Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs

Mehmet E. Belviranli
Oak Ridge National Laboratory
belviranlime@ornl.gov

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@ornl.gov

Laxmi N. Bhuyan
University of California, Riverside
bhuyan@cs.ucr.edu

Abstract

Scientific applications with single instruction, multiple data (SIMD) computations show considerable performance improvements when run on today's graphics processing units (GPUs). However, the existence of data dependences across thread blocks may significantly impact the speedup by requiring global synchronization across multiprocessors (SMs) inside the GPU. To efficiently run applications with inter-block data dependences, we need fine-granular task-based execution models that will treat SMs inside a GPU as stand-alone parallel processing units. Such a scheme will enable faster execution by utilizing all internal computation elements inside the GPU and eliminating unnecessary waits during device-wide global barriers.

In this paper, we propose *Juggler*, a task-based execution scheme for GPU workloads with data dependences. The Juggler framework takes applications embedding OpenMP 4.5 tasks as input and executes them on the GPU via an efficient in-device runtime, hence eliminating the need for kernel-wide global synchronization. Juggler requires no or little modification to the source code, and once launched, the runtime entirely runs on the GPU without relying on the host through the entire execution. We have evaluated Juggler on an NVIDIA Tesla P100 GPU and obtained up to 31% performance improvement against global barrier based implementation, with minimal runtime overhead.

CCS Concepts • **Software and its engineering → Runtime environments**; *Parallel programming languages*;

Keywords GP-GPU programming, task-based execution, data dependence, OpenMP 4.5.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

PPoPP '18, February 24–28, 2018, Vienna, Austria

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178492

ACM Reference Format:
Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. In *Proceedings of PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3178487.3178492

1 Introduction

Using GPUs for general-purpose (GPGPU) computation has become increasingly popular over the last decade. Massive parallelism provided by hundreds of GPU cores offers considerable speedups for SIMD type computations. In a typical GPGPU-based execution scenario, data are mapped to thousands of threads, which are logically grouped as thread blocks (TBs). Each TB is assigned to and executed on one of the GPU's multiprocessor units (e.g., SMs, SMXs, or CUs), each packing hundreds of single processing cores (e.g., SIMD lanes, SPs, or CUDA cores). If a GPU application has computational dependences across different parts of the data, hence the TBs, then the developer needs to re-design the application and break the execution into multiple kernel launches separated by device-level synchronization calls (i.e., *global barriers*). This is due to the fact that existing GPGPU programming models, such as CUDA, force TBs running on different SMs to be unaware of each other's progress because SMs lack efficient hardware communication mechanisms between each other.

A global barrier based GPU execution scheme usually leads to two main problems: (1) Some SMs are forced to stay idle as the execution approaches the synchronization point, hence leading to underutilization of computing resources [7, 32]. (2) Because the GPU hardware scheduler assigns TBs into SMs in an indefinite order, any inherent locality between the data-dependent TBs is lost [28].

We believe GPUs can be used more *efficiently* to solve problems with *data-dependent* characteristics. This needs development of task-based execution models and efficient synchronization techniques similar to how applications with MIMD style of computation are handled on multi-core systems [24]. TBs in CUDA (or workgroups in OpenCL) are the lowest level inter-SM scheduling units in current GPUs. The

conventional GPGPU programming models (e.g., OpenCL and CUDA) follow a *DOALL* parallelism model and assume that a TB can be executed on any SM in no specific order. As an alternative, we argue that GPU applications can be represented as a graph of *task*s. An in-GPU runtime should track the dependences between these tasks dynamically and assign them into SMs as soon as their computational precedences are satisfied. In addition to solving the above *global synchronization* issues, such an execution scheme would greatly expand the class of applications that can be accelerated on GPUs, without the need for a re-design.

In this paper, we propose Juggler, a new data-dependence-aware task execution framework for GPUs. The compile-time component of Juggler takes applications written with OpenMP 4.5 task directives as input and applies source-to-source transformations to form a DAG that will later represent the data/task flow. At the time of execution, the code injected by Juggler creates the DAG using live input parameters of the application and feeds it into our *all-in-GPU* device runtime. The Juggler runtime employs persistent worker TBs that grab tasks from distributed lock-free queues and process them in the correct dependence order. The runtime also employs various queue placement policies to achieve maximum load balance across SMs while preserving task locality.

Novelty: To the best of our knowledge, Juggler is the first study *to implement OpenMP 4.5 device runtime for inter-SM task execution*. Also, Juggler is the only in-GPU task-based execution framework that *resolves dependences on-the-fly & in-the-GPU* and schedules newly released tasks *without going back to CPUs*.

We implemented the compiler transformations on OpenARC [18], an open-source research compiler, and evaluated our runtime on an NVIDIA Tesla P100 GPU. Juggler improved performance up to 31% compared to the classic global barrier based approach. The overhead of Juggler with best performing scheduling policy was no more than 6%.

This paper makes the following contributions:

- We implement a *novel*, dependence-aware, in-device task execution runtime for GPUs.
- We present compiler transformations to integrate our runtime automatically into applications embedding OpenMP 4.5 task construct.
- We further explore the performance of the proposed system using different queue-insertion policies.
- We demonstrate that Juggler improves execution with minimal overhead when compared to device-wide global barrier based approaches.

2 Motivation

Many GPGPU kernels and applications used in scientific computing embed data dependences across different phases of execution [12]. In a classical CUDA implementation, the device is treated as a single computational unit, and the data hazards are prevented by using device-wide synchronization,

```
1  cudaMalloc(&a_d, matrix_size);
2  cudaMemcpy(a_d, a_h, matrix_size, H2D);
3  for (int kk=0; kk<nblocks; kk++){
4    lu0<<<1,blockSize>>>(kk, nblocks, a_d);
5    int nBlocksSub = nBlocks-kk-1;
6    if (nBlocksSub > 0){
7      fwd<<<nBlocksSub, blockSize>>>(kk, nblocks, a_d);
8      bdiv<<<nBlocksSub, blockSize>>>(kk, nblocks, a_d);
9      bmod<<<nBlocksSub*nBlocksSub, blockSize>>>(kk,nblocks,a_d);
10   }
11 }
12 cudaMemcpy(a_h, a_d, matrix_size, D2H);
```

Listing 1. Host-side pseudocode for a global barrier based CUDA implementation of LU decomposition.

which we will refer to as *global barrier*s. Listing 1 shows a pseudocode for LU decomposition (LUD), a core BLAS operation that relies on four different kernels to carry out various phases of the computation. The kernels to process the blocks in the sub-matrices need to be synchronously issued (via `cudaDeviceSynchronize()` or using the default stream) due to the computational dependences.

2.1 Task-Based Execution on GPUs

In this paper, we explore an alternative to *global barrier* based CUDA execution. We argue that treating SMs as independent stand-alone processors capable of running tasks with dependences will provide a generalized and effective solution to design- and performance-related issues with existing GPU programming models. Once applications are represented as task graphs, similar to multi-core CPU systems, SMs will be

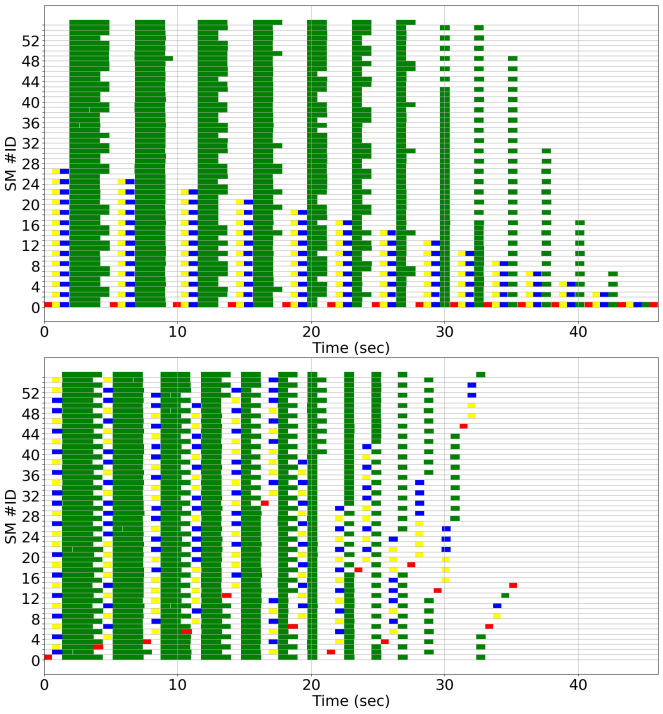

Figure 1. Per-SM TB/task execution timelines for global barriers (top) and task-based approach (bottom).

