Adaptive Task Aggregation for High-Performance Sparse Solvers on GPUs

Ahmed E. Helal*, Ashwin M. Aji†, Michael L. Chu†, Bradford M. Beckmann†, and Wu-chun Feng*‡
Electrical & Computer Eng.*, and Computer Science‡, Virginia Tech,
AMD Research, Advanced Micro Devices, Inc.†
Email: {ammhelal,wfeng}@vt.edu, {Ashwin.Aji, Mike.Chu, Brad.Beckmann}@amd.com

***Abstract*—Sparse solvers are heavily used in computational fluid dynamics (CFD), computer-aided design (CAD), and other important application domains. These solvers remain challenging to execute on massively parallel architectures, due to the sequential dependencies between the fine-grained application tasks. In particular, parallel sparse solvers typically suffer from substantial scheduling and dependency-management overheads relative to the compute operations. We propose adaptive task aggregation (ATA) to efficiently execute such irregular computations on GPU architectures via hierarchical dependency management and low-latency task scheduling. On a gamut of representative problems with different data-dependency structures, ATA significantly outperforms existing GPU task-execution approaches, achieving a geometric mean speedup of 2.2× to 3.7× across different sparse kernels (with speedups of up to two orders of magnitude).**

***Index Terms*—data dependency, fine-grained parallelism, GPUs, runtime adaptation, scheduling, sparse linear algebra, task-parallel execution**

I. Introduction

Iterative and direct solvers for sparse linear systems [1], [2] constitute the core kernels in many application domains, including computational fluid dynamics (CFD), computer-aided design (CAD), data analytics, and machine learning [3]–[9]; thus, sparse benchmarks are used in the procurement and ranking of high-performance computing (HPC) systems [10]. Sparse solvers are inherently sequential due to data dependencies between the application tasks. Representing such irregular computations as directed acyclic graphs (DAGs), where nodes are compute tasks and edges are data dependencies across tasks, exposes concurrent tasks that can run in parallel without violating the strict partial order in user applications.

DAG execution requires mechanisms to determine when a task is ready by tracking the progress of its predecessors (i.e., *dependency tracking*) and by ensuring that all its dependencies are met (i.e., *dependency resolution*). Thus, the performance of a task-parallel DAG is largely limited by its processing overhead, that is, launching the application tasks and managing their dependencies. Since sparse solvers consist of fine-grained tasks with relatively few operations, the task-launch latency and dependency-management overhead can severely impact the speedup on massively parallel architectures, such as GPUs. Therefore, the efficient execution of fine-grained, task-parallel DAGs on data-parallel architectures remains an open problem. With the increasing performance and energy efficiency of GPUs [11], [12], driven by the exponential growth of data analytics and machine learning applications [13], [14], addressing this problem has become paramount.

Many software approaches have been proposed to improve the performance of irregular applications with fine-grained, data-dependent parallelism on many-core GPUs. Level-set methods [15]–[19] adopt the bulk synchronous parallel (BSP) model [20] by aggregating the independent tasks in each DAG level to execute them concurrently with barrier synchronizations between levels. Hence, these approaches are constrained by the available parallelism in the level-set DAG, which limits their applicability to problems with a short critical path. Furthermore, since level-set execution manages all data dependencies using global barriers, it suffers from significant workload imbalance and resource underutilization.

Self-scheduling techniques [21]–[25] minimize the latency of task launching by dispatching all the application tasks at once and having them actively wait (spin-loop) until their predecessors complete and the required data is available. However, active waiting not only wastes compute cycles, but it also severely reduces the effective memory bandwidth due to resource/memory contention. Specifically, the application tasks at lower DAG levels incur substantial active-waiting overhead and interfere with their predecessor tasks, including those on the critical path. Moreover, the application data, along with its task-parallel DAG, must fit in the limited GPU memory, which is typically much smaller than the host memory. To avoid deadlocks, these self-scheduling schemes rely on application-specific characteristics or memory locks [26], which restrict their portability and performance.

Hence, there exists a compelling need for a scalable approach to manage data dependencies across millions of fine-grained tasks on many-core architectures. To this end, we propose *adaptive task aggregation (ATA)*, a software approach for the efficient execution of fine-grained, irregular applications such as sparse solvers on GPUs. ATA represents these irregular applications as hierarchical DAGs, where nodes are multi-grained application tasks and edges are their aggregated data dependencies, to match the capabilities of massively parallel GPUs by minimizing the DAG processing overheads while exposing the maximum fine-grained parallelism.

Specifically, ATA ensures deadlock-free execution and performs *multi-level dependency tracking and resolution* to amortize the task launch and dependency management overheads.

First, it leverages GPU streams/queues to manage data dependencies across the aggregated tasks [27]. Second, it uses low-latency scheduling and in-device dependency management to enforce the execution order between the fine-grained tasks in each aggregated task. Unlike previous work, ATA is aware of the structure and processing overhead of application DAGs. Thus, ATA provides generalized support for efficient fine-grained, task-parallel execution on GPUs without needing additional hardware logic. In all, our contributions are as follows:

- Unlike previous studies, we show that the performance of a fine-grained, task-parallel DAG depends not only on the problem size and the length of critical path (i.e., number of levels) but also on the DAG shape and structure. We point out that self-scheduling approaches [21]–[25] are even worse than traditional data-parallel execution for problems with a wide DAG (§IV).
- We propose the adaptive task aggregation (ATA) framework to efficiently execute irregular applications with fine-grained, data-dependent parallelism as a hierarchical DAG on GPU architectures, regardless of the data-dependency characteristics or the shape of their fine-grained DAGs (§III).
- The experimental results for a set of important sparse solver kernels, namely sparse triangular solve (SpTS) and sparse incomplete LU factorization (SpILU0) across a wide range of representative problems, show that ATA achieves a geometric mean speedup of 2.2× to 3.7× (with speedups of up to two orders of magnitude) over state-of-the-art DAG execution approaches on AMD GPUs (§IV).

II. Background and Motivation

A. GPU Architecture and Execution Models

Figure 1 depicts the recent VEGA GPU architecture from AMD [28], which consists of multiple compute units (CUs) organized into shader engines (SEs). Each CU contains single-instruction, multiple-data (SIMD) processing elements. SEs share global memory and level-2 (L2) cache, while CUs have their own dedicated local memory and level-1 (L1) cache. At runtime, the control/command processor (CP) dispatches the workload (kernels) to the available SEs and their CUs. Like GPU hardware, GPU kernels have a hierarchical thread organization consisting of workgroups of multiple 64-thread wavefronts. The SIMD elements execute each wavefront in lockstep; thus, wavefronts are the basic scheduling units.

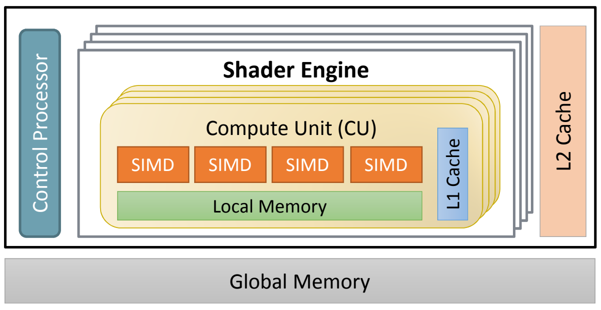


Fig. 1: VEGA GPU architecture.

Massively parallel GPUs provide fundamental support for the bulk synchronous parallel (BSP) execution model [20], where the computations proceed in data-parallel supersteps. Figure 2 depicts a BSP superstep that consists of three phases: local computations on each CU, global communication (data exchange) via main memory, and barrier synchronization. In BSP execution, the computations in each superstep must be independent and can be executed in any order. To improve workload balance, each CU should perform a similar amount of operations. Moreover, GPUs require massive computations in each superstep to utilize the available compute resources and to hide the long memory-access latency. Due to these limitations, the efficient BSP execution of irregular applications with variable and data-dependent parallelism is challenging.

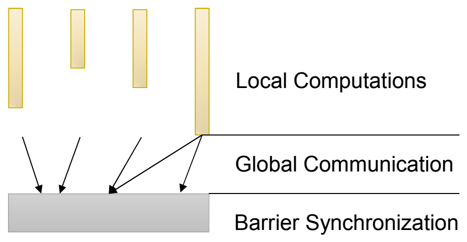


Fig. 2: The execution of a BSP superstep.

