUs. Note that SLATE [17] is not on the figure since it is a linear algebra library and it does not allow custom tasks.

Figure 5a shows pure overhead of scheduling random graphs with no computation with different dependency patterns on 4 GPUs compared to baseline, which is single-GPU `cudaGraph` execution. On a 4 GPU system, StarPU shows 16.95x slowdown, while Mustard has 6.17x, making it 4.25x faster than StarPU. The figure shows that dynamic allocation of tasks in Mustard has comparable overhead to statically partitioned approaches, which do not employ runtime decision-making mechanisms. The overhead result also shows that the enriched graph approach in tool provides a viable, low overhead design for device-side synchronization and subsequent implementation of the runtime components on the device.

5.1.3 Varying task length. Most irregular applications can be represented in a form of a directed acyclic graph. However, in case of a CUDA Graphs, every vertex may represent a computation of arbitrary complexity. As such, even a single vertex can be an irregular application of its own. Since Mustard's scope is orchestration of tasks that is agnostic of the tasks themselves, Mustard does not improve the performance of individual kernels. Yet, it is able to reduce the load imbalance between devices that may arise from variable runtime of sparse kernels without any modifications to the approach being used. Figure 5b shows that Mustard is in fact able to load balance compute kernels of random compute intensity better than partitioning algorithms.

To better understand the efficiency of load balancing we have conducted experiments with randomized compute load using FMA (Fused-Multipy-Add) instructions. On Figure 5b the time-to-solution of the graph execution methods under increasing compute load is given. The reported speedups are average across 7 different generated graphs with varying amount of parallelism, vertex degrees and topologies. All graphs have 120 vertices, which allows to have each separate task to be represented by a separate schedulable subgraph. The reported numbers show the speedup against single-GPU `cudaGraph` implementation.

Methods that are based on the use of static partitioning of the `cudaGraph` start seeing the benefit of multi-GPU execution only when the tasks become sufficiently long, reaching up to 1,000,000 FMA instructions per task. StarPU has been unable to yield any speedup and displayed at least 3x slowdown compared to the single-GPU baseline. In a 4-GPU scenario with 10,000 maximum possible FMA instruction per task Mustard is able to show scaling, unlike the other methods, due to lower overhead and better load balancing.

5.2 Real Task Graphs: LU and Cholesky

To showcase Mustard's performance, we implemented dense LU and Cholesky decompositions. The applications are implemented using NVIDIA's cuSOLVER, and cuBLAS

Table 3: Experiment results for LU Decomposition. The numbers show speedup over single-GPU `cuSolverGetrf` call. The tile sizes range from $2,000\times2,000$ to $10,000\times10,000$. NVIDIA's *cuSmGF* is a fine-grained tiling `cuSolverMgGetrf` call with 120^2 many tiles, with tile size from 100×100 to 500×500. Best performance is shown in bold.

	2GPU				4GPU				8GPU			
size	*Mustard*	cuSmGF	SLATE	StarPU	*Mustard*	cuSmGF	SLATE	StarPU	*Mustard*	cuSmGF	SLATE	StarPU
12000	**1.659**	1.217	0.077	0.009	**2.733**	1.247	0.111	0.234	**5.216**	1.191	0.102	0.222
24000	**1.546**	1.099	0.055	0.224	**2.560**	1.520	0.052	0.223	**3.163**	1.540	0.049	0.210
36000	**1.724**	1.365	0.058	0.270	**2.879**	2.217	0.057	0.272	**3.658**	2.738	0.055	0.267
48000	**1.707**	1.563	0.053	0.267	**2.925**	2.583	0.051	0.274	**4.082**	3.401	0.048	0.275
60000	**1.720**	1.598	0.040	0.272	**2.999**	2.732	0.050	0.277	**4.164**	3.856	0.049	0.278

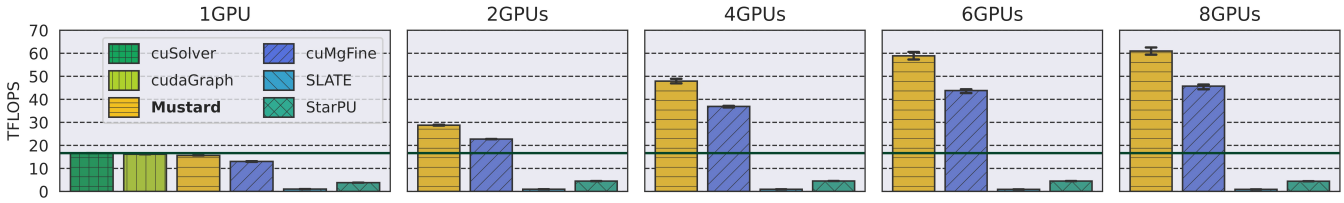

Figure 6: LU performance results compared to related work on up to 8 GPUs on a 36000x36000 matrix. Higher is better. Horizontal solid line is the baseline, a single-GPU cuSolver library call.