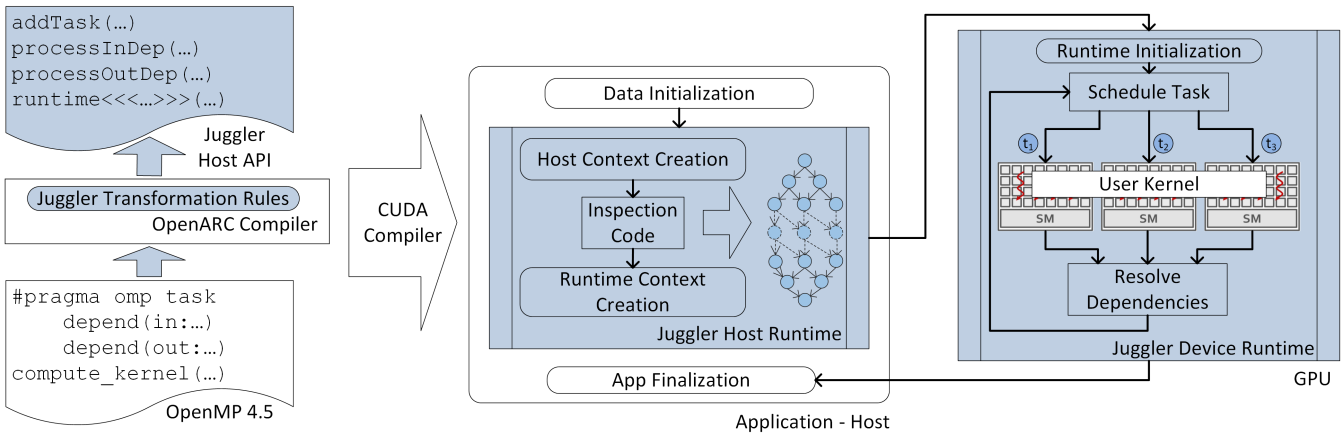

Figure 2. An overview of Juggler framework. Juggler-related components and operations are shaded.

able to consume ready tasks as they become idle, without relying on global synchronization mechanisms.

Figure 1 illustrates an example where task-based execution on an NVIDIA P100 GPU improves the total execution time for the LUD code given above. The figure depicts the timelines for each SM and shows how long they were busy with processing TBs belonging to the same or different, color-coded kernels. Global barrier based execution (shown in the top chart) requires device-wide synchronization between each kernel launch, whereas Juggler's task-based execution (shown on the bottom chart) overlaps the tasks belonging to different phases through local dependence resolution. As a result, removing global barriers increases total SM utilization and improves execution time by more than 20%, for this specific example.

As shown in the example above, managing dependences locally provides considerable performance benefits. However, shifting from current *DOALL*-based GPGPU execution paradigm with global barriers to the task-based operation with local synchronization presents several challenges.

2.2 Design Considerations

The first question that a task-based framework needs to address is *how should tasks and their dependences be represented.* Most of the notable studies on heterogeneous task-based execution [2, 4, 11, 25, 34] have created their own tasking APIs. However, re-designing existing scientific applications to use these custom APIs or specialized tasking models requires additional effort for the developers. Moreover, most of these solutions are designed for a specific class of applications. *A GPU tasking runtime based on a widely used programming model will minimize the integration effort.*

The next challenge in implementing an in-GPU task-based execution scheme is figuring out *how to track and solve dependences* between tasks. These operations require maintaining in-memory data structures, such as DAGs, and updating them on-the-fly as the tasks are processed. A few studies [16, 22, 30, 33] have proposed software-based runtimes that grab tasks from ready-queues and execute them on SMs. However, these approaches commonly rely on host-based queue population and dependence resolution mechanisms, therefore introducing a two-way task transfer bottleneck over the PCI-e bus [20]. *An all-in-GPU runtime would allow quick retrieval and insertion of the tasks as their parents are processed, without going back to the host.*

Another consideration is about *employing an efficient fine-grained synchronization mechanism between SMs.* Intra-SM synchronization primitives (i.e., `syncthreads()`) are able to provide fine-granular communication only between the threads of the same wrap or TB, and cannot be used to address the global barrier problem in question. CUDA dynamic parallelism (CUDA-DP) enables in-GPU kernel-wide synchronization between device initiated launches, however, it has been shown to severely degrade the performance [9] as the number of launches increase. Cooperative groups (CG) recently introduced in CUDA 9.0 allow creation of thread groups, which can span a subset or superset of threads belonging to a single or multiple TBs, respectively. However, CG requires extensive programming effort to implicitly handle inter-TB dependences by creating thread groups for every parent–child relation. *An ideal task-based execution scheme should utilize a peer-wise communication mechanism between desired SMs without forcing any cumulative synchronization points.*

3 Juggler Framework Overview

To address the above issues, we propose Juggler, a task-based execution framework for GPUs. Juggler implementation is entirely in software and runs on most recent GPUs. Different from previous studies, Juggler uniquely features:

- An in-GPU SW-based device-runtime with dynamic dependence tracking and resolution.
- Automated compiler support for applications using OpenMP 4.5 task directives.

Figure 2 gives an overview of our proposed framework. The compiler front-end of Juggler implements source-to-source transformations to automatically convert applications written with OpenMP 4.5 task directives to CUDA code that uses our task-based GPU runtime. This is achieved by instrumenting the input source code with calls to the Juggler host APIs. These API calls are used to (1) create/modify task execution graphs (i.e., DAGs) before the device execution, (2) initialize host/device contexts that keep necessary runtime information, and (3) launch the device runtime.

Once the Juggler-integrated application is compiled with *nvcc* and executed, a previously injected *inspection code* first creates a DAG with the input parameters that are available at application launch. The DAG is then fed into the Juggler device runtime along with other application-related context that is necessary for the execution. The device runtime is responsible for assigning tasks in the DAG to workers, executing them by calling the associated user kernels, and resolving the dependences after they are processed.

Running an application with the Juggler framework requires a minimal effort from the user (e.g., if they want to further optimize the CUDA kernel generated), provided that the application is properly embedded with OpenMP 4.5 task constructs. In addition to being transparent to such applications, Juggler also provides a general public CUDA API regardless of the programming model being used.

4 Juggler: Compiler Transformations

Juggler utilizes the OpenMP parallel programming model to bring task-based execution into the *inside* of the GPUs (i.e., inter-SM). Accelerator support is introduced in version 4.0 via `target` pragmas and the ability to declare dependences across tasks is included version 4.5. With this addition, OpenMP can enable a parallel application to specify data access precedences without relying on the parent–child relationship on which Cilk and CUDA-DP depend.

To minimize the programming effort and enlarge the scope of the applications with which Juggler can be utilized, we have implemented compiler transformations to automatically incorporate OpenMP 4.5 task-based applications with Juggler API calls. We have used OpenARC [18], an open-source research compiler that supports various heterogeneous platforms and parallel programming paradigms.

For this work, we extended OpenARC with new source-to-source transformation passes to convert OpenMP 4.5 device constructs in an input program into output CUDA codes. If `target` constructs in the input OpenMP program contains tasks, the following transformation steps are applied to generate output codes embedded with Juggler API calls:

- Identify a `target parallel` region that contains the tasks.
- Create a host context for the application.
- Generate an *inspection code* that creates *Juggler task*s and process dependences.
- Create device contexts for the application and the runtime.

