GOPipe: A Granularity-Oblivious Programming Framework for Pipelined Stencil Executions on GPU

Chanyoung Oh
University of Seoul
alspace11@uos.ac.kr

Zhen Zheng
Alibaba Group
james.zz@alibaba-inc.com

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

Jidong Zhai
Tsinghua University, BNRist
zhaijidong@tsinghua.edu.cn

Youngmin Yi
University of Seoul
ymyi@uos.ac.kr

ABSTRACT

Recent studies have shown promising performance benefits when multiple stages of a pipelined stencil application are mapped to different parts of a GPU to run concurrently. An important factor for the computing efficiency of such pipelines is the granularity of a task. In previous programming frameworks that support true pipelined computations on GPU, the choice has to be made by the programmers during the application development time. Due to many difficulties, programmers' decisions are often far from optimal, causing inferior performance and performance portability.

This paper presents GOPipe, a granularity-oblivious programming framework for efficient pipelined stencil executions on GPU. With GOPipe, programmers no longer need to specify the appropriate task granularity. GOPipe automatically finds it, and dynamically schedules tasks of that granularity for efficiency while observing all inter-task and inter-stage data dependencies. In our experiments on six real-life applications and various scenarios, GOPipe outperforms the state-of-the-art system by 1.39× on average with a much better programming productivity.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; • **Computer systems organization** → **Parallel architectures**; • **General and reference** → *Performance*.

KEYWORDS

Programming Framework; GPU; Optimizations

ACM Reference Format:
Chanyoung Oh, Zhen Zheng, Xipeng Shen, Jidong Zhai, and Youngmin Yi. 2020. GOPipe: A Granularity-Oblivious Programming Framework for Pipelined Stencil Executions on GPU. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414656

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414656

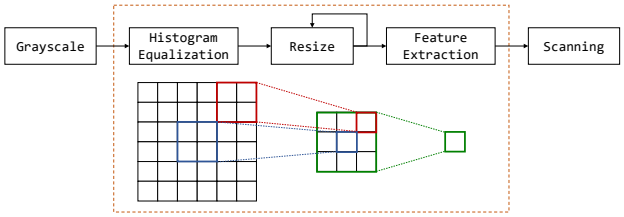

Figure 1: Pipeline of `Face Detection` with a stencil dependence. Computation of a pixel depends on a tile of pixels in the previous stage.

1 INTRODUCTION

Pipelined stencil applications are stencil applications consisting of multiple stages of computations, and in each stage, the computation follows a stencil pattern (i.e., a kernel applies to each tile of pixels). The output of a stage is the input of the next stage. Fig. 1 shows `Face Detection` pipeline [13], which consists of one recursive stage and four other stages to detect faces in an image. The `Resize` stage and the `Feature Extraction` stage compute the stencil patterns of 2×2 and 3×3, respectively.

As a fundamental form of parallelism besides data parallelism, pipeline parallelism is inherent in a large array of applications in various domains, ranging from face detection to network packet processing, graph rendering, video encoding and decoding, and various streaming data processing [8, 11, 17, 19, 24, 26, 31, 34, 39]. An effective capitalization of pipeline parallelism is essential for meeting their relentless demands for higher throughput or responsiveness in modern computing.

In recent years, the problem has attracted strong interest [8, 33, 40] in using Graphics Processing Units (GPUs) to accelerate pipeline applications. As a processor with tremendous parallel computing power, GPU shows large potential for computing accelerations. Meanwhile, its higher-level programming models make it much easier for general software developers to program than alternative choices (e.g., FPGA).

Despite promising performance shown in recently developed programming frameworks, an important barrier stands, *efficient support of flexible granularities of tasks on GPU for each stage of a pipeline*. The barrier has been preventing pipeline application developers from tapping into the full potential of GPU and the existing programming frameworks.

Here, a **task** of a pipeline stage conceptually refers to a subset of the result values that the stage needs to produce[1]. It is the scheduling unit of a stage. The processing of a task may involve one or more GPU threads, but usually no more than a thread block.