Alternatively, kernel-level DAG execution models [27], [29]–[31] support irregular applications by launching each user task as a GPU kernel and by using host/device-side streams/queues to manage data dependencies across kernels. In such runtime systems, the task launch overhead is on the order of microseconds, and the dependency management using streams/queues only supports a finite number of pending dependencies. Thus, these execution models are limited to coarse-grained DAGs, where user tasks execute thousands of instructions, which is atypical of sparse solvers.

Meanwhile, approaches with persistent threads (PT) [32]–[37] use distributed task queues to manage data dependencies and balance workload across persistent workers on the GPU, which introduces significant processing overhead. Moreover, PT execution reduces resource utilization and limits the ability of hardware schedulers to hide data access latencies. While GPUs require massive multithreading to hide memory latency [28], [38], [39], PT execution runs one worker per compute unit. Therefore, these frameworks typically achieve limited performance improvement compared to the traditional data-parallel execution (e.g., 1.05 to 1.30-fold speedup [37]) with portability issues across different GPU devices.

B. Sparse Solvers

The iterative [2] and direct methods [1] for solving sparse linear systems generally consist of two phases: (1) a pre-processing phase that is performed only once to analyze and exploit the underlying sparse structure and (2) a system solution phase that is repeated several times. The system solution phase is typically dominated by irregular computations with data-dependent parallelism, namely, preconditioners and triangular solve in iterative methods and matrix factorization/decomposition and triangular solve in direct methods.

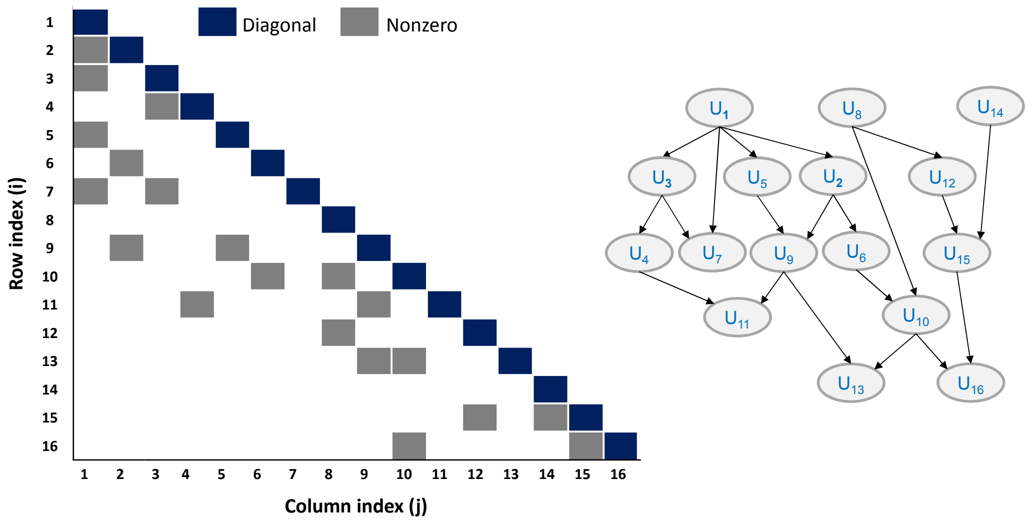


Fig. 3: A triangular matrix and the corresponding DAG for SpTS.

Such data-dependent kernels can be executed in parallel as a computational DAG, where each node represents the compute task associated with a sparse row/column and edges are the dependencies across tasks.

Algorithm 1 Sparse Triangular Solve (SpTS)

```
Input: L, RHS          ▷ Triangular matrix and right-hand side vector
Output: u                                 ▷ Solution vector for unknowns
 1: for  $i = 1$ to $n$ do
 2:     $u(i) = RHS(i)$
 3:     for  $j = 1$ to $i - 1$ where $L(i, j) \neq 0$ do        ▷ Predecessors
 4:         $u(i) = u(i) - L(i, j) \times u(j)$
 5:     end for
 6:     $u(i) = u(i)/L(i, i)$
 7: end for
```

Algorithm 1 and Figure 3 show an example of the irregular computations in sparse solvers. In SpTS, each nonzero entry (i, j) in the triangular matrix indicates that the solution of unknown i (task u_i) depends on the solution of unknown j (task u_j); hence, the DAG representation of SpTS associates an edge from node u_j to node u_i. The resulting DAG can be executed using a push or pull traversal [40]. In push traversal, the active tasks push their results and active state to the successor tasks; while in pull traversal, the active tasks pull the results from their predecessor tasks. In addition to the representative SpTS and SpILU0 kernels that are extensively discussed in this work, several sparse solver kernels (e.g., LU/Cholesky factorization, Gauss-Seidel, and successive over-relaxation [1], [2], [41]) exhibit similar irregular computations.

To execute a task-parallel DAG on GPUs using the data-parallel BSP model, the independent tasks in each DAG level are aggregated and executed concurrently with global barrier synchronization between the different levels. (This parallelization approach is often called level-set execution or wavefront parallelism [2], [23].) For example, the BSP execution of the DAG in Figure 3 runs tasks U_1, U_8, and U_{14} first, while the rest of tasks will wait for their completion at the global barrier. Since the local dependencies between tasks are replaced with global barriers, the BSP execution of a DAG suffers from barrier synchronization overhead, workload imbalance, and idle/waiting time. Furthermore, the GPU performance becomes even worse for sparse systems with limited parallelism and few nonzero elements per row/column [5], [42]. At this fine granularity, the dispatch, scheduling, and dependency management overheads can become the dominant bottlenecks.

III. Adaptive Task Aggregation (ATA)

To address the limitations of the traditional data-parallel execution and previous approaches for fine-grained, task-parallel applications, we propose the *adaptive task aggregation (ATA)* framework. The main goal of ATA is to efficiently execute irregular computations, where the parallelism is limited by data dependencies, on throughput-oriented, many-core architectures with thousands of threads. On the one hand, there is a tradeoff between the task granularity and concurrency; that is, the maximum parallelism and workload balance are only attainable at the finest task granularity (e.g., a sparse row/column in sparse solvers). On the other hand, the overhead of managing data dependencies and launching ready tasks at this fine-grained level can adversely impact the overall performance.

Thus, ATA strives to dispatch fine-grained tasks, as soon as their dependencies are met, to the available compute units (CUs) with minimal overhead and regardless of the DAG structure of the underlying problem. First, ATA represents the irregular computations as a hierarchical DAG by means of dependency-aware task aggregation for high-performance execution on GPUs (§III-A). Second, it ensures efficient, deadlock-free execution of the hierarchical DAG using multi-level dependency management and sorted eager-task (SET) scheduling (§III-B). Furthermore, ATA supports both the push and pull execution models of task-parallel DAGs and works on current GPU architectures without the need for special hardware support. While any input/architecture-aware task aggregation can be used to benefit from ATA's hierarchical execution and efficient scheduling and dependency management, we propose concurrency-aware and locality-aware aggregation policies to provide additional performance trade-offs (§III-C).

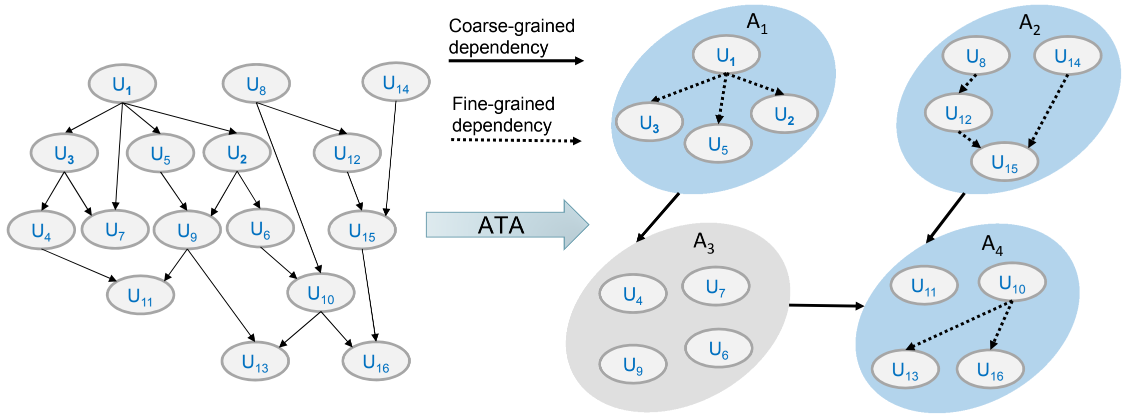


Fig. 4: ATA transformation of the application DAG in Figure 3 for hierarchical execution and dependency management. The adaptive tasks A_1, A_2, and A_4 require fine-grained dependency tracking and resolution, while A_3 can be executed as a data-parallel kernel.