4.1 Identifying Target Regions Containing Tasks

The Juggler compiler pass operates on OpenMP `target parallel` regions that contain task-generating loops, and the main objective is to *merge as many task executions (i.e., kernel launches) as possible into a single Juggler runtime instance* so that the benefits of global-barrier-free task execution are maximized *within* a single launch. The Juggler compiler pass identifies a `target parallel` region containing tasks as "eligible for execution by a single Juggler runtime instance" (i.e., *Juggler region*) only if the following two conditions hold:

(1) The computation carried within OpenMP `tasks` does not affect the instruction flow on the host: Since the tasks will be off-loaded to the GPU, conditionals within the *Juggler region* should not rely on the outcome of the off-loaded computation. Otherwise, the compiler pass will not be able to produce a runtime procedure without synchronizing with the host. To check this condition, the Juggler compiler uses a classical *reaching definition* data-flow analysis, which statically determines the definitions that may reach a given point in the code. For a given instruction, the analysis finds an earlier instruction whose target variable can reach the given one without an intervening assignment. If any reaching definitions for the instructions located within the `target parallel` region (but outside the tasks) are originating from the off-loaded tasks, then the check fails.

(2) The data flow between consecutive OpenMP tasks within the region remains in GPU memory: This will hold as long as tasks are launched from the same *Juggler region*. Otherwise, tasks from different regions will be mapped to separate Juggler runtime instances.

If the compiler finds a *Juggler region* satisfying the above conditions, the eligible region is annotated with internal directives so that later passes can perform code transformations that will explained later in this section. If the compiler fails to identify eligible regions, users can overwrite the compiler behavior by explicitly inserting the internal directive into the input source code.

4.2 Inspection Code & Application Context

After a *Juggler region* is identified for execution by the Juggler device runtime, a series of source-to-source transformations is applied to convert it into an *inspection code* that will run prior to the corresponding device launch. The main function of the inspection code is to iterate over the tasks without actually executing them and form the DAG, which will be required by the runtime later. Pre-executing the inspection code is necessary since the loop boundaries usually rely on the input data, which are not available at compile time.

As the first step of inspection code creation, the compiler pass packs all application-specific data and parameters into a generalized interface called `APP_CONTEXT`. As shown on the left side of Listing 2, this struct maintains common data

```
struct APP_CONTEXT {
  int tbSize;
  TASK* tasks;
  int totalTaskCount;
  int* dependencesCsr;
  APP_DATA appData;
};
```

```
struct TASK {
  char kernelType;
  char nChildren;
  int childrenStartIndex;
  int nDependeeParents;
  TASK_DATA taskData;
};
```

Listing 2. Juggler `APP_CONTEXT` (left) and `TASK` (right).

like the task and their dependencies as well as application-specific information, which are collected under `APP_DATA`. All the pointers and variable names that fall under the scope of the *Juggler region* are automatically declared inside `APP_DATA`, and the references in the original code are initialized and replaced with their counterparts in `APP_DATA`.

Juggler maintains two copies of `APP_CONTEXT`, host and device, to transparently handle memory copy operations, which are expressed implicitly via `target map` clauses in OpenMP. Our compiler pass generates proper `cudaMalloc` and `cudaMemcpy` calls between two contexts before and after the runtime execution according to the `map` properties. Once the host-side application context is initialized, the Juggler compiler pass identifies the tasks that are to be off-loaded to GPUs (i.e., `omp task` constructs within the *Juggler region*). For each OpenMP task, we use Juggler host-side API calls to first create a *Juggler task* and then process their dependences specified by `depend` clauses.

4.3 Task Creation

*Juggler task*s are the building blocks of the proposed execution environment. They are derived from structured blocks preceded by `omp task` pragmas, which can be either a compound statement or a function call. The pseudocode given in Listing 3 represents a task-based implementation of LU decomposition using OpenMP 4.5 *task* directives and *target parallel* regions. The code shows calls to only the first two operations of LUD, `lu0` and `fwd`, and excludes the latter two, `bdiv` and `bmod`. The dependences between the two different types of OpenMP tasks, `lu0` and `fwd`, are represented as memory ranges.

```
#pragma omp target parallel map(a[0:mSize]) private(kk,ii,jj)
{
#pragma omp single
  for (kk=0; kk<nBlocks; kk++){
    int diagBlock = kk*nBlocks*bSize+kk*bSize;
#pragma omp task firstprivate(diagBlock) shared(a) \
          depend(inout: a[diagBlock:bSize])
    lu0(&(a[diagBlock]),bSize);
    for (jj=kk+1; jj<nBlocks; jj++){
      int colBlock = kk*nBlocks*bSize+jj*bSize;
#pragma omp task firstprivate(diagBlock, colBlock) shared(a) \
          depend(in: a[diagBlock:bSize]) \
          depend(inout: a[colBlock:bSize])
      fwd(&(a[diagBlock]),&(a[colBlock]),bSize);
    }
    // Calls to bdiv and bmod are omitted.
  }
}
```

Listing 3. Task-based LUD with OpenMP 4.5 directives.

```
// Application kernel wrapper, invoked by the Juggler runtime
device__ void app_kernel(TASK* task, APP_CONTEXT* appContext,
                         RT_CONTEXT* rtContext) {
  APP_DATA appData = appContext->appData;
  TASK_DATA taskData = task->taskData;
  if (task->kernelType == KERNEL_TYPE_LU0) {
    lu0(&(appData.a[taskData.diagBlock]),appData.bSize);
  } else if (task->kernelType == KERNEL_TYPE_FWD) {
    fwd(&(appData.a[taskData.diagBlock]),
        &(appData.a[taskData.colBlock]),appData.bSize);
  }
  // Cases for bdiv and bmod are omitted.
}
```

Listing 4. Contents of the compiler-generated `app_kernel()` function for LUD.

In the task-based execution scenario shown in Listing 3, fine-grained task dependences remove the need for having separate kernel launches for each sub-target region (i.e. `lu0`, `fwd`, `bdiv` and `bmod`). Functions operating on different data regions (i.e., blocks) are mapped to separate tasks, and `omp task depend` pragmas allow the dependences to be identified with the array index ranges to which each data block is mapped. The functions called by each OpenMP task are converted into CUDA-specific `__device__` functions, and they are parallelized for GPUs if they further contain inner parallel regions (e.g., `omp parallel` pragmas). If an OpenMP `task` construct is a compound statement, the Juggler compiler will outline it as a separate function before converting it into a CUDA device function. The `__device__` functions are indirectly called later by the Juggler device runtime, unlike the *global barrier* based scenario where the kernel is explicitly launched from the host.

A *Juggler task*, as shown on the right side of Listing 2, stores information essential to dependence tracking by the runtime (`nChildren`, `nDependeeParents`, and `childrenStartIndex`), a user function identifier (`kernelType`), and the task-specific data (`taskData`). Listing 5 shows the pseudocode generated by the Juggler compiler pass, for the OpenMP 4.5 based code given in Listing 3. For each OpenMP task, the pass injects the code to initialize the task using the host API function `addtask()` along with a compiler-generated kernel identifier (e.g., `KERNEL_TYPE_FWD`). Once the task is created, the `taskData` field is populated with task-specific data that are specified in the `firstprivate` clause of the OpenMP task directive. *It is important to note that the actual function call to perform the task is not kept inside the inspection code.*

The Juggler runtime uses `kernelType` and `taskData` to identify which `__device__` function is to be called within the `app_kernel()`, as shown in Listing 4. The contents of this API function are populated by the Juggler compiler pass. The injected code supplies the application and task-specific parameters packed in `APP_DATA` and `TASK_DATA` objects to the corresponding `__device__` functions.