For a stage in a pipeline, there are often many valid granularity choices for a task. Consider an image smoothing stage that produces a smoothed image with each pixel equaling the average of that pixel and its neighbors in the image output by an earlier stage in a pipeline. A task of the smoothing stage could be as large as the entire image to produce, or as small as a tile or even just a single pixel in the smoothed image.

Task granularity definition is essential for the performance of pipeline applications on GPU as it sits at the center of a three-way tradeoff, the tradeoff among parallelism, synchronization overhead, and data locality. A large granularity could limit the amount of parallelism and hence leaves GPU underutilized. While a fine granularity can provide sufficient parallelism, it may cause more dependency among workers in different stages and hence more synchronizations. It could also degrade data locality as spatially adjacent data elements may be put into different fine-grained tasks processed by different workers. Our experiments show that different granularities on GPU could lead to over 6× differences in performance (details in Sec. 5.2). Therefore, for a programming framework to better support pipeline computations on GPU, it is important to have the ability to transparently adapt task granularities. The adaptation needs to be across applications as well as the different usage scenarios of an application. The suitable granularity often changes with usage scenarios; for an application detecting faces in a stream of input images, for example, the best granularity differs for images of different resolutions and also input image streams of different arrival rates.

No prior frameworks for pipeline executions on GPU provide such a support. The few existing frameworks that support flexible task granularities are primarily for pipelines built on CPU [23, 29]. Halide [29], for instance, uses GPU as a monolithic accelerator, each time running the kernel of only one pipeline stage on GPU; it may fuse some kernels into one but supports no true cross-stage pipelined computations on GPU. PolyMage [23] has no support of GPU. Existing true GPU pipeline programming frameworks [33, 40] use GPU more effectively, but they are all rigid in handling task granularities. It is challenging to support flexible task granularity in true pipelined executions on GPU in a user-transparent way when the stages have stencil dependence. As the execution model is not kernel-by-kernel, it cannot rely on kernel boundaries for implicit synchronizations but demands efficient scheduling for coordinating tasks of various granularities in different stages. A naive approach would require users to write substantially different code for a stage and for dealing with the interplay with other stages for different task granularities. And the performance is often disappointing. NVIDIA's Cooperative Groups [15], with which more flexible synchronization is possible, cannot help either if a stencil dependency exists as a thread would then need to belong to multiple groups which is not supported. Even if multiple task granularities are supported, to select the best ones for each stage, a programmer has to develop many versions of code of different granularity definitions and run them to identify the good choices for each usage scenario. The process is tedious and time-consuming; often programmers just pick a granularity most intuitive or convenient for programming and use it for all usage scenarios, resulting in poor performance as Sec. 5.2 will show.

This work proposes a *granularity-oblivious* concept for the design and implementation of GPU pipeline programming frameworks. A pipeline programming framework is *granularity-oblivious* if code written in the framework can work on an arbitrary task granularity, and the framework itself can automatically select and adapt the granularity to suit each usage scenario of the application. The process is transparent to application developers in the sense that no code refactoring is needed to redefine the corresponding computations or any other parts of the program.

The key challenge for creating a *granularity-oblivious* framework for GPU is on the creation of a mechanism that can provide the flexibility while incurring minimum runtime overhead. The problem is especially challenging on GPU due to its massive parallelism: One pipeline stage often runs concurrently by hundreds or thousands of GPU threads; special designs are needed for coordination among stages and management of task queues and other shared resources.

This paper presents our solution. A key idea is *adaptive trigger-based scheduling*, which consists of the designs of three schemes for tracking and notifying a stage of the readiness of tasks to resolve the dependencies at various task granularities on the fly. It offers the key to our efficient solution when coupled with three other techniques. One is *automatic task grouping*, which automatically generates code for a larger granularity from the code by programmers that handles tasks of the smallest granularity. The second is *granularity autotuning*, which automatically selects the scheme best suiting an application usage scenario with the highest efficiency. The third idea is *reference-type task representation*, which decouples the representation of a task from the data it needs to process to significantly reduce the redundant data copies among tasks.