libraries. For a fair comparison, all the baseline methods rely on these libraries too, thus the resulting performances are mostly due to the runtime components only, not the kernel implementations.

The LU decomposition is a factorization of a square matrix A into a lower triangular matrix L and an upper triangular matrix U, such that $L*U = A$. Cholesky, on the other hand, is a special case where the input matrix is *symmetric*, resulting in a decomposition where U is the transpose of L, such that $L * L^T = A$. It is used for solving linear systems efficiently using forward or backward substitution. The *block* version [16] of the LU and Cholesky decompositions was designed for efficient parallelized execution.

Block LU decomposition can be represented in form of a task graph with 4 task types: (1) a GETRF kernel that performs LU decomposition of an individual matrix block, (2) lower and (3) upper TRSM (triangular solve) kernels that solve off-diagonal blocks upon completion of GETRF, and (4) GEMM kernel that multiplies the resulting off-diagonal blocks. An example task graph for 3x3 block configuration is given in Figure 3.

Block Cholesky decomposition's DAG has 4 task types as well: (1) a POTRF kernel performs Cholesky decomposition of a block, (2) lower TRSM kernel that solves the POTRF result, (3) SYRK kernel that updates the upper part of the matrix to be the transpose of the lower part, and (4) GEMM kernel that multiplies the resulting off-diagonal blocks.

In a single-node scenario, each task is captured into a subgraph, which may contain multiple vertices based on the cuSOLVER or cuBLAS libraries' decisions. Because of device-side launch limitations the subgraph cannot contain `cudaMalloc` kernels, so workspace for both cuSOLVER and cuBLAS has to be allocated in advance. Since cuBLAS tasks can be scheduled on a device simultaneously, multiple cuBLAS workspaces are allocated. Due to device-side launch limitations, GETRF tasks with block dimensions higher than 4096 are impossible to launch using Mustard, since the captured graph of the cuSOLVER kernel contains `cudaMalloc` nodes regardless of the pre-allocated workspace. To alleviate this issue we recursively create an LU decomposition cudaGraph with smaller block dimensions for such large GETRF tasks. Cholesky decomposition's POTRF kernel does not insert `cudaMalloc` nodes even for large block sizes. Because of the dynamic task allocation, the experiments are conducted with matrix sizes that can be allocated in a single GPU's memory, as there is no a priori task allocation and no guarantee that more than a single GPU will be working.

The implementation of LU decomposition for the multi-node scenario follows the methodology described in Section 3.1.4. The graph is partitioned between the devices and inter-GPU dependencies are handled by Mustard. Additionally, the method uses a circular buffer for the communicated data and the data races are avoided by introduction of `cudaGraph` dependencies between the vertices accessing the same buffer regions.

5.3 Single-Node Results of LU and Cholesky with Dynamic Scheduling

We compare our implementation against StarPU [4] runtime and SLATE [17], as well as single- and multi-GPU cuSOLVER implementations and single-GPU CUDA Graph.

Table 4: Experiment results for Cholesky Decomposition. The numbers show speedup over single-GPU `cuSolverPotrf` call. The tile sizes range from $1,500\times1,500$ to $7,500\times7,500$. NVIDIA's *cuSmGF* is a fine-grained tiling `cuSolverMgPotrf` call with 120^2 many tiles, with tile size from 100×100 to 500×500. Best performance is shown in bold.