4.4 Dependence Processing

Juggler compiler passes require any dependence relation between tasks to be explicitly provided by the `depend` clause

```
1  // Inspection Code
2  for (kk=0; kk<nBlocks; kk++){
3   int diagBlock = kk*nBlocks*bSize+kk*bSize;
4   task=addTask(&appContext_h,KERNEL_TYPE_LU0);
5   task.taskData.diagBlock = diagBlock;
6   processInDep(task,appContext_h.appData.a,diagBlock,bSize);
7   processOutDep(task,appContext_h.appData.a,diagBlock,bSize);
8   for (jj=kk+1; jj<nBlocks; jj++){
9    int colBlock = kk*nBlocks*bSize+jj*bSize;
10   task=addTask(&appContext_h,KERNEL_TYPE_FWD);
11   task.taskData.colBlock = colBlock;
12   task.taskData.diagBlock = diagBlock;
13   processInDep(task,appContext_h.appData.a,diagBlock,bSize);
14   processInDep(task,appContext_h.appData.a,colBlock,bSize);
15   processOutDep(task,appContext_h.appData.a,colBlock,bSize);
16  }
17  // Task and dependency handling for bdiv and bmod are omitted.
18 }
19 buildCSR(appContext_h);
20 initAppContext_D(appContext_h,appContext_d);
21 initRtContext_D(appContext_h,rtContext_d);
22 cudaMalloc(&(appContext_d.appData.a),mSize);
23 cudaMemcpy(appCntxt_d.appData.a,appCntxt_h.appData.a,mSize,h2d);
24 appContext_d.appData.mSize = mSize;
25 appContext_d.appData.bSize= bSize;
26 runtime<<<nWTB,nT>>>(appContext_d,rtContext);
27 cudaDeviceSynchronize();
28 cudaMemcpy(a,appContext_d.appData.a,mSize,d2h);
```

Listing 5. Pseudocode after Juggler transformations.

of OpenMP 4.5. The `omp task depend` clause takes three types of list arguments: `in`, `out`, and `inout`. Juggler *inspection code* keeps track of these arguments for each newly created task and builds a dependence tree (DAG) on-the-fly. The example given in Listing 5 demonstrates how these dependences are identified and processed during the inspection via two Juggler host API functions: `processInDep()` and `processOutDep()`. Juggler first processes the array sections listed in the `depend` clauses to identify data regions that this task reads and writes, and then it inserts a separate arrow in the DAG (lines 6-7 and 13-15) between two tasks containing read-after-write (RAW), write-after-read (WAR), or write-after-write (WAR) dependences. Juggler supports dependences with only one-to-one matching data ranges.

After the inspection code finishes building the DAG, `buildCSR()` (line 19) is called to convert DAG into CSR representation. This function also populates a "ready-to-execute" list of tasks that are placed into worker queues prior to launch. Once all application- and runtime-related contexts are transferred into the GPU (lines 20-23), `runtime` kernel is called (line 26) to propagate the execution to device.

5 Juggler: Runtime

The Juggler runtime is launched from the host as a CUDA kernel with `APP_CONTEXT` and `RT_CONTEXT` as its inputs, and it relies on persistent TBs [3] to process the tasks supplied in the `APP_CONTEXT`.

`RT_CONTEXT`, shown in Listing 6, is application-independent, and it stores internal structures that the runtime uses, such as task queues, inter-TB signaling constructs, and other bookkeeping variables.

```
1 struct RT_CONTEXT{
2  int nWTB; // total number of worker TBs
3  int* queues; // task indices for APP_CONTEXT.tasks[] array
4  int* queueEndIndex; // queue tail pointer
5  int* IQS; // Input queue size array, one index per WTB
6 }
```

Listing 6. Juggler `RT_CONTEXT` (runtime context).

5.1 Worker TBs

The Juggler runtime utilizes persistent TBs as *worker*s that operate on dedicated task queues. In Juggler, worker TBs (*WTB*s) serve three purposes: they (1) retrieve tasks from their queues, (2) call user functions to execute these tasks, and (3) schedule new tasks as their precedents are processed.

The number of workers, `nWTB`, is specified by the dimension of the grid supplied during the *runtime kernel* launch. Because all *WTB*s need to be persistent and running altogether, `nWTB` should be:

$$nWTB \leq \#\ of\ SMs \times max\ concurrent\ TBs\ per\ SM \quad (1)$$

While the number of SMs is architecture-specific, maximum concurrent TBs per SM is determined by the combined resource usage of the Juggler runtime and the wrapped user functions for a given GPU. On the other hand, number of threads per TB, `nT`, is application-specific and usually obtained via the OpenMP parameter `num_threads`.

Figure 3 gives an overview of how *WTB*s operate in Juggler. The top part illustrates workers along with their dedicated task queues, whereas the bottom part shows the initial *tasks* list, *dependence matrix*, and *user data* that are indexable with the information held in tasks. The scenario demonstrates runtime operation with three workers:

❶ Initially, the first task, T_1, has already been inserted into task queue Q_1 of the first worker, WTB_1, by the host-side.

❷ As soon as the *runtime kernel* is launched, *WTB*s start continuously checking their corresponding task queues for

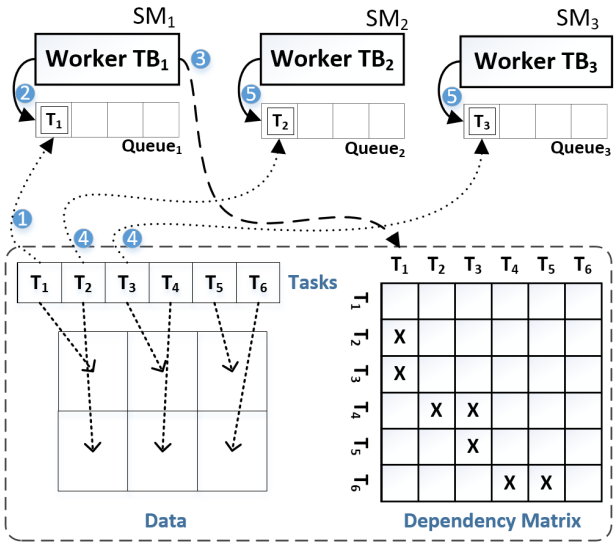

Figure 3. Overview of the Juggler runtime operation

"ready-to-execute" tasks. In this case, WTB_1 grabs T_1 and passes it to `app_kernel()` to execute it. Here, the application can fully utilize the entire SM on which WTB was working, since the entire control is transferred to the `app_kernel()`.
❸ `app_kernel()` returns, and WTB_1 gets the control back. The worker checks the dependence matrix for the children (i.e., T_2 and T_3) of the processed task T_1 and decrements the dependence counters (`nDependeeParents`) of both. Since these tasks do not have any other parents, their `nDependeeParents` become zero, and T_2 and T_3 are now ready to be executed.
❹ WTB_1 inserts the newly "freed" tasks T_2 and T_3 into the task queues of WTB_2 and WTB_3, respectively.
❺ The two idle workers, WTB_2 and WTB_3, detect the new tasks in their queues and start processing them.

The operations described between ❷ and ❺ correspond to a *work cycle*, and this cycle is repeated as long as there are ready tasks to process. The execution finishes when there are no tasks in any of the task queues. The rest of this section will explain the details of the operations summarized above.

5.2 Distributed Queues

The Juggler runtime employs a distributed queue scheme to eliminate the need for a global lock on queue retrieval. Moreover, since there is no central scheduler; therefore, each *WTB* does the scheduling decisions (i.e., task insertion and retrieval) in a distributed manner. Because no programmable HW-based inter-SM communication mechanisms are available for current GPUs, *WTB*s can interact with each other only via global memory. The Juggler runtime utilizes three data structures, `IQS[]`, `queues[]`, and `queueEndIndex[]`, which are declared under `RT_CONTEXT` (shown in Figure 6), to manage inter-WTB task exchange operations through distributed queues:

`IQS[]` (i.e., input queue size) array allows a simple but efficient signaling mechanism to let all `WTBs` know that there is a task exchange. Each WTB_i polls their corresponding index, `IQS[i]`, at the beginning of their *work cycle*. If `IQS[i]` is greater than zero, there are ready tasks; if `IQS[i]` is zero, the worker needs to keep checking as other workers may assign new tasks. To decrease the high cost of continuous memory polling in GPUs, we force WTBs to pause after a zero `IQS[]` read, by inserting a compute-intensive operation (i.e., `cosine_sleep()`). At the end of each work cycle, if a WTB places a task into the queue of another WTB, a corresponding `IQS` index is atomically incremented.