A. *Hierarchical DAG Transformation*

The first stage of our ATA framework analyzes the given fine-grained DAG and then generates a hierarchy of tasks to better balance the processing overheads, that is, task launch and dependency management overheads, while exposing the maximum parallelism to many-core GPUs. This transformation can be incorporated in the preprocessing phase of irregular applications, such as sparse solvers, with negligible additional overhead (see §IV).

Consider an application DAG, $G(U, E)$, where U is a set of nodes that represents user (or application) tasks and E is a set of edges that represents data dependencies. Further, let n be the number of user tasks and m be the number of dependency edges across user tasks. ATA aggregates user tasks into adaptive tasks such that each adaptive task has a positive integer number S of the fine-grained user tasks, where S is an architecture-dependent parameter that can be estimated and tuned using profiling (as detailed in §III-C). The resulting set A of adaptive tasks partitions the application DAG such that $A_1 \cup A_2 \cdots \cup A_p = U$ and $A_i \cap A_j = \phi \; \forall i$ and j, where p is the number of adaptive tasks, $p \leq n$, and $i \neq j$.

This task aggregation delivers several benefits on many-core GPUs. First, the resulting adaptive tasks incur a fraction ($1/S$) of the launch overhead of user tasks. Second, adaptive tasks reduce the execution latency of their user tasks by dispatching the irregular computations to CUs as soon as their pending coarse-grained dependencies are resolved. Third, task aggregation eliminates dependency edges across user tasks that exist in different adaptive tasks, such that an adaptive task with independent user tasks does *not* require any dependency management. Hence, ATA generates a transformed DAG with c coarse-grained dependencies across adaptive tasks and f fine-grain dependencies across user tasks that exist in the same adaptive task, where $c + f < m$.

Figure 4 shows an example of the DAG transformation with an arbitrary task aggregation policy (see §III-C for our proposed policies). The original DAG consists of 16 user tasks with 20 dependency edges; after the DAG transformation such that each adaptive task has four user tasks, ATA generates a hierarchical DAG that consists of four adaptive tasks with only three coarse-grained dependency edges and eight fine-grained dependency edges. Since the DAG processing overhead depends on the number of tasks and dependency edges, the transformed DAG is more efficient for execution on GPU architectures. Specifically, unlike level-set execution, which is constrained by managing all data dependencies using global barriers, ATA can launch more tasks per GPU kernel to amortize the cost of kernel launch and to reduce the idle/waiting time. Most importantly, compared to self-scheduling approaches, ATA adjusts to the underlying dependency structure of target problems by executing adaptive tasks without dependency management when it is possible and by dispatching the waiting user tasks when there is limited concurrency to efficiently utilize the GPU resources. That way, ATA dispatches the ready adaptive tasks rather than the whole DAG, and as a result, the waiting adaptive tasks along with their user tasks do not incur any active-waiting overhead.

Previous work showed the benefits of aggregating fine-grained application tasks on CPU architectures [43]; however, each aggregated task (or super-task) was assigned to one thread/core to execute *sequentially* without the need for managing data dependencies across its fine-grained computations. In contrast, GPU architectures (with their massive number of compute resources) demand *parallel* execution both within and across aggregated tasks, which introduces several challenges and requires an efficient approach for managing the data dependencies and executing the irregular computations at each hierarchy level of the transformed DAG.

B. *Hierarchical DAG Execution on GPUs*

The ATA framework orchestrates the processing of millions of fine-grained user tasks, which are organized into a hierarchical DAG of adaptive tasks. Such adaptive tasks execute as GPU kernels on multiple CUs, while their user tasks run on the finest scheduling unit defined by the GPU architecture, such as wavefronts, to improve workload balance and to expose maximum parallelism.

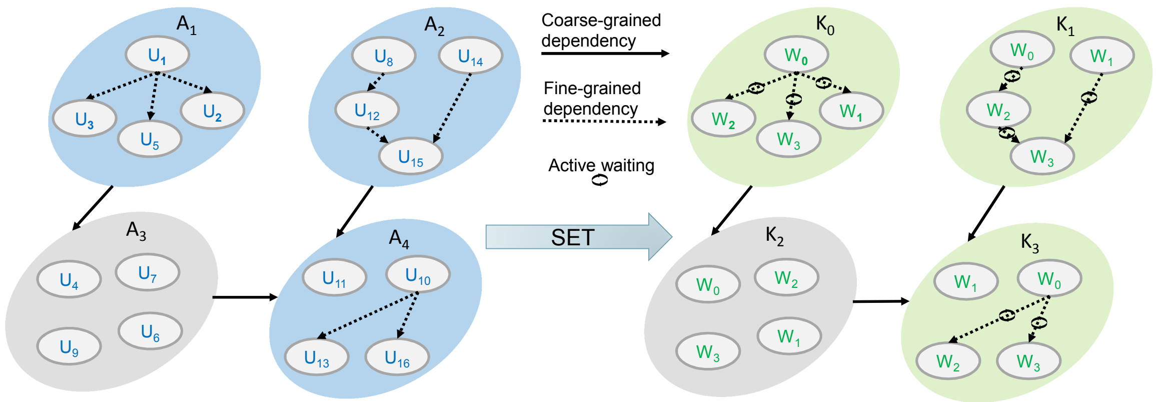


Fig. 5: SET scheduling of the hierarchical DAG in Figure 4. Each adaptive (A) task executes as a GPU kernel (K) with fine-grained dependency management using active waiting when deemed necessary. SET scheduling ensures forward progress by mapping the user (U) tasks to the worker wavefronts (W).

ATA performs hierarchical dependency management by tracking and resolving data dependencies at two levels: (1) a coarse-grained level across adaptive tasks and (2) a fine-grained level across user tasks in the same adaptive task. The coarse-grained dependency management relies on host- or device-side streams/queues to monitor the progress of GPU kernels that represent adaptive tasks and to dispatch the waiting adaptive tasks to the GPU device once their coarse-grained dependencies are met. Currently, ATA leverages the open-source ATMI runtime [27] to dispatch adaptive tasks and to manage the coarse-grained (kernel-level) dependencies.

The fine-grained dependency management requires a low-latency approach with minimal overhead, relative to the execution time of the fine-grained user tasks. Thus, ATA manages the fine-grained dependencies using lock-free data structures, where each user task tracks and resolves its pending dependencies using active waiting (i.e., polling on the shared data structures) to enforce the DAG execution order. Most importantly, ATA ensures forward progress and minimizes the active waiting overhead by assigning the waiting user tasks that are more likely to meet their dependencies sooner to the active scheduling units (wavefronts) on a GPU using SET scheduling.

1) SET Scheduling: To efficiently execute adaptive tasks on many-core GPUs, we propose sorted eager task (SET) scheduling, which aims to minimize the processing overhead by eliminating the launch and dependency resolution overheads using eager task launching and by minimizing the dependency-tracking overhead using an implicit priority scheme.

Figure 5 shows the SET scheduling of a hierarchical DAG with 16 user tasks and four (4) adaptive (aggregated) tasks. First, SET dispatches all the user tasks in an adaptive task as a GPU kernel to eliminate the task launch overhead. That way, the entire adaptive task can be processed by the GPU command processor (CP) to assign its user tasks to CUs before their predecessors even complete. However, user tasks with pending dependencies check that their predecessors finish execution using active waiting to prevent data races. Once the predecessors of a waiting user task complete, it becomes immediately ready and proceeds for execution, which eliminates the dependency-resolution overhead. To ensure forward progress, the waiting user tasks cannot be scheduled on a compute unit before their predecessors are active. While hardware memory locks [26] can be used to avoid deadlocks, they are not suitable for scheduling *large-scale* DAGs with *fine-grained* tasks because of their limited number and significant scheduling latency. In contrast, SET proposes a priority scheme that achieves deadlock-free execution with minimal overhead and without needing specialized hardware.