	2 GPUs					4 GPUs					8 GPUs				
size	*Mustard*	cuSmG	cuSmGF	Slate	StarPU	*Mustard*	cuSmG	cuSmGF	SLATE	StarPU	*Mustard*	cuSmG	cuSmGF	SLATE	StarPU
12K	**1.153**	0.932	0.188	0.161	0.099	**1.515**	0.834	0.177	0.111	0.083	**1.839**	0.729	0.163	0.399	0.061
24K	**1.378**	1.166	0.819	0.211	0.252	**1.857**	1.195	0.934	0.176	0.235	**1.978**	1.046	0.766	0.678	0.211
36K	**1.399**	1.227	1.198	0.302	0.318	**2.013**	1.425	1.829	0.256	0.323	**2.288**	1.308	1.906	1.008	0.303
48K	**1.471**	1.249	1.412	0.373	0.357	2.164	1.477	**2.286**	0.330	0.360	2.449	1.406	**2.720**	1.366	0.356
60K	**1.525**	1.252	1.491	0.412	0.366	2.258	1.487	**2.500**	0.362	0.388	2.728	1.463	**3.185**	1.657	0.386

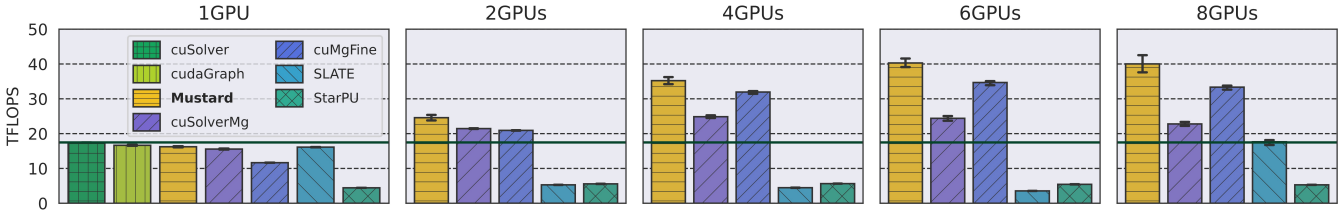

Figure 7: Cholesky performance results compared to related work on up to 8 GPUs on a 36000x36000 matrix. Higher is better. Horizontal solid line is the baseline, a single-GPU cuSolver library call.

While SLATE is not a runtime, it relies on OpenMP tasking for scheduling of linear solvers and other linear algebra applications. The results are reported for 6^2 tiles setup for LU and 8^2 tiles for Cholesky, since it allows to avoid task graph partitioning. This tiling allows to keep the graph structure the same regardless of matrix size. The tile sizes therefore, range from $2,000\times2,000$ to $10,000\times10,000$ in LU and $1,500\times1,500$ to $7,500\times7,500$ in Cholesky.

Most implementations benefit from bigger tile sizes and therefore lower task granularity, since the underlying BLAS libraries such as cuBLAS yield best performance with large tile sizes in order of thousands. `cuSolverMg` calls, however, benefit from fine-grained tiling due to its multi-GPU matrix partitioning algorithm. It allows to expose parallelism and efficiently balance the load between devices. To showcase the effect of fine-grained tiling and for fair comparison to the state of the art we report cuSOLVER results with 120^2 tiles as well, labeled as *cuSmGF*, with tile sizes ranging from 100×100 to 500×500.

The results are reported in Tables 3-4 and Figures 6- 7 for LU and Cholesky, respectively. The reported numbers in the figures show TFLOPS achieved (higher is better). There is no *cuSolverMg* data because the `cuSolverMgGetrf` kernel terminated with the large tile size setting.

Overhead. Single-GPU results display the comparison and overhead of the existing methods. Mustard shows little slowdown even with small matrix sizes, which consists of the overhead of `cudaGraph` together with the GPU-side runtime components. Additionally, Figure 7 shows how fine-grained tiling of *cuSmGF* `cuSolverMgPotrf` (with 120^2 many tiles) introduces additional overhead compared to coarse-grained *cuSolverMg* version.

Scaling. Mustard displays good strong scaling up to 6 GPUs. Due to the lack of locality awareness, communication between devices start to restrict the potential performance improvements with higher GPU counts, meanwhile the diminishing inter-GPU parallelism exposed by the reducing number of schedulable tasks in tiled matrix factorization algorithms exacerbates the scaling issue. Despite this, Mustard manages to achieve better or similar 8 GPU performance to proprietary NVIDIA cuSOLVER library. Moreover, with smaller matrices Mustard exhibits better scaling than all of the tested methods. It is able to provide best time-to-solution for both algorithms with matrix size up to 36000. With smaller number of GPUs, Mustard performs well too, winning in all of the tested runs.