`queues[]` is the global data structure that stores queues of all *WTB*s in a flattened one-dimensional array. The queue entries are task indices for the `APP_CONTEXT.tasks[]` array, and the total size of the `queues[]` array is `nWTB`×`QUEUE_LENGTH`. Juggler redundantly sets `QUEUE_LENGTH` equal to the total number of tasks to prevent overflow.

`queueEndIndex[]` array holds the tail indices for each queue, and its elements are atomically updated whenever a task is assigned to the corresponding *WTB*. On the other hand, the head indices for each queue, **`queueStartIndex`**, are kept locally in a single `__shared__` variable by each *WTB*, and they are not placed in the global memory. Because queue head pointers are exclusively modified by the owner WTBs, making them local does not violate our consistency model (as explained later in this section).

5.3 Task Retrieval

At the beginning of each work cycle, every WTB_i reads its corresponding `IQS[i]` index and checks whether both of the following conditions hold:

⇒ $IQS[i] > 0$: A positive value indicates that another *WTB* has inserted a task to the queue of WTB_i.

⇒ $IQS[i] == queueEndIndex[i] - queueStartIndex$: This check ensures that ongoing insertions for the same queue are completed, and it prevents possible race conditions. If the IQS[i] value does not match the difference between the head and the tail of local queue (i.e., current queue size) , WTB_i will keep reading `IQS[i]` until the conditional is satisfied.

Once the checks above hold and `k=IQS[i]` is set to a positive value, the actual access to the `queues[]` array is performed in parallel using the first k threads of WTB_i. The threads first read the task indices stored in the `queues[]` array and then successively access `APP_CONTEXT.tasks[]` array using these indices. Each retrieved `TASK` object is written into the local `task_buffer[]` located in the shared memory of WTB_i. For each of the k tasks retrieved, WTB_i calls the `app_kernel()` in a loop, since only one task can be processed by a *WTB* at a time.

5.4 Dependence Resolution

After a `task` is processed by a *WTB*, the Juggler runtime checks whether the children of the `task` have any other dependences left. This check is handled via the following procedure:

(1) The first `nChildren` threads of the worker access `appContext.dependencesCsr[]` array in parallel starting from `task.childrenStartIndex`. The dependences array returns the children indices for the `appContext.tasks`.
(2) The `TASK` objects corresponding to the children indices are also retrieved in parallel. Juggler stores the total number of unsatisfied dependences in the `nDependeeParents` field of each child `TASK`. Because the parent of the children is already processed, these fields are atomically decremented once.
(3) If the return value of atomic decrement indicates that the child does not depend on any other parent tasks, then the corresponding index is saved into a shared memory buffer for the scheduling phase.

Unlike the related work on GPU task execution, Juggler's *parallel dependence resolution* exploits architecture-specific

optimizations. In step (1), the parallel accesses to `dependences_csr` array are coalesced since CSR formatted DAG representation allows indices of a parent's children to be placed adjacently in the memory. Moreover, actual `TASK` creation during the *inspection* phase follows the dependence order, and the task indices of a specific parent's children are usually not located far from each other in the memory. Therefore, in step (2) the memory coalescing unit in each SM is able to decrease the number of memory transactions required by the GPU threads to access the tasks array.

5.5 Task Insertion

Once Juggler resolves dependences and identifies new tasks that are ready to execute, *WTB*s process each of the newly freed child *TASK*s, identified by `childTaskIndex`, as follows:

(1) Using one of the policies explained in the next subsection, decide the target *WTB* (`target_w`) to which the new task should be assigned.
(2) Atomically increment `queueEndIndex[target_w]` to reserve a spot for the new task.
(3) Write `childTaskIndex` to the `queues[]` location of the target *WTB* at the index reserved above.
(4) Atomically increment `IQS[target_w]` so that the target *WTB* will see the new task on its next *work cycle*.

After all newly freed children tasks are assigned to other *WTB*s, the *WTB* that processed the parent `TASK` updates its local `queueStartIndex` value and also atomically decrements its own `IQS[i]`.

In their new *work cycle*, WTBs keep checking the `IQS` array as long as the returned value, k, is greater than or equal to zero. If all the values in `IQS` array are zero, it means all WTBs are idle. Then the first WTB noticing this sets all the other `IQS` values to -1. This forces all *WTB*s to terminate their main loop at the beginning of their next work cycle. When all WTBs exit, the `cudaDeviceSync()` right after the `runtime` kernel launch on the host side is unblocked and the execution is concluded.

5.6 Scheduling Policies

As in all other multiprocessor-based decentralized scheduling schemes, queue insertion policies play an important role in achieving load balance while also preserving data locality. Juggler employs three different queue insertion policies while deciding which *WTB* (i.e., `target_w`) to assign the next "ready to execute" task:

Juggler-LRR: [*Local round robin*] Each worker keeps a *local* counter in its private shared memory. The counter is incremented by one for each child task freed by that worker, and `target_w` is set equal to the current local counter's value.

Juggler-GRR: [*Global round robin*] A single *global* counter stored in the GPU memory is used to identify the target worker. It is atomically incremented as workers insert new tasks to others' queues. *GRR* allows a perfectly even distribution across workers, if the task execution times are similar.

Juggler-LF: [*Local first*] This policy prefers to insert the first freed task to a local queue and others to the neighboring queues in an increasing fashion. *LF* prefers locality and does not prioritize load balancing as an objective. However, the incremental behavior allows a natural distribution.

Both round-robin-based approaches are good for load balancing; however, their locality benefits are limited. On the other hand, LF enables spatial locality as well, and exploits further benefits, at the cost of reduced load balance. Performance implications of these policies are investigated in the Evaluation section.

5.7 Memory Consistency

In the CUDA execution model, memory consistency is ensured by atomic operations. Since the Juggler device runtime maintains multiple variables to manage distributed queues, atomic updates alone are not sufficient to handle possible memory races during queue operations (i.e., insertion and retrieval). When two or more *WTB*s attempt to insert two different tasks into the same queue, the *target WTB* might see an inconsistent view of its own queue. For example:

⇒ $\texttt{WTB}_1$ reserves a space in $\texttt{WTB}_0$'s queue, by atomically incrementing `queueEndIndex[0]`.
⇒ $\texttt{WTB}_2$ also reserves a space in $\texttt{WTB}_0$'s queue, by atomically incrementing `queueEndIndex[0]`.
⇒ $\texttt{WTB}_2$ writes a new task $\texttt{t}_2$ into its reserved space.
⇒ $\texttt{WTB}_2$ atomically increments `IQS[0]` to signal $\texttt{WTB}_0$.
⇒ $\texttt{WTB}_0$ reads `IQS[0]` and retrieves the task in the location reserved by $\texttt{WTB}_1$. «*Inconsistency*».
⇒ $\texttt{WTB}_1$ writes a new task $\texttt{t}_1$ into its reserved space.
⇒ $\texttt{WTB}_1$ atomically increments `IQS[0]`.

In the scenario above, $\texttt{WTB}_0$ will read an incorrect task object. It will be neither $\texttt{t}_1$ nor $\texttt{t}_2$. One approach to prevent such situations is to lock the queue of the *WTB* while assigning a new task to it, during the insertion phase. However, lock-based queue accesses have proven to be highly ineffective on GPUs [31]. Instead, Juggler ensures consistency during the task retrieval phase. As explained in Section 5.3, each WTB_i performs a second check while reading its `IQS[i]`. This check ensures that the returned *IQS* value matches the difference between the *global* `queueEndIndex[i]` and the

```
1  // Inspection code related functions
2  initAppContext_H(appContext_h,nTasks,nEdges);
3  int addTask(appContext_h,kernelType);
4  addDependency(appContext_h,srcTaskIndex,dstTaskIndex);
5  processOutDep(appContext_h,taskIndex,...);
6  processInDep(appContext_h,taskIndex,...);
7  // Device initialization related host functions
8  initRtContext_D(appContext,rtContext);
9  initAppContext_D(appContext_h,appContext_d);
10 // Main entry point to the Juggler runtime
11 __global__ void runtime(appContext,rtContext);
12 // App kernel, generated during transformation.
13 __device__ void  app_kernel(task,appContext,dynContext);
14
```

Listing 7. Juggler Host & Device API.