SET prioritizes the execution of the waiting user tasks that are more likely to be ready soon to minimize the dependency-tracking (active waiting) overhead and to prevent deadlocks. However, current many-core architectures do not provide a priority scheme with enough explicit priorities to handle a large number (potentially millions) of tasks. Thus, SET uses a more implicit technique and exploits the knowledge that hardware schedulers execute wavefronts and workgroups with lower global ID first. According to GPU programming and execution specifications, such as the HSA programming manual [39], only the oldest workgroup (and its wavefronts) is guaranteed to make forward progress; hence, the workgroup scheduler dispatches the oldest workgroup first when there are enough resources on target CUs. Moreover, the wavefront scheduler runs a single wavefront until it stalls and then picks the oldest ready wavefront [44]. In turn, the oldest hardware scheduling units (wavefronts) with the smallest global IDs are implicitly prioritized.

Therefore, SET assigns user tasks with *fewer number of dependency levels* to *older wavefronts*. Since GPU hardware schedules concurrent wavefronts to maximize resource utilization as noted in §III-C, the dependency level of a user task approximates its waiting time for dependency resolution. If there are multiple user tasks with the same number of dependency levels, SET assigns neighboring user tasks to adjacent wavefronts to improve data locality. For example, in Figure 5, U_1 is a root task (no predecessors), while U_3, U_5, and U_2 have one level of data dependency; hence, SET assigns U_1, U_3, U_5, and U_2 to the worker wavefronts W_0, W_2, W_3, and W_1 in kernel K_0. Since all U tasks in A_3 are independent, SET executes A_3 without any dependency tracking and resolution and assigns the neighboring U_4, U_6, U_7, and U_9 tasks to the adjacent W_0, W_1, W_2, and W_3 wavefronts in K_2.

Algorithm 2 Push or pull execution of adaptive tasks on GPU architectures.

```
Require: app_data                ▷ For example, a sparse matrix.
Require: SET_sched               ▷ Schedule of U tasks on worker wavefronts.
Require: u_deps                  ▷ No. of pending dependencies for each U task.
Require: u_done                  ▷ The state of each U task.
 1: for ∀ U tasks in parallel do
 2:     $i = \mathbb{GET\_UTASK}$(SET_sched)
 3:     while ATOMIC(u_deps($i$) $\neq 0$) do           ▷ Active waiting
 4:         NOOP
 5:     end while
 6:     Compute task $i$ on the worker SIMD units
 7:     for each $j$ successor of task $i$ do
 8:         ATOMIC(u_deps($j$) = u_deps($j$) $- 1$)
 9:     end for
10:     for each $j$ predecessor of task $i$ do
11:         while ATOMIC(u_done($j$) $\neq 1$) do       ▷ Active waiting
12:             NOOP
13:         end while
14:         Perform ready ops. of task $i$ on worker SIMD units
15:     end for
16:     ATOMIC(u_done($i$) = 1)
17: end for
```

2) Push vs. Pull Execution within Adaptive Tasks: ATA supports both the push and pull execution models of a computational DAG. Algorithm 2 shows the high-level (abstract) execution of adaptive tasks with fine-grained data dependencies using push or pull models (as indicated by the different gray backgrounds). In push execution, ATA uses an auxiliary data structure (*u_deps*) to manage the fine-grained data dependencies by tracking the number of pending dependencies for each user task. Once all dependencies are met, user tasks can proceed to execute on the SIMD units of their worker wavefront. (The assignment of user tasks to worker wavefronts is determined by the SET schedule.) When a user task completes its operations, it pushes the active state to its successors by decreasing their pending dependencies. Hence, push execution often needs many atomic write operations. Conversely, pull execution tracks the active state of user tasks using the *u_done* data structure. As such, each user task pulls the state of its predecessors and cannot perform the dependent computations until the predecessor tasks finish execution. Once a user task completes, it updates the corresponding state in *u_done*. Thus, pull execution performs more read operations compared to the push model. However, it can pipeline the computations (lines 10 and 14 in Algorithm 2) to hide the memory access latency.

C. Task Aggregation Policies

Finding the optimal granularity of a given application's DAG on a many-core GPU is a complicated process. First, the active waiting (dependency tracking) overhead increases with the size of aggregated tasks. In addition, a user task on the critical path can delay the execution of its aggregated task, including the other co-located user tasks. On the other hand, as the size of aggregated tasks becomes larger, the cost of managing their coarse-grained dependencies and launching user tasks decreases; moreover, increasing the size of aggregated tasks reduces the idle/waiting time, including dependency resolution time, which improves the resource utilization. Therefore, optimal task aggregation requires detailed application and architecture modeling as well as sophisticated tuning and profiling. However, by leveraging the knowledge of the target hardware architecture and application domain, simple heuristics can achieve near-optimal performance.

Unlike CPU architectures, GPUs are throughput-oriented and rely on massive multithreading (i.e., dispatching more threads/wavefronts than the available compute resources) to maximize resource utilization and to hide the execution and memory-access latencies [38]. This massive multithreading is possible due to the negligible scheduling overhead between stalled wavefronts and other active wavefronts. Thus, the GPU hardware can be efficiently used, if and only if, enough concurrent wavefronts are active (or in-flight). Hence, if each user task executes on a wavefront, the minimum size of an adaptive task, S_{min}, is limited by the number of CUs and the occupancy (active wavefronts per CU) of the GPU device:

$$S_{min} = num\_CU \times occupancy \tag{1}$$

As detailed before, increasing the size of an adaptive task has several side effects. However, any aggregation heuristic should ensure that the size of an adaptive task is large enough to amortize the cost of launching the aggregated tasks and tracking their progress. On GPUs, such cost is typically dominated by launching the aggregated tasks as GPU kernels (T_l). If the average execution time of a user task is $\overline{T_u}$, the size of an adaptive task can be tuned as follows:

$$S = R \times (T_l/\overline{T_u}), \quad S > 1 \text{ and } R > 0 \tag{2}$$

The above equation indicates that the execution time of an aggregated task should be much larger than its launch cost. Typically, R is selected such that T_l is less than 1% of the average time of an adaptive task, while the execution time of user tasks can be estimated by profiling them in parallel to determine $\overline{T_u}$. Since the dependency management overhead can be several orders of magnitude higher than the execution time of user tasks (as shown in §IV), and the profiling is performed only once in the preprocessing phase, the additional profiling overhead is negligible.

In summary, the proposed heuristic for tuning the granularity/size (S) of adaptive tasks, using Eq. (1) and (2), ensures that the performance is limited by the inherent application dependencies rather than resource underutilization, idle/waiting time, or kernel launch cost. Once the granularity is selected, different task aggregation mechanisms can be used with additional performance trade-offs. In particular, we propose the following concurrency-aware and locality-aware aggregation techniques, which are formally detailed in Algorithms 3 and 4.

Concurrency-aware (CA) Aggregation. ATA aggregates user tasks starting from the root DAG level before moving to the next levels. If the current DAG level has more than S user tasks, ATA launches this level as an adaptive task. Otherwise, it merges the next level in the current adaptive task and continues aggregating. That way, adaptive tasks end up having at least a size of S user tasks. Such an aggregation mechanism increases

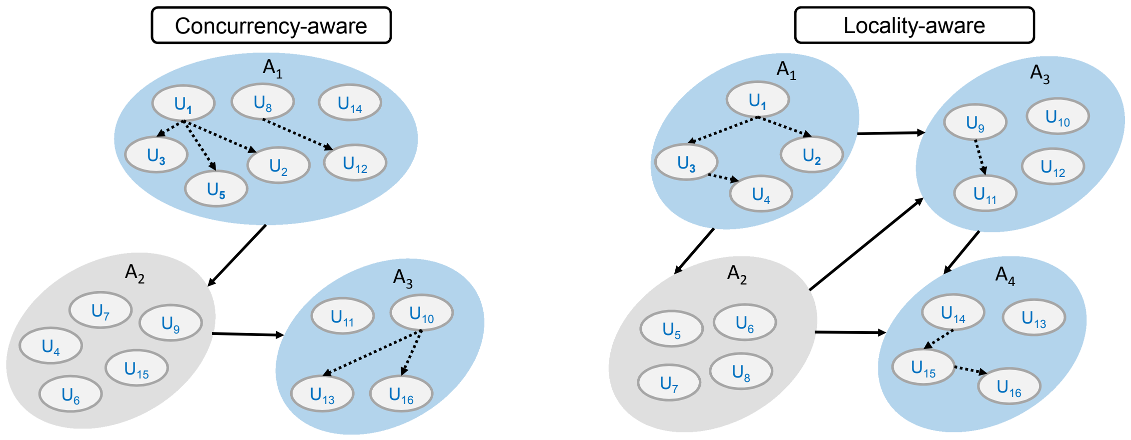


Fig. 6: Concurrency- and locality-aware aggregations of the application DAG in Figure 3. The adaptive task granularity is four.