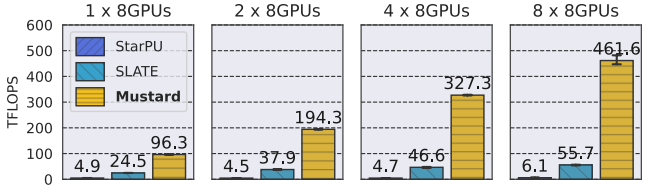

Figure 8: LU performance results compared to related work on up to 8 nodes on a 120,000x120,000 matrix.

Table 5: Experiment results for multi-node (partitioned) LU Decomposition. The numbers show theoretical TFLOPS achieved. All methods use 120^2 many tiles, with tile sizes ranging from 1000×1000 to 3000×3000. Every node has 8 NVIDIA A100 GPUs.

	Matrix sizes							
	120K				240K			360K
method	*1 node*	*2 nodes*	*4 nodes*	*8 nodes*	*2 nodes*	*4 nodes*	*8 nodes*	*8 nodes*
StarPU	4.886	4.545	4.667	6.136	39.00	35.17	36.778	116.31
SLATE	24.527	37.942	46.588	55.717	72.832	89.62	110.35	126.97
Mustard	**96.287**	**194.25**	**327.28**	**461.62**	**205.06**	**332.73**	**535.53**	**551.98**

5.4 Multi-Node Results of LU and Cholesky with Static Scheduling

In a multi-node scenario with up to 8 nodes Mustard shows highest performance compared to existing methods in all test cases. All example use the same tile sizes for fair comparison. Since cuSOLVER calls do not support multi-node execution, only SLATE and StarPU were compared against. Unlike in the single-node case, SLATE shows good scaling, as it is aimed at HPC environments.

Figure 8 displays that both SLATE and Mustard are able to scale with the increasing node count, while Mustard also achieves highest FLOP rates. Table 5 shows results for three different matrix sizes. Note that it does not show all node count combinations due to memory limitations. With larger matrices, performance of Mustard shows a marginal increase in performance. However, despite the increased performance of other methods with larger matrices, the relative difference compared to Mustard and the reported FP64 performance of NVIDIA A100 GPU indicates that it is able to achieve FLOP rates close to the theoretical peak even when computational costs are small. As a result, the speedup of Mustard ranges between 2.81x and 8.28x, and between 4.75x and 75.2x compared to SLATE and StarPU, respectively.

6 Conclusion

Mustard is designed for executing task graphs on multi-GPU systems, prioritizing load balancing and communication management without relying on host CPU control. Once provided with a task graph, Mustard undertakes several steps to modify and execute a highly complex graph on a multi-GPU system without changing the CUDA kernels. On a single GPU with synthetic graphs, Mustard's overhead is reasonable. In single-node settings, the experimental results with LU and Cholesky decompositions demonstrate that Mustard's load balancing feature can achieve much higher performance than related works, including StarPU, SLATE, as well as multi-GPU cuSOLVER implementations. On multiple compute nodes, Mustard achieves efficiency close to theoretical peak of the hardware. Future work will demonstrate Mustard on AMD GPUs using ROCm technologies and include more application studies to show its applicability to other task graphs.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

References

[1] Elena Agostini, Davide Rossetti, and Sreeram Potluri. 2017. Offloading communication control logic in GPU accelerated applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 248–257.

[2] AMD. 2023. HIP Runtime API reference: Graph Management. https://docs.amd.com/projects/HIP/en/docs-5.0.0/doxygen/html/group___graph.html.

[3] Cedric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. 2024. CUDASTF: Bridging the Gap Between CUDA and Task Parallelism . In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–17.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th Int'l Euro-Par Conference*. Springer, 863–874.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[6] Javid Baydamirli, Tal Ben Nun, and Didem Unat. 2024. Autonomous Execution for Multi-GPU Systems: Compiler Support. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1129–1140. https://doi.org/10.1109/SCW63240.2024.00155

[7] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. *SIGPLAN Not.* 53, 1 (feb 2018), 54–67.

[8] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. ACM, New York, NY, USA, 235–248.

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[10] Tiago Carneiro Pessoa, Jan Gmys, Francisco Heron de Carvalho Júnior, Nouredine Melab, and Daniel Tuyttens. 2018. GPU-accelerated backtracking using CUDA Dynamic Parallelism. *Concurrency and Computation: Practice and Experience* 30, 9 (2018), e4374.

[11] Long Chen, Oreste Villa, and Guang R Gao. 2011. Exploring fine-grained task-based execution on multi-GPU systems. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 386–394.