Application	Description	# tasks	Δ parallelism
DT-WARP	*Dynamic time warping:* Two time series are warped to find an optimal match. The algorithm creates an intermediate 2D matrix and dynamically iterates over it.	5184	1-72
HEAT-SIM	*Heat simulation*: Simulation of heat over elements in a 2D surface. Calculates a single time step and relies on elements calculated in previous and current steps.	8100	1-90
INT-HIST	*Integral histogram:* A progressive image processing algorithm to calculate histograms. It is based on wavefront propagation.	8100	1-90
JACOBI	*Jacobi iteration:* Time-based iteration of a five-point stencil over a 2D matrix. Involves two separate functions working on the same data in the dependence order.	3600	900
LU-D	*LU decomposition:* A common operation in linear algebra, which factors a matrix as the product of its lower and upper triangular matrices.	1240	1-196
SA-TABLE	*Summed area table:* A popular imaging algorithm that is used to generate sum of values in a rectangular subset of grid.	8100	90
SM-W	*Smith Waterman:* Bioinformatics-based string matching algorithm to find similar patterns between two input gene sequences.	5184	1-72

Table 1. Kernels used in our evaluation

local `queueStartIndex` values. If the difference and the *IQS* values do not match, this indicates that concurrent writes are in progress, and the *WTB* needs to wait until the values match. During our experiments with repetitive runs, we have ensured that the control scheme above successfully ensures memory consistency of Juggler's runtime operations.

5.8 Runtime API

Juggler is transparent to programmers and does not require them to explicitly call the Juggler interfaces. However, Juggler can also be incorporated by any application manually via the APIs shown in Listing 7. These operations have previously been explained in Section 4, and we leave the listing as a reference for users.

6 Evaluation

This section evaluates the performance of the Juggler runtime. We explain the tested platform and the applications, and then we elaborate on performance analysis.

6.1 Experimental Setup

Applications: We used seven commonly used kernels in scientific computations. All of them employ functional and/or data-based dependences. Table 1 lists these kernels along with their brief descriptions and the total number of the tasks they employ in our runs. The last column shows the minimum and maximum amount of task-level (i.e., SM) parallelism that an application can exploit. For all applications, we used the largest amount of global memory permitted by `cudaMalloc`.

Methodology: We have tested two versions of each application:

(1) *Global barriers*: This is the baseline version in native CUDA and uses separate synchronous kernel launches to ensure that data dependences are satisfied. We have adopted the implementations previously published in [7] and compared them against Juggler.

(2) *Juggler*: For this version, we first ported the kernels to use OpenMP 4.5 tasking and off-loading constructs. During this process, we have also benefited from the OpenMP based implementations presented in [19]. Then, we ran our modified version of OpenARC to convert the OpenMP code into CUDA. We tested Juggler with three different insertion policies, as described in the previous section: *Juggler-LRR*, *Juggler-GRR*, and *Juggler-LF*.

Reported results for *Juggler-{LRR,GRR,LF}* include kernel execution times plus *all host- and device-related overhead additionally introduced by Juggler*. This overhead includes the times for inspection code execution, CSR graph creation, application and device context initialization & transfers, task graph transfer, device-side dependence resolution, and queue operations. We excluded the original data staging and copy phases of the applications from the execution times since they are common to both global barriers and Juggler versions.

Platform: We compiled both versions using `nvcc` of CUDA 8.0. We ran our experiments on a single NVIDIA Tesla P100 GPU with 12 GB HBM2 global memory and 56 SMs. We have set total number of workers, $nWTB$, to a multiple of the number of SMs, and the cofactor is determined based on the resource usage of each application. We executed every benchmark five times for each of the four versions and reported the average. We used `nvprof` and wall clock to measure timings.

6.2 Results

Execution time: As our main experiment, we measure the speedup obtained by Juggler. We take the global barrier approach as the baseline and report the speedup results in Figure 4. The numbers above global barrier bars are for reference only and they are the actual kernel execution times of the global barrier version, in seconds. We also provide the task distribution variances across Juggler policies in Figure 5 for a better understanding and justification of the main results. The latter figure shows the lowest, highest, and average values of the differences between the workers' maximum and

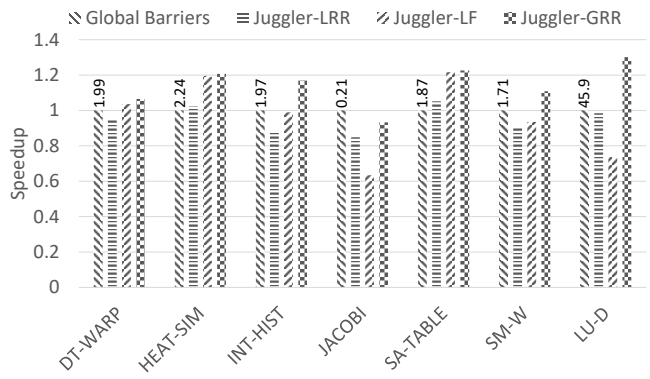

Figure 4. Speedup obtained by the Juggler runtime

minimum number of tasks assigned, across 5 runs. Lower and closer values mean that the scheduler consistently balanced the workload across workers, whereas higher and distant values indicate the load was not balanced and the results vary greatly across each run.

Overall, Juggler-GRR improves performance up to 31% when compared to the global barrier approach, and it falls under the baseline only for JACOBI, with a 6% slowdown. Multiple observations have been derived from the results:

Observation 1: Juggler-GRR demonstrates stable performance in all cases by providing an optimal load distribution. GRR exploits the full benefits of task-based execution by keeping all workers busy most of the time, despite varying amounts of parallelism. Increased utilization demonstrates the validity of the initial motivation we built against global barriers. On the other hand, LF is better than LRR in half of the cases. Although LRR performs better in load distribution, locality benefits provided by LF are beter. That being said, the optimal load balance (i.e., GRR) always provides the best results, regardless of locality.

Observation 2: JACOBI is the only exceptional case where Juggler provides no performance improvement. Each JACOBI iteration employs two types of kernels, *copy* and *compute*, and all the TBs (i.e., tasks) within each kernel can execute in parallel. The dependences are only across different kernels operating on the same block; therefore, the number of tasks that can be executed in parallel is at least 900, at all times. JACOBI is an example where task-based execution provides no benefit, and 6% slowdown by GRR corresponds mostly to the in-device task-management overhead. LF performs significantly slower, because batch insertion of tasks causes a biased inclination towards local-WTB assignment, hence eventually increasing the load imbalance.

Observation 3: LRR and LF perform worse than global barriers in three other cases: DT-WARP, INT-HIST, and SM-W. These applications, differently from others, employ additional array accesses (i.e., time&gene sequences and a bin for histogram values), which in turn puts additional traffic on the interconnect. The global memory operations on which Juggler runtime relies to manage communications are adversely affected by the increased traffic, therefore causing additional overhead. LU-D, on the other hand, employs four different kernels with diverse computational requirements along with a vastly varying amount of parallelism. This causes a significant slowdown for LF, which may *overassign* tasks to a specific WTB and cause a crucial bottleneck for the remaining dependences. GRR is still able to improve the performance over global barriers, in these cases as well.