Algorithm 3 Concurrency-aware (CA) Aggregation

```
Require: u_levels                          ▷ User tasks in each DAG level.
Require: S                     ▷ Granularity/size of adaptive tasks.
Ensure: a_tasks                                      ▷ Adaptive tasks.
1: $a\_task = \mathbb{GET\_CURRENT\_ATASK}$(a_tasks)
2: for ∀ U levels do
3:     $i = \mathbb{GET\_LEVEL}$(u_levels)
4:     $\mathbb{ADD\_UTASKS}$(a_task, u_levels(i))   ▷ Aggregate U tasks.
5:     if $\mathbb{SIZE}$(a_task) $\geq S$ then
6:         $a\_task = \mathbb{CREAT\_ATASK}$(a_tasks)
7:     end if
8: end for
```

Algorithm 4 Locality-aware (LA) Aggregation

```
Require: u_tasks, u_loc            ▷ User tasks and their locality info.
Require: S                     ▷ Granularity/size of adaptive tasks.
Ensure: a_tasks                                      ▷ Adaptive tasks.
1: $a\_task = \mathbb{GET\_CURRENT\_ATASK}$(a_tasks)
2: for ∀ U tasks do
3:     $i = \mathbb{GET\_U\_ID}$(u_loc)
4:     $\mathbb{ADD\_UTASK}$(a_task, u_tasks(i))    ▷ Aggregate U tasks.
5:     if $\mathbb{SIZE}$(a_task) $\geq S$ then
6:         $a\_task = \mathbb{CREAT\_ATASK}$(a_tasks)
7:     end if
8: end for
```

concurrency among user tasks in the same adaptive task and minimizes the overall critical path of the hierarchical DAG; however, it ignores data locality.

Locality-aware (LA) Aggregation. This policy improves data locality across the memory hierarchy by merging neighboring user tasks into the same adaptive task, which can benefit applications with high spatial locality. The task locality information is based on knowledge of standard sparse formats, and it can also be incorporated as a programmer hint. Unlike the CA approach, LA aggregation may increase the overall critical path of hierarchical DAGs, as a user task on the critical path can delay the execution of neighboring user tasks.

Figure 6 shows an example of the different aggregation policies, where the adaptive task granularity is four (4) user tasks. Due to the limited concurrency at the root DAG level, CA aggregation combines this level and the next one into the adaptive task A_1. Next, it encapsulates the third DAG level in A_2 which does not require any fine-grained dependency management. Finally, CA aggregation merges the fourth and fifth DAG levels in A_3 to reach the required granularity. In contrast, LA aggregation merges the neighboring user tasks into four adaptive tasks. While CA aggregation achieves the same critical path as the original application DAG, that is, five user tasks ($U_1 \rightarrow U_2 \rightarrow U_6 \rightarrow U_{10} \rightarrow U_{16}$), the resulting hierarchical DAG from LA aggregation has a longer critical path of nine user tasks ($U_1 \rightarrow U_3 \rightarrow U_4 \rightarrow U_5 \rightarrow U_9 \rightarrow U_{11} \rightarrow U_{14} \rightarrow U_{15} \rightarrow U_{16}$).

We also considered greedy aggregation, which combines the maximum number of user tasks that can fit on the GPU[1] in a single adaptive task. Compared to other aggregation policies, greedy aggregation does *not* adapt to the DAG structure, leading to excessive active waiting for application DAGs with high concurrency, as adaptive tasks are unlikely to execute without needing a fine-grained dependency management.

IV. Performance Evaluation

We evaluate the proposed ATA framework using a set of representative kernels for sparse solvers. These kernels implement the sparse triangular solve (SpTS) and sparse incomplete LU factorization with zero level of fill in (SpILU0) algorithms, which are detailed in Algorithms 1 and 5. Specifically, we consider the push and pull execution variants of SpTS using the compressed sparse column (CSC) and compressed sparse row (CSR) formats, respectively, and the left-looking pull execution of SpILU0 using the CSC format [1], [2]. In addition, we evaluate the end-to-end solver performance using the preconditioned conjugate gradient (PCG) method [2].

We compare ATA to level-set execution [15]–[19] and self-scheduling approaches [21]–[24]. The target GPU kernels are implemented in OpenCL, while the host code is written in C++ and leverages the open-source ATMI runtime [27] to dispatch GPU kernels. Using double-precision arithmetic, we report the performance and overhead numbers for the system solution phase as an average over 100 runs[2]. It is important to note that the different DAG execution approaches, namely, ATA, level-set, and self-scheduling, generate identical results using

[1]This number is limited by the available memory and maximum number of active wavefronts on the GPU.

[2]The reported performance for SpTS (push traversal) with self-scheduling approach is based on executing the OpenCL code from Liu et al. [23], [24].

Algorithm 5 Sparse Incomplete LU Factorization with zero level of fill in (SpILU0)

```
Require: A    ▷ Input matrix that will be decomposed into L and U
 1: for $j = 1$ to $n$ do                                   ▷ Current column
 2:     for $k = 1$ to $j - 1$ where $A(k, j) \neq 0$ do      ▷ Predecessors
 3:         for $i = k + 1$ to $n$ where $A(i, k)$ & $A(i, j) \neq 0$ do
 4:             $A(i, j) = A(i, j) - A(i, k) \times A(k, j)$  ▷ Elimination
 5:         end for
 6:     end for
 7:     for $i = j + 1$ to $n$ where $A(i, j) \neq 0$ do
 8:         $A(i, j) = A(i, j)/A(j, j)$                      ▷ Normalization
 9:     end for
10: end for
```

the same computations and only differ in the data-dependency management, as detailed in the previous sections.

A. Experimental Setup

1) Input Data: The experiments consider representative problems with different sizes and dependency structures that cover a wide range of application domains, such as fluid dynamics, electromagnetics, mechanics, atmospheric models, structural analysis, thermal analysis, power networks, and circuit simulation [45]. Table I presents the characteristics of the test problems, where the problem ID is assigned in an ascending order of the number of unknowns. Further, to clarify the experimental results, we classify the resulting application DAG of the input problems into wide DAG, L-shape DAG, and parallel DAG. Figure 7 shows an example of these different DAG types. The parallel DAG has a short critical path (typically less than 100 user tasks) such that the performance is bounded by the execution time rather than the data dependencies. In L-shape DAGs, most of the user tasks are in the higher DAG levels and the number of concurrent user tasks significantly decreases as we move down the critical path. Conversely, in wide DAGs, the majority of DAG levels are wide with enough user tasks to utilize at least the available SIMD elements in each compute unit (i.e., four wavefronts per CU in target GPUs).

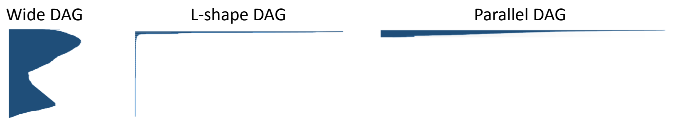


Fig. 7: An example of the different DAG classes. The x-axis shows the number of user tasks, while the y-axis represents the DAG levels (critical path).

2) Test Platform: The test platform is a Linux server with an Intel Xeon E5-2637 CPU host running at 3.50 GHz and multiple GPU devices. The server runs the Debian 8 distribution and ROCm 1.8.1 software stack, and the applications are built using GCC 7.3 and CLOC (CL Offline Compiler) 1.3.2. In the experiments, we consider two different generations of AMD GPU devices: Radeon Vega Frontier Edition [28] (VEGA-FE) and Radeon R9 Nano [46] (R9-NANO). Table II details the specifications of the target GPUs. For brevity, we only show the detailed results for the VEGA-FE GPU. In addition, we use micro-benchmarks to profile the overhead of atomic operations and kernel launch.