Based on a recent pipeline framework VersaPipe [40], we put all the techniques together and create a new pipeline framework called *GOPipe* (Granularity-Oblivious Pipeline framework). To our best knowledge, GOPipe is the first GPU pipeline programming framework that automatically determines the appropriate granularity of tasks in each stage of a pipeline and schedules them dynamically without interventions from developers. GOPipe removes the effort current frameworks require programmers to pay to find good task granularities, while at the same time leading to significant performance gains. Compared to the fixed granularity schemes hard-coded in existing programming frameworks, GOPipe improves the performance of several benchmarks and real-world pipeline programs by 1.39× on average.

Overall, this work makes the following major contributions:

- It develops GOPipe, the first *granularity-oblivious* pipeline programming framework for GPU.
- It proposes *adaptive trigger-based scheduling* to dynamically schedule tasks of various granularities to achieve high efficiency on GPU.

[1]In implementations of pipeline frameworks, a task is typically embodied with the set of data the stage needs to process to produce the result values in the task. The term "task" sometimes takes this meaning (e.g., in the context of "enqueue" or "dequeue" a task).

- It empirically demonstrates both the productivity and the performance benefits of GOPipe.

2 BACKGROUND AND TERMINOLOGY

This section provides the background and terminology that are necessary for readers to follow the rest of the paper.

Persistent Threads. Persistent threads [7, 14, 33, 40] are used in most GPU pipeline frameworks. This work uses it as well. In persistent thread executions, the number of GPU threads created is limited such that all of them can be active and no one is in GPU waiting queues. The kernel function is written such that each thread iterates through a while loop and continuously tries to grab more work to do after it completes one piece of work. Persistent threads avoid frequent kernel launches and at the same time make global synchronizations possible.

VersaPipe. Our solution builds on VersaPipe [40], an open-source GPU programming framework for pipeline applications. It features a support of a variety of pipeline execution models on GPU:

- **Run to completion (RTC).** RTC organizes all stages of a pipeline into a single kernel. The execution of a thread goes through all the stages from the beginning to the end.
- **Kernel by kernel (KBK).** In KBK, multiple kernels are used to realize a pipeline program, with each kernel implementing one or multiple stages; these kernels are invoked one after another.
- **Megakernel.** Megakernel [33] puts all pipeline stages into one large kernel. Implemented with persistent threads, each MegaKernel thread works in a while loop, grabbing new work items continuously; here, one workitem is the operation of one stage of the pipeline on a task. Which stage to use is decided by a runtime scheduler.
- **Coarse Pipeline.** In this model, the different stages of a pipeline are assigned to different SMs, forming an SM-level pipeline.
- **Fine Pipeline.** This model is similar to coarse pipeline, except that the resource allocation is in a much finer grain; multiple stages may share one SM.

These models have different pros and cons on the utilization of GPU computing resource. For a pipeline program written in VersaPipe API, VersaPipe automatically finds out the best pipeline execution model for the program.

VersaPipe eases the creation of pipeline execution models, but not the selection of task granularity. It requires the programmer to choose the task granularity for every pipeline stage, and to code both the operations of each stage and the scheduling among them accordingly. As a result, changing the granularity of one stage would require lots of code changes.

Terminology This part introduces some terms used in later discussions. A *task* refers to the work that produces a set of the result values of a pipeline stage. A *unit task* is a task that produces the minimal set of the result values, or *unit output*. For image processing, for instance, a *unit task* of a stage is typically the work that produces one pixel in the output of that stage. A *unit input* is the input that a *unit task* uses to produce a *unit output*. *Unit input dimension* is the size of a *unit input*. For example, the *unit output* of the `Resize` stage in Fig. 1 is a single pixel in the output of the stage, and its *unit input* is a set of 2×2 pixels in the output of the stage prior to `Resize`; the *unit input dimension* is 2×2. A *unit stride* is defined as the distance between the *unit inputs* of two adjacent *unit tasks*. For instance, the *unit stride* shown in Fig. 1 is $(2, 2)$ as there is no overlap between two adjacent *unit inputs* (i.e., 2×2 tiles). *Task granularity* is the size of a task, which could consist of a group of adjacent *unit tasks*. The size of the entire output of a stage is called the *stage output dimension*. In general, to produce the entire result of a stage, a *unit task* needs to be invoked as many times as the *stage output dimension*.