L2 cache misses: To better understand how Juggler improves locality, we also measured the L2 cache write/read miss rates, as shown in Figure 6, using NVIDIA's `nvprof` utility. We used the write&read variations of two event types:

- `l2_subp0_{write|read}_sector_misses`
- `l2_subp0_total_{write|read}_sector_queries`

The results show that, despite the contamination caused by Juggler runtime operations (i.e., global memory accesses), LF policy clearly reduces percentage of read misses. On the other hand, GRR shows a moderate amount of improvement over global barriers due to the temporal locality exploited by child–parent dependences. As mentioned previously, since the amount of maximum parallelism in JACOBI is significantly larger, the benefits of locality are not observable for this application.

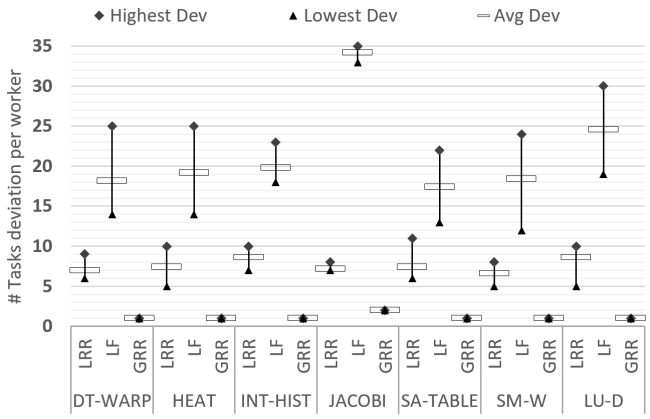

Figure 5. Per worker task load deviation

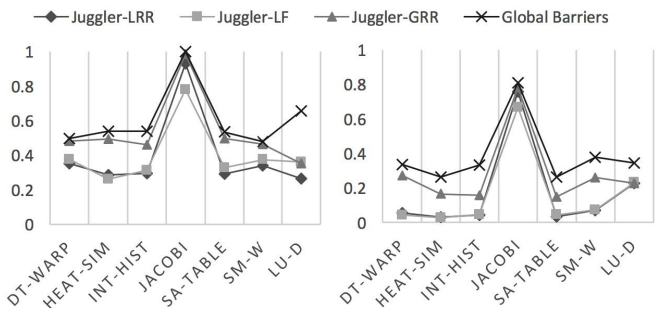

Figure 6. L2 cache miss rates for writes(left) and reads(right).

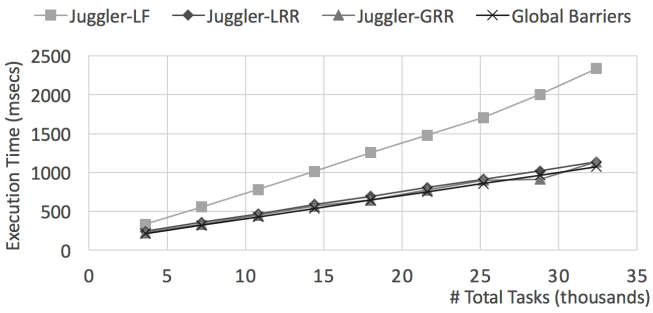

Figure 7. Jacobi execution as the task counts increase

Scalability: We have used JACOBI iteration to observe the effects of increasing total task count using a larger number of time iterations. For other applications, increasing the number of tasks required more data to be allocated; however, this was not possible due to device limits. We varied the number of tasks for JACOBI from 3,600 to 32,400 and showed the execution times in Figure 7. We observe that LRR and GRR are on par with global barriers and are able to maintain performance at a linear slope. On the other hand, the imbalance caused by LF worsens as the task counts increase.

7 Additional Related Work

Early studies [4, 13] developed high-level task management frameworks to distribute a workload across heterogeneous processors including CPUs and GPUs. These schemes stay at a coarse-granular level and treat the GPU as a single processing entity. Finer granular scheduling solutions [8, 14, 15, 23, 27] proposed schemes that assign tasks into multiple streams attached to CPUs and GPUs. While these solutions exploit concurrent kernel execution on GPUs via asynchronized streams and pipelines, they do not schedule tasks at the SM level, hence failing to exploit a true SM-aware task-based execution scheme. Another class of study, warp specialization [5, 6] addresses the issues caused by intra-SM barriers (`syncthreads()`), inter-SM level global synchronization is left unaddressed.

Study in [17] proposed locality-aware hardware TB schedulers but failed to consider the dependences. Other HW solutions [1, 21, 28, 29] introduced special hardware to manage dependences. However, the cost of maintaining dependence-related data structures in HW for actual scientific applications would be impractically high as the data sizes go beyond the limits of the simulation environment. A few software-based approaches [9, 21] have adapted existing CPU-oriented task models like Cilk and CUDA-DP to represent dynamically created in-GPU tasks. Others [12, 30] built compiler techniques to detect dependences statically via loop analysis.

Only a handful of works [10, 16, 22, 33] have brought the task execution paradigm into a finer granular level where each SM is treated as a stand-alone processing unit. These studies operate in a consumer–producer fashion where the scheduler running on the host side sends tasks to the GPU as their dependences are resolved. The performance of this approach is highly limited by the slow memory transfers over PCI-e bus. Juggler, on the other hand, operates on HBM2 memory, which is tens of times faster.

In-GPU dependence resolution has been considered by only two studies [25, 26], to the best of our knowledge. These papers propose loosely defined abstract programming techniques and manual code modifications, which are limited to one or two applications. Therefore, they are far from being a practical frameworks for real-life applications.

8 Conclusion

In this study we have proposed Juggler, a new, dynamic task-based execution scheme for GPGPU applications with data dependences. Different from previous studies, Juggler implements an in-GPU runtime for applications with OpenMP 4.5–based dependences. The runtime uniquely employs in-GPU dependence resolution and task placement. Our experimental evaluation of seven scientific kernels with data dependences on an NVIDIA Tesla P100 GPU showed that Juggler improves kernel execution performance up to 31% when compared to global barrier based implementation.

Our results demonstrate that the conventional GPGPU programming paradigms relying on grid-based execution with global synchronization can be replaced with DAG-based, dependence-aware task processing to increase the performance of scientific applications.

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC05-00OR22725. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

This work was also partially supported by NSF Grants CCF-1423108, and CCF-1513201 given to University of California, Riverside. We thank Matt Martineau and Simon McIntosh-Smith from University of Bristol for providing their code for some of the OpenMP based kernels.

References

[1] Amir Ali Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. 2017. Wireframe: Supporting Data-dependent Parallelism Through Dependency Graph Execution in GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*.

[2] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. 2011. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS'11)*.

[3] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics (HPG '09)*.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing (Euro-Par '09)*.

[5] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*.

[6] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging Warp Specialization for High Performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*.

[7] Mehmet E. Belviranli, Peng Deng, Laxmi N. Bhuyan, Rajiv Gupta, and Qi Zhu. 2015. PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*.

[8] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei W. Hwu. 2015. Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*.

[9] Guoyang Chen and Xipeng Shen. 2015. Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO '15)*.

[10] L. Chen, O. Villa, S. Krishnamoorthy, and Guang R Gao. 2010. Dynamic load balancing on single-and multi-GPU systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '10)*.

[11] T. Gautier, J. V. F. Lima, N. Maillard, and B. 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 (IPDPS '13)*.

[12] R. Govindarajan and Jayvant Anantpur. 2013. Runtime Dependence Computation and Execution of Loops on Heterogeneous Systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*.

[13] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. 2010. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *Euro-Par 2010-Parallel Processing (Euro-Par, 10)*.

[14] Huynh Phung Huynh, Andrei Hagiescu, and Rick Siow Mong Goh. 2012. Scalable framework for mapping streaming applications onto multi-GPU systems. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12)*.

[15] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. 2014. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*.

[16] Scott J. Krieder, Justin M. Wozniak, Timothy Armstrong, Michael Wilde, Daniel S. Katz, Ian T. Foster, and Ioan Raicu. 2014. 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 '14)*.

[17] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*.

[18] Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*.

[19] Matt Martineau, Simon McIntosh-Smith, Carlo Bertolli, Jacob, et al. 2016. Performance analysis and optimization of Clang's OpenMP 4.5 GPU support *(PMBS '16)*.