TABLE I: Characteristics of the sparse problems

Prob. ID	Name	#unknowns	#nonzeros
P1	onetone2	36,057	222,596
P2	onetone1	36,057	335,552
P3	TSOPF_RS_b300_c3	42,138	4,413,449
P4	bcircuit	68,902	375,558
P5	circuit_4	80,209	307,604
p6	ASIC_100ks	99,190	578,890
P7	hcircuit	105,676	513,072
P8	twotone	120,750	1,206,265
P9	FEM_3D_thermal2	147,900	3,489,300
P10	G2_circuit	150,102	726,674
P11	scircuit	170,998	958,936
P12	hvdc2	189,860	1,339,638
P13	thermomech_dK	204,316	2,846,228
P14	offshore	259,789	4,242,673
P15	ASIC_320ks	321,671	1,316,085
P16	rajat21	411,676	1,876,011
P17	cage13	445,315	7,479,343
P18	af_shell3	504,855	17,562,051
P19	parabolic_fem	525,825	3,674,625
P20	ASIC_680ks	682,712	1,693,767
P21	apache2	715,176	4,817,870
P22	ecology2	999,999	4,995,991
P23	thermal2	1,228,045	8,580,313
P24	atmosmodd	1,270,432	8,814,880
P25	G3_circuit	1,585,478	7,660,826
P26	memchip	2,707,524	13,343,948
P27	Freescale2	2,999,349	14,313,235
P28	Freescale1	3,428,755	17,052,626
P29	circuit5M_dc	3,523,317	14,865,409
P30	rajat31	4,690,002	20,316,253

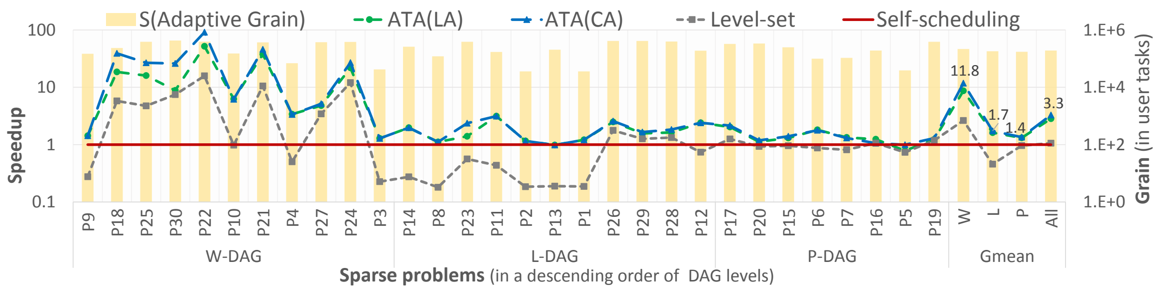
TABLE II: Target GPU architectures

GPU	Max. freq.	Memory	Mem. BW	#cores
R9-NANO	1000 MHz	4 GB	512 GB/s	4,096
VEGA-FE	1600 MHz	16 GB	483 GB/s	4,096

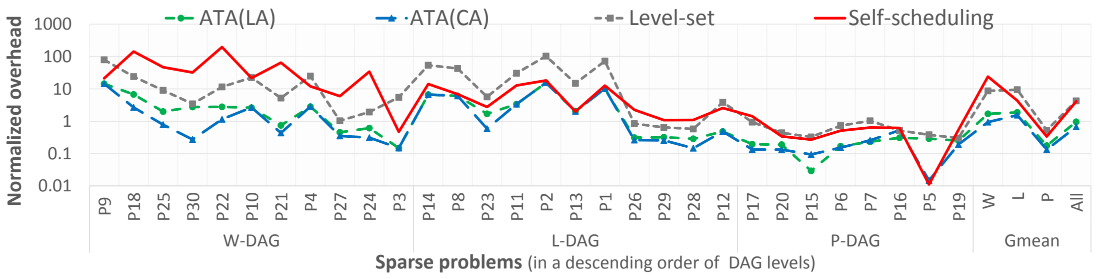
B. Experimental Results

The results reported here demonstrate the capabilities of the ATA framework with its different aggregation policies, where the adaptive task granularity (S) is selected using the profiling-based heuristic from Eq. (1) and (2). In the experiments, we set R to 100 in Eq. (2) to ensure that the overhead of coarse-grained dependency management across adaptive tasks is less than 1% of their average execution time. To measure the overhead of managing the data dependencies of the application DAGs, we execute these DAGs without any dependency management and with the different DAG execution approaches. Such overhead represents the kernel launch and workload imbalance (global synchronization) for level-set execution and the active waiting for self-scheduling methods, while it shows the processing cost of hierarchical DAGs for ATA execution as illustrated in §III-B.

Figure 8 shows the performance and overhead of the SpTS kernel using push traversal and CSC format. The results demonstrate that the ATA framework significantly outperforms the other approaches, achieving a geometric mean speedup of 3.3× and 3.7× on VEGA-FE and R9-NANO



(a) Speedup and adaptive grain



(b) Overhead relative to execution time without dependencies

Fig. 8: The performance and overhead of SpTS (push traversal) kernels using the different execution approaches on VEGA-FE.

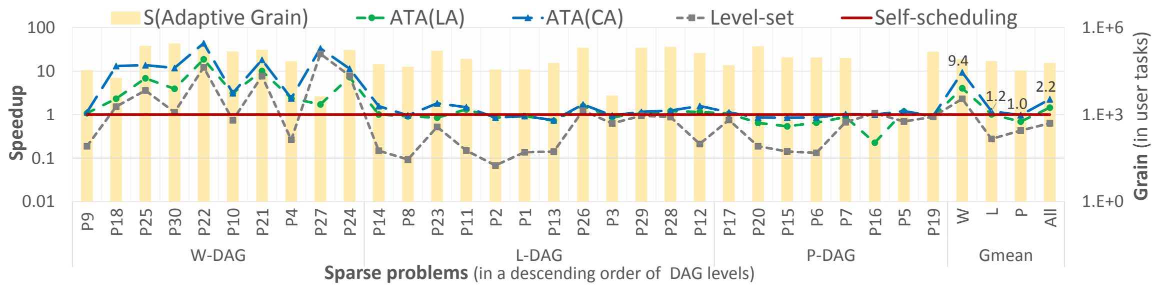
GPUs, respectively. Due to the higher cost of active waiting on the R9-NANO GPU, ATA achieves better performance compared to self-scheduling. Furthermore, the results indicate that concurrency-aware (CA) aggregation outperforms locality-aware (LA) aggregation, as sparse applications tend to have limited spatial locality and LA aggregation can also increase the critical execution path (see §III-C). In addition, the pull variant of SpTS shows a similar trend to the push execution (omitted for brevity). However, ATA has a slightly lower geometric mean speedup of 3.0× and 3.3× on VEGA-FE and R9-NANO GPUs, respectively, compared to the push execution as the pull execution requires lower dependency management overhead (see §III-B).

Most importantly, ATA has better performance across the different types of application DAGs due to its hierarchical dependency management and efficient mapping of user tasks to the active wavefronts using SET scheduling. In particular, the self-scheduling approach is even worse than level-set execution for wide DAGs because the large number of user tasks at the lower DAG levels incur significant active-waiting overhead; such overhead can be higher than the computation time by more than two orders of magnitude for large-scale problems with long critical paths, as explained in Figure 8. For L-shaped DAGs, the average performance of level-set execution is significantly worse than the other methods because of the limited number of concurrent user tasks in the majority of DAG levels; hence, the overhead of global barrier synchronization becomes prohibitive, especially for problems with deeper critical paths. On the other hand, the results for L-shaped and parallel DAGs show that level-set execution achieves comparable (or better) performance to self-scheduling as the length of the critical path (i.e., number of DAG levels) decreases, due to the higher concurrency and the lower overhead of global barrier synchronization.

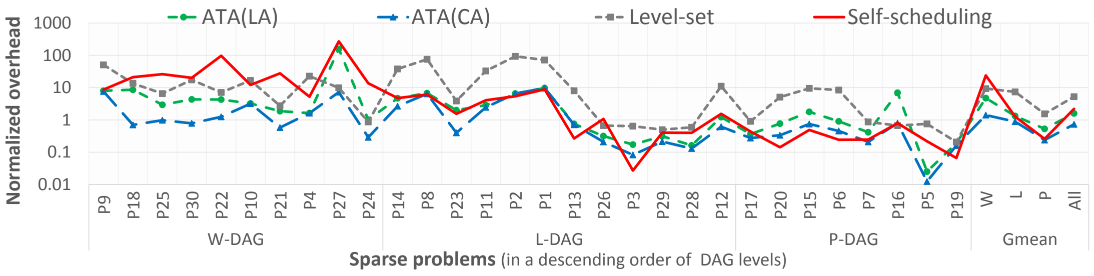
Figure 9 shows the performance and overhead of the pull execution of SpILU0 using the different DAG execution methods. Since SpILU0 performs more operations than SpTS, the dependency-management overhead is relatively smaller compared to the computation time. Specifically, in SpILU0, the number of operations is relative to the number of non-zero elements of each user task and also the non-zero elements of its predecessors, which results in roughly an order of magnitude smaller adaptive grain size (S) compared to SpTS. Hence, ATA achieves a geometric mean speedup of 2.2× for the SpILU0 kernel in comparison with a geometric mean speedup of 3.0×–3.7× for the different variants of SpTS.