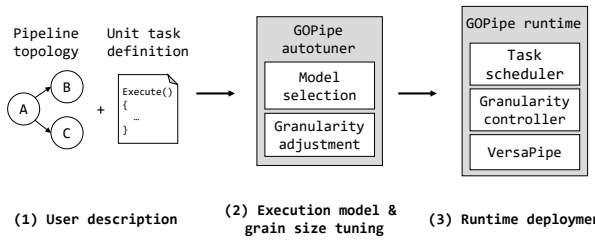

Figure 2: Overall structure and workflow of GOPipe.

3 OVERVIEW OF GOPIPE

GOPipe is the first programming framework that enables *granularity-oblivious* programming of pipeline applications on GPU. Fig. 2 shows its workflow and major components.

When developing a pipeline program in GOPipe, a user needs to specify only the structure of the pipeline and the computation of each stage. She has no need to worry about what granularity is the best to use for a stage, or how to schedule the tasks. GOPipe takes care of them automatically.

At a high level, GOPipe works as follows. First, GOPipe takes as input the pipeline structure in a form of directed acyclic graph (DAG), where the nodes represent the pipeline stages and the edges represent the task queues between stages. For each stage in a DAG, users describe the computation for a *unit task*, and specify parameters such as stage output dimension, unit stride, as well as unit input size. If not all data in a unit input are used, the stencil dependency pattern within the tile should be given. (Although it is possible to automatically recognize the patterns from the code, the current implementation of GOPipe focuses on runtime support rather than static code analysis; so we leave that part for the future.)

Then, GOPipe parses the specified pipeline structure, instantiates and initializes the stages. It employs a *reference-based task* representation and includes the buffer allocation for each stage. Based on user-specified unit task code and parameters as well as the dependency patterns, GOPipe selects the appropriate task grain size for each of the pipeline stages and the execution model via its *autotuner*. If the selected grain size is not unit task size, GOPipe materializes the new grain size through automatic task grouping.

Finally, the framework launches GPU kernels in form of persistent threads. A Cooperative Thread Array (CTA) dequeues a task from the task queue, and the task is then processed by the threads in that CTA. After processing, the CTA does not enqueue the newly produced task into the task queue of the next stage, but writes the

```
1: void FeatureExtraction::execute(TaskItem data) {
2:   unsigned char* src = data.src; // data has only a reference
3:   unsigned char res = 0;
4:   for (int j=0; j<3; j++)
5:     for (int i=0; i<3; i++)
6:       if (src[1*3+1] > src[j*3 + i]) res += (1 << (j*3+i));
7:   write(res, data);
8: }
```

Figure 3: An example of *unit task* of `Feature Extraction` stage in `Face Detection`.

produced output to the buffer of its own stage. Then, it updates a flag or relevant counters to indicate the readiness of the output. When all the required data items for a task are ready, the *trigger-based task scheduler* of GOPipe enqueues the ready task into the task queue of the next stage. GOPipe uses *reference-type task representation* to represent a task, which helps avoid unnecessary data movements (details in Sec. 4.1). Its carefully designed scheduling algorithm helps minimize delays in the execution of the stages.

4 GOPIPE FRAMEWORK

This section presents the key techniques of GOPipe. We start with two concepts, *unit task specification* and *reference-type task*, and then describe three schemes we have designed for efficient *trigger-based scheduling* and their respective pros and cons. After explaining automatic *task grouping*, we describe the *autotuner*.