[20] Pinar Muyan-Özçelik and John D. Owens. 2016. Multitasking Real-time Embedded GPU Computing Tasks. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '16)*.

[21] Marc S Orr, Bradford M Beckmann, Steven K Reinhardt, and David A Wood. 2014. Fine-grain task aggregation and coordination on GPUs. In *41st International Symposium on Computer Architecture (ISCA '14)*.

[22] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*.

[23] Daniel Sanchez, David Lo, Richard M Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic fine-grain scheduling of pipeline parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on (PACT '11)*.

[24] Fengguang Song, Asim YarKhan, and Jack Dongarra. 2009. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*.

[25] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: Dynamic Scheduling on GPUs. In *ACM Trans. Graph.*, Vol. 31.

[26] Stanley Tzeng, Brandon Lloyd, and John D Owens. 2012. A GPU Task-Parallel Model with Dependency Resolution. *Computer* (2012).

[27] U. Verner, A. Schuster, and M. Silberstein. 2011. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*.

[28] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. 2016. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*.

[29] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*.

[30] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU Through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*.

[31] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '10)*.

[32] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*.

[33] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. 2017. 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 '17)*.

[34] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, and Wenguang Chen. 2017. Versapipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*.

A Artifact Description

A.1 Abstract

Artifact described in this section includes the source code for Juggler host and device runtimes and the sources for the applications used in our evaluation.

The scripts to compile the source code, generate inputs, execute binaries, validate results, and parse the outputs are also included in the artifact and explained below in detail.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Hardware:** NVIDIA Tesla P100 or newer GPU with at least 12GB of memory, and x86-64 modern CPU.
- **Program:** CUDA 8.0 APIs and above.
- **Compilation:** NVIDIA `nvcc` version 8.0 and aboce.
- **Binary:** CUDA host(x86-64) and device executable. Linux binary is included. Source code and scripts to regenerate the binaries are also included.
- **Data set:** Dynamically generated data, prior to the execution.
- **Runtime environment:** CUDA 8.0 APIs and drivers. They are included with CUDA 8.0 toolkit distribution.
- **Output:** Verification results and detailed timings such as execution times and runtime overhead.
- **Experiment workflow:** Linux `bash` scripts.
- **Publicly available?:** Yes

A.2.2 How software can be obtained (if available)

The source code for Juggler host & device runtime and the experimented applications (both baseline and Juggler-integrated version) can be accessed via:

https://code.ornl.gov/fub/juggler

A.2.3 Hardware dependencies

We performed our experiments on an NVIDIA P100 GPU. While Juggler is compatible with Kepler architecture and later, the provided artifact will work only for Pascal architecture or later, with at least 12GB of memory.

A.2.4 Software dependencies

CUDA 8.0 toolkit is required for compilation and profiling. At the time of CUDA driver installation, GCC 5.2 was installed on the system. To run the scripts, `bash` and `egrep` are sufficient. The scripts assume that `CUDA_HOME` is properly set to the installation directory of CUDA. The default location:

```
$ export CUDA_HOME=/usr/local/cuda
```

A.2.5 Datasets

Due to large sizes, each application dynamically generates and populates the input dataset, as part of the initialization.

A.3 Installation

Clone Juggler from the ORNL repository:

```
$ git clone https://code.ornl.gov/fub/juggler.git
$ export JUGGLER_HOME=$(pwd)/juggler
```

A.4 Experiment workflow

To repeat the main experiments presented in the evaluation section, we have created a script named **exp**. It is located under `$JUGGLER_HOME/build` and the parameters to the script are as follows:

```
exp {scriptMode} {outFilePrefix} {runGB} {runJG} {nRuns} {nProfRuns}
```

To repeat the main experiments presented in Figures 4 and 5, run:

```
$ cd $JUGGLER_HOME/build
$ bash exp 0 output 1 1 5 1
```

When run with `scriptMode=0`, `exp` script compiles each application for each scheduling policy; then runs each of them five times (i.e., `nRuns=5`) for global barriers (i.e., `runGB=1`) and also five times for the Juggler integrated versions (i.e., `runJG=1`). It also performs an additional run with profiling enabled (i.e., `nProfRuns=1`). The program output for each application is written into separate files prefixed by `outFilePrefix`.

A.5 Evaluation and expected result

Execution of the `exp` script with the parameters above will produce a series of output files, named `output.$APPNAME` and `output.PROF.$APPNAME`, for each application. The same `exp` script can also be used in parsing mode (i.e., `scriptMode=1`) to parse these output files and combine the values from all runs in a *tab separated value* (TSV) format.

1. **To list the kernel execution times** (i.e., the values used to draw Figure 4) for all seven applications, in separate columns:

   ```
   $ bash exp 1 output execTime 2 formattedResults1.tsv
   ```

   Parsed values will be written into `formattedResults1.tsv`, in a tabular format. There will be seven columns in total, one for each application. The number of rows in the tsv file will be equal to $nRuns \times 4$ (i.e., 20, when main experiment is run five times). The first 15 rows will be for *LRR*, *GRR*, and *LF*, respectively, in groups of five. The last five rows will be for global barriers.

2. **To see verification results** against serial execution:

   ```
   $ bash exp 1 output check 2 formattedResults2.tsv
   ```

   The parsed output will be written into the `formattedResults2.tsv` file, and the values will be either SUCCESS or FAIL.

3. **To parse task load deviations**, run:

   ```
   $ bash exp 1 output minTaskLoad 2 formattedResults3.tsv
   $ bash exp 1 output maxTaskLoad 2 formattedResults4.tsv
   ```

   Figure 5 in the paper is drawn as "HIGH-LOW-CLOSE" chart in Microsoft Excel, where HIGH, LOW and CLOSE are the highest, lowest and average values, respectively, of the differences between `maxTaskLoad` and `maxTaskLoad`, across five runs.

A.6 Experiment customization

Number of runs for each type of run in the main experiment can be modified by changing the input parameters of `exp` script, as explained in section A.4.

Additionally, if only a specific subset of applications is desired to be tested, the values in the bash array, named `$APPS`, in the `exp` script can be modified.

More information can be parsed from the output files by providing the *key string* and *column number* in the value parser (i.e., *"exp 1"*). A few examples:

1. Cache miss data:

```
$ bash exp 1 output.PROF write_sector_misses 7  formatted.tsv
$ bash exp 1 output.PROF write_sector_queries 7  formatted.tsv
$ bash exp 1 output.PROF read_sector_misses 7  formatted.tsv
$ bash exp 1 output.PROF read_sector_queries 7  formatted.tsv
```

2. Host runtime and inspection loop overhead breakdown:

```
$ bash exp 1 output initAppContext_H 2  formatted.tsv
$ bash exp 1 output initAppContext_D 2  formatted.tsv
$ bash exp 1 output initRtContext_D 2  formatted.tsv
$ bash exp 1 output inspectionLoop 2  formatted.tsv
$ bash exp 1 output buildCSR 2  formatted.tsv
```

3. Total application runtime, including user data initialization and transfers:

```
$ bash exp 1 output totalTime 2  formatted.tsv
```

Compiling and running a single application: In our test suite, applications are distinguished with compiler directives to optimize the resource usage for the ones that share common kernels. Similarly, the Juggler runtime requires a recompilation if internal runtime parameters (e.g., scheduling policy) are changed.

1. To recompile Juggler for the desired application and scheduling policy:

```
$ cd $JUGGLER_HOME/build
$ bash switchAPP {DTW|HEAT|INT|JACOBI|SAT|SW|LUD} {LRR|GRR|LF}}
```

2. To run the compiled application with the Juggler runtime:

```
$ cd $JUGGLER_HOME/build
$ ./OMP_CUDART -n {matrix_size} -b {block_size} -d {1|2} [-c]
```

   The `-d` parameter indicates the run mode, which is 1 for Juggler, and 2 for global barriers. The optional `-c` parameter enables verification against serial version and compares the two outputs. By default, `-c` is enabled.

A.7 Notes

For up-to-date instructions, please follow the README file under the root directory of the repository.