Finally, Figure 10 presents the preprocessing cost required to generate ATA's hierarchical DAG from the fine-grained application DAG for each sparse problem. Since such a cost depends on the number of user tasks and data dependencies, it increases with the problem size; however, the maximum cost is approximately 0.1 second in the target benchmarks, which include sparse problems with millions of tasks (i.e., unknowns) and tens of millions of data dependencies (i.e., nonzeros). LA aggregation has a higher cost than CA aggregation because it typically uses a larger number of data dependencies, as explained in §III-C. Specifically, the geometric mean cost of generating the hierarchical DAG is 18 ms and 22 ms for the CA and LA aggregation policies, respectively.

It is important to note that once the hierarchical DAG is generated, it can be used many times during the application run. Target user applications, such as CFD and CAD applications, typically solve a nonlinear system of equations at many time points; each *nonlinear* system solution requires several iterations of a *linear* system solver, which in turn needs tens to hundreds of iterations to converge [2]. Thus, in practice, such a preprocessing cost is negligible. In addition, the preprocessing phase can be overlapped with other operations, including the



(a) Speedup and adaptive grain



(b) Overhead relative to execution time without dependencies

Fig. 9: The performance and overhead of SpILU0 (pull traversal) using the different execution approaches on VEGA-FE.

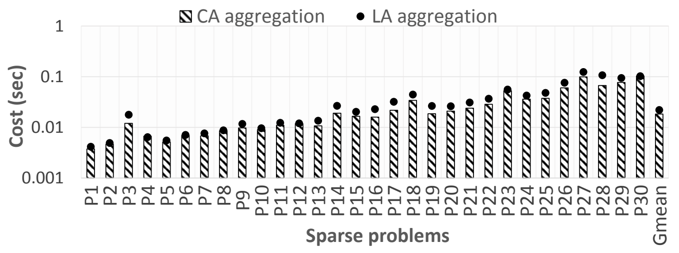


Fig. 10: The cost of hierarchical DAG transformation using the different task aggregation policies.

system solution phase. Optimizing the preprocessing cost is outside the scope of this paper and a further reduction of this cost is feasible.

C. *End-to-End Solver Performance*

To evaluate the end-to-end solver performance, we use the preconditioned conjugate gradient (PCG) method for solving linear systems with a symmetric and positive-definite (SPD) matrix [2]. We implemented a PCG solver, based on Algorithm 9.1 from Saad [2], using the data-dependent kernels discussed in the paper (namely, SpTS and SpILU0) and open-source SpMV and BLAS kernels from clSPARSE library [47]. Specifically, the data-dependent kernels of PCG solver perform pull traversal of application DAGs to execute the compute tasks. In the experiment, the right-hand side is a unit vector and the maximum number of iterations allowed to find a solution is 2000. The PCG solver converges when the relative residual (tolerance) is below one millionth (10^{-6}), starting from an initial guess of zero. We evaluate three versions of the PCG solver; each version uses different SpTS and SpILU0 kernels, based on ATA and prior level-set and self-scheduling approaches, and the same SpMV and BLAS kernels.

Figure 11 presents the execution time and number of iterations of the PCG solver for the set of SPD problems in Table I. The detailed profiling indicates that data-dependent kernels constitute the majority of execution time, ranging from 76% to 99% of total runtime across PCG solver versions and input problems. As a result, the performance of the PCG solver shows a similar trend to data-dependent kernels, where ATA significantly outperforms previous methods across the different sparse problems. The performance gain depends on the characteristics of input problems, as discussed in §IV-B. Overall, this experiment demonstrates the efficacy of the proposed framework to greatly improve end-to-end solver performance. Specifically, ATA's PCG solver achieves a geometric mean speedup of 4.4× and 8.9× compared to PCG solvers implemented using prior level-set and self-scheduling methods, respectively.

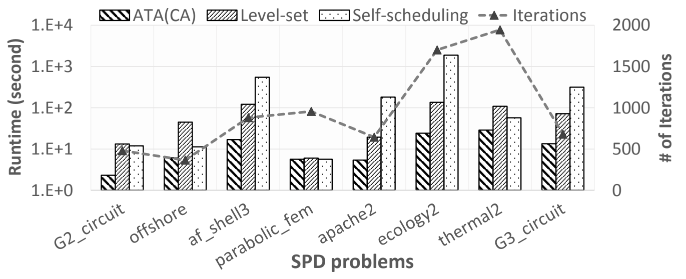


Fig. 11: The performance of PCG solver on VEGA-FE.

V. Related Work

A. *GPU Sparse Solvers*

In addition to the widely adopted level-set [15]–[19] and self-scheduling [21]–[25] techniques, which we discussed in previous sections, several approaches have been proposed to improve the performance of sparse solvers on GPUs. Graph-coloring methods [48], [49] can increase the parallelism of sparse solvers by permuting the rows and columns of input matrices; however, such a permutation breaks the original data dependencies and the corresponding DAG execution order, which affects the accuracy of the system solution and typically increases the number of iterations needed for convergence in iterative solvers [2]. In addition, finding the color sets of a DAG is an NP-complete problem that requires significant preprocessing overhead. Prior work [50], [51] used approximate algorithms to solve the target sparse system without dependency management. Similar to graph-coloring methods, these approximation algorithms affect the solution accuracy and convergence rate of sparse solvers.

Various approaches [52]–[55] exploit dense patterns in the underlying sparse problems and use dense BLAS [56] routines/kernels to improve data locality and to reduce the indirect addressing overhead; however, such techniques are limited to problems with structured sparsity patterns. Recently, Wang et al. [42] proposed sparse level tile (SLT) format, which is tailored for locality-aware execution of SpTS on Sunway architecture. Nevertheless, end users need to either refactore existing applications to use such a specialized format or convert their sparse data before and after each call to SLT-based solvers.

B. Dependency Management Frameworks

Researchers have designed many software and hardware frameworks to address the limitations of the *data-parallel* execution of irregular applications with a *data-dependent* parallelism on GPU architectures.

Juggler [37] is a DAG-based execution scheme for Nvidia GPUs that maintains task queues on the different compute units and employs persistent workers (workgroups) to execute ready tasks and to resolve the data dependencies of waiting tasks. Other frameworks [33], [34], [36] adopt a similar execution model with persistent threads (PT) on GPUs. PT execution significantly reduces GPU throughput by running one worker per compute unit, which limits the latency hiding ability of GPU hardware schedulers. Conversely, ATA executes multiple workers per compute unit to maximize the utilization of GPU resources and to expose the inherent parallelism of user applications. Moreover, achieving workload balance using distributed task queues is difficult and requires significant processing overhead. As a result, PT approaches typically execute user tasks at the granularity of workgroups. In contrast, ATA leverages the existing hardware schedulers for GPUs, which perform dynamic resource management across active wavefronts, to reduce the idle/waiting time by concurrently executing the available tasks in user applications and mapping them to active wavefronts. Further, ATA can support a wide range of granularity from wavefronts to workgroups.

Pagoda [35] and GeMTC [57] adopt a centralized scheduling approach to execute independent tasks on GPUs using a resident kernel, which distributes ready tasks to compute units at the warp/wavefront granularity. However, these frameworks assume that all dispatched tasks are ready for execution and do not support dependency tracking and resolution. Specifically, they rely on the host to send ready tasks to GPU after their dependencies are resolved. Therefore, Pagoda and GeMTC suffer from host-device communication which is the limited by the PCI-E bandwidth.

Runtime systems for task management such as StarPU [29] and Legion [30] schedule the data-dependent computations on a heterogeneous architecture consisting of multiple CPUs and GPUs. These systems consider a single device as a worker which limits their applicability to irregular applications with coarse-grained tasks, where the dependency management overhead is a fraction of the overall execution time. In addition, managing data dependencies on the host introduces significant host-device communication and synchronization overhead.