[12] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D. Owens. 2022. Scalable Irregular Parallelism with GPUs: Getting CPUs out of the Way. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC'22)*. IEEE, Article 50, 16 pages.

[13] Jaemin Choi, David F Richards, and Laxmikant V Kale. 2021. CharminG: A Scalable GPU-resident Runtime System. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 261–262.

[14] Francisco Corbera, Andrés Rodríguez, Rafael Asenjo, Angeles Navarro, Antonio Vilches, and María J Garzarán. 2015. Reducing overheads of dynamic scheduling on heterogeneous chips. *arXiv preprint arXiv:1501.03336* (2015).

[15] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-Side Library for High Performance Networking from GPU Kernels. In *Proceedings of the 6th Int'l Workshop on Runtime and Operating Systems for Supercomputers (ROSS'16)*. ACM, Article 6, 8 pages.

[16] James W Demmely, Nicholas J Highamz, and Robert S Schreiberx. 1992. Block LU Factorization. (1992).

[17] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. ACM, New York, NY, USA, Article 26, 18 pages. https://doi.org/10.1145/3295500.3356223

[18] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. 2013. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 1299–1308.

[19] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–14.

[20] Julien Herrmann, M. Yusuf Özkaya, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. 2019. Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs. *SIAM Journal on Scientific Computing (SISC)* 41, 4 (2019), A2117–A2145. https://doi.org/10.1137/18M1176865

[21] Reazul Hoque and Pavel Shamis. 2018. Distributed Task-Based Runtime Systems - Current State and Micro-Benchmark Performance. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 934–941.

[22] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs Are Not in Charge. In *Proceedings of the 37th International Conference on Supercomputing (ICS'23)*. ACM, 192–202.

[23] Łukasz Jarząbek and Paweł Czarnul. 2017. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *The Journal of Supercomputing* 73 (2017), 5378–5401.

[24] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proc. VLDB Endow.* 11, 3 (nov 2017), 297–310.

[25] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[26] Bernhard Kerbl, Michael Kenzel, Joerg H Mueller, Dieter Schmalstieg, and Markus Steinberger. 2018. The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu. In *Proceedings of the 2018 Int'l Conference on Supercomputing*. 76–85.

[27] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 8343–8354.

[28] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2017. GPU Triggered Networking for Intra-Kernel Communications. In *Proceedings of the Int'l Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC'17)*. ACM, Article 22, 12 pages.

[29] NVIDIA. 2023. CUDA Runtime API: Graph Management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html.

[30] NVIDIA. 2023. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[31] NVIDIA. 2023. NVIDIA GPUDirect Storage. https://developer.nvidia.com/gpudirect-storage.

[32] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[33] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: dynamic scheduling on GPUs. *ACM Trans. on Graphics (TOG)* 31, 6 (2012), 1–11.

[34] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (nov 2014), 11 pages.

[35] David Troendle, Tuan Ta, and Byunghyun Jang. 2019. A specialized concurrent queue for scheduling irregular workloads on GPUs. In *Proceedings of the 48th Int'l Conference on Parallel Processing*. 1–11.

[36] Ilyas Turimbetov, Muhammad Aditya Sasongko, and Didem Unat. 2024. GPU-Initiated Resource Allocation for Irregular Workloads. In *Proceedings of the 3rd International Workshop on Extreme Heterogeneity Solutions* (Edinburgh, United Kingdom) *(ExHET '24)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3642961.3643799

[37] Stanley Tzeng, Anjul Patney, and John D Owens. 2010. Task management for irregular-parallel workloads on the GPU. (2010).

[38] Didem Unat, Ilyas Turimbetov, Mohammed Kefah Taha Issa, Doğan Sağbili, Flavio Vella, Daniele De Sensi, and Ismayil Ismayilov. 2024. The Landscape of GPU-Centric Communication. arXiv:2409.09874 [cs.DC] https://arxiv.org/abs/2409.09874

[39] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. 2018. Generic system calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 843–856.

[40] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. 2020. Ouroboros: virtualized queues for dynamic memory management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. ACM, New York, NY, USA, Article 38, 12 pages.

[41] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. 2023. PERKS: a Locality-Optimized Execution Model for Iterative Memory-bound GPU Applications. In *Proceedings of the 37th ACM International Conference on Supercomputing (ICS '23)*. ACM, New York, NY, USA, 167–179.