4.1 Unit Task and Reference-Type Task Representation

In GOPipe, programmers need to specify the unit task computation of each stage in an `execute()` function. Fig. 3 shows an example of unit task specification of a `feature extraction` stage, which computes binarization of the pixels in `data`, a 3×3 patch.

In conventional task-based programming frameworks, the produced output is directly enqueued into the task queue of the next stage, which can cause redundant data copies, especially for stencil pipeline applications. In such applications, a task usually consumes a tile of data produced by the previous stage, and the tiles consumed by adjacent tasks often overlap. As each of the tiles is treated as a standalone task and is enqueued into the task queue, the overlapped parts end up being copied multiple times.

The *reference-type task* representation used in GOPipe avoids the problem. As Fig. 4 shows, GOPipe equips each stage with a *frame buffer* that resides in global memory. As already mentioned, once a task has been processed in a stage, the output is not directly put into the task queue of the succeeding stage. Instead, the output is written into a *frame buffer*, and later, concise references to the *frame buffer* that indicate the range of data to be consumed by the task are enqueued into the task queue by the task scheduler.

The format of the reference is the starting location (e.g., the top left cell in a 2D stencil) of the data in the *frame buffer*. There is no need to specify the range of the data in the reference, as it is the same for all tasks in that stage. As data are not directly embedded into a task, this design avoids redundant data copies across tasks.

The similar concept to the *reference-type task* representation has been introduced in previous work [28, 41], in which it helps them avoid redundant copies between producers and consumers.

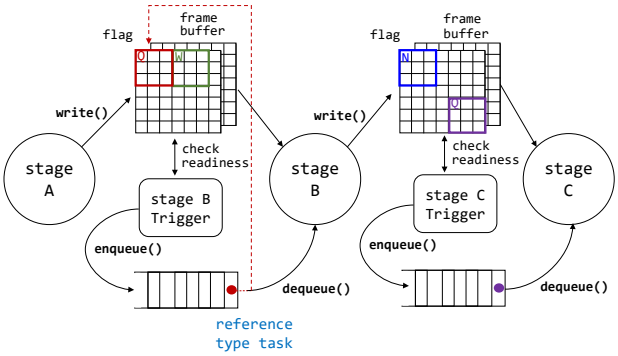

Figure 4: The trigger-based pipeline execution model in GOPipe.

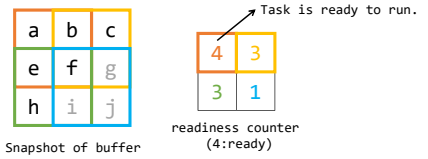

Figure 5: Counter-based integrated task scheduling.

In GOPipe, it is used to efficiently support flexible task granularity; no matter how the tasks are grouped (Sec. 4.3), the reference to data for the grouped task can be easily calculated by GOPipe, which allows users to only specify the computation as a unit task.

Note that a buffer is allocated to a stage when it is instantiated, and if there is frame-level parallelism, having more than one buffer for one stage can help prevent the potential write blocking.

After understanding these concepts, we are ready to see a key technique in GOPipe for scheduling tasks of an arbitrary granularity. It is worth noting that the separation of data from task representation also offers conveniences for task scheduling as we will see next.

4.2 Trigger-Based Task Scheduling

A major challenge for enabling granularity-oblivious programming frameworks is efficiently tracking the status of tasks of various granularities, the readiness of their dependent data items, and scheduling the tasks effectively when they are ready to run. We tackle the challenge by designing three scheduling mechanisms, with two of them decoupling computation and scheduling to minimize the interference from scheduling operations on computations. The three mechanisms feature different strengths and weaknesses, providing the set of options the *autotuner* (Sec. 4.4) can pick to provide efficient support for various pipeline program execution scenarios.

4.2.1 Counter-Based Integrated Task Scheduling. In some cases, finding a ready task for the next stage could be done efficiently at the time of writing the output data item, which can avoid the overhead of