Prior software systems [58]–[61] improve the performance of dynamic parallelism, where a GPU kernel can launch child kernels, by aggregating the *independent* work-items across child kernels to amortize the kernel launch overhead; however, these techniques are not suitable to execute application DAGs with many-to-many relationship between predecessor and successor tasks. Conversely, in this paper, work aggregation is used to execute irregular applications with *data-dependent* tasks within and across GPU kernels, which requires efficient dependency tracking and resolution. Hence, ATA aggregates *data-dependent* work across user tasks with a DAG execution order and then enforces this order using a hierarchical dependency management and task scheduling scheme.

Alternatively, hardware approaches [41], [62]–[64] aggregate and execute data-dependent computations on many-core GPUs using dedicated hardware units or specialized workgroup (thread-block) schedulers. Unlike these approaches, ATA works on current

VI. Conclusion

In this paper, we proposed adaptive task aggregation (ATA) to greatly reduce the dispatch, scheduling, and dependency management overhead of irregular computations with fine-grained tasks and strong data dependencies on GPU architectures. Unlike previous work, ATA adapts to the dependency structure of underlying problems using (1) hierarchical dependency management at multiple levels of granularity and (2) efficient sorted eager task (SET) scheduling of the application tasks based on their expected dependency-resolution time.

As such, the ATA framework achieves significant performance gains across the different types of application problems. Specifically, the experiments with various sparse solver kernels demonstrated a geometric mean speedup of 2.2× to 3.7× over the existing DAG execution approaches and up to two orders-of-magnitude speedups for large-scale problems with a wide DAG and long critical path.

Acknowledgments

We thank Joe Greathouse for the technical discussions related to SpTS. This work was supported in part by the DOE PathForward program and the Synergistic Environments for Experimental Computing (SEEC) Center via a seed grant from the Institute for Critical Technology and Applied Science (ICTAS), an institute dedicated to transformative, interdisciplinary research for a sustainable future.

References

[1] T. Davis, *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Tech. Rep., 2006.

[4] B. Catanzaro, K. Keutzer, and B.-Y. Su, "Parallelizing CAD: A Timely Research Agenda for EDA," in *Proceedings of the 45th annual Design Automation Conference (DAC)*. ACM, 2008, pp. 12–17.

[5] Y. S. Deng, B. D. Wang, and S. Mu, "Taming Irregular EDA Applications on GPUs," in *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD)*. ACM, 2009, pp. 539–546.

[6] A. E. Helal, A. M. Bayoumi, and Y. Y. Hanafy, "Parallel Circuit Simulation Using the Direct Method on a Heterogeneous Cloud," in *Proceedings of the 52nd Annual Design Automation Conference (DAC)*. ACM, 2015, pp. 186:1–186:6.

[7] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011.

[8] Y. Koren, R. Bell, and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug. 2009.

[9] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A New Framework for Parallel Machine Learning," *arXiv preprint arXiv:1408.2041*, 2014.

[10] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-Performance Conjugate-Gradient Benchmark: A New Metric for Ranking High-Performance Computing Systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[11] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[12] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan, "Survey of Techniques and Architectures for Designing Energy-Efficient Data Centers," *IEEE Systems Journal*, vol. 10, no. 2, pp. 507–519, 2016.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[14] Y. Wang, Y. Pan, A. Davidson *et al.*, "Gunrock: GPU Graph Analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.

[15] E. Anderson and Y. Saad, "Solving Sparse Triangular Linear Systems on Parallel Computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[16] J. H. Saltz, "Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors," *SIAM journal on scientific and statistical computing*, vol. 11, no. 1, pp. 123–144, 1990.

[17] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU," NVIDIA, Tech. Rep. NVR-2011-001, 2011.

[18] ——, "Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU," NVIDIA, Tech. Rep. NVR-2012-003, 2012.

[19] R. Li and Y. Saad, "GPU-Accelerated Preconditioned Iterative Linear Solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[20] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[21] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Transactions on computers*, vol. 40, no. 5, pp. 603–612, 1991.

[22] L.-S. Chien, "How to Avoid Global Synchronization by Domino Scheme," in *GPU Technology Conference (GTC)*, 2014.

[23] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves," in *Proceedings of the 22nd European Conference on Parallel Processing (Euro-Par)*. Springer, 2016, pp. 617–630.

[24] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4244, 2017.

[25] J. I. Aliaga, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Accelerating the Task/Data-Parallel Version of ILUPACK's BiCG in Multi-CPU/GPU Configurations," *Parallel Computing*, vol. 85, pp. 79–87, 2019.

[26] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-Grained Synchronizations and Dataflow Programming on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*. ACM, 2015, pp. 109–118.

[27] S. Puthoor, A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers, "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU)*. ACM, 2016, pp. 53–62.

[28] AMD, "Radeon's Next-Generation Vega Architecture," https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf, 2017.

[29] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[30] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012, pp. 1–11.

[31] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A Runtime System for Dataflow Task Programming on Heterogeneous Architectures," in *Proceedings of IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2013, pp. 1299–1308.

[32] K. Gupta, J. A. Stuart, and J. D. Owens, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," in *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*. IEEE, 2012, pp. 1–14.

[33] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg, "Softshell: Dynamic Scheduling on GPUs," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, p. 161, 2012.

[34] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, "Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 6, p. 228, 2014.

[35] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2017, pp. 221–234.

[36] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: A Versatile Programming Framework for Pipelined Computing on GPU," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 587–599.

[37] M. E. Belviranli, S. Lee, J. S. Vetter, and L. N. Bhuyan, "Juggler: A Dependence-aware Task-based Execution Framework for GPUs," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2018, pp. 54–67.

[38] M. Garland and D. B. Kirk, "Understanding Throughput-Oriented Architectures," *Communications of the ACM*, vol. 53, no. 11, pp. 58–66, Nov. 2010.

[39] P. Rogers and A. Fellow, "Heterogeneous System Architecture Overview," in *IEEE Hot Chips 25 Symposium (HCS)*. IEEE, 2013, pp. 1–41.

[40] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 456–471.

[41] A. A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting Data-Dependent Parallelism through Dependency Graph Execution in GPUs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 600–611.

[42] X. Wang, W. Xue, W. Liu, and L. Wu, "swSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2018, pp. 338–353.

[43] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver," in *Proceedings of the 29th International Supercomputing Conference (ISC)*. Springer, 2014, p. 124.

[44] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2012, pp. 72–83.

[45] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[46] J. Macri, "AMD's Next-Generation GPU and High-Bandwidth Memory Architecture: FURY," in *IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–26.

[47] J. L. Greathouse, K. Knox, J. Poła, K. Varaganti, and M. Daga, "clSPARSE: A Vendor-Optimized Open-Source Sparse BLAS Library," in *Proceedings of the 4th International Workshop on OpenCL (IWOCL)*. ACM, 2016, p. 7.

[48] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, "Adapting Sparse Triangular Solution to GPUs," in *2012 41st International Conference on Parallel Processing (ICPP) Workshops*. IEEE, 2012, pp. 140–148.

[49] M. Naumov, P. Castonguay, and J. Cohen, "Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU," *Nvidia White Paper*, 2015.

[50] E. Chow and A. Patel, "Fine-Grained Parallel Incomplete LU Factorization," *SIAM journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[51] H. Anzt, E. Chow, and J. Dongarra, "Iterative Sparse Triangular Solves for Preconditioning," in *Proceedings of the 21st European Conference on Parallel Processing (Euro-Par)*. Springer, 2015, pp. 650–661.

[52] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal Factorization of Sparse SPD Matrices on GPUs," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 372–383.

[53] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating Sparse Cholesky Factorization on GPUs," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 2014, pp. 9–16.

[54] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault, "Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops*. IEEE, 2014, pp. 29–38.

[55] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse QR Factorization on the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 2, p. 17, 2017.

[56] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 1, pp. 1–17, 1988.

[57] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and Evaluation of the GeMTC Framework for GPU-Enabled Many-Task Computing," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2014, pp. 153–164.

[58] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[59] G. Chen and X. Shen, "Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. ACM, 2015, pp. 407–419.

[60] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 649–660.

[61] I. El Hajj, "Techniques for Optimizing Dynamic Parallelism on Graphics Processing Units," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2018.

[62] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-Grain Task Aggregation and Coordination on GPUs," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 181–192.

[63] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2015, pp. 528–540.

[64] ——, "LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 583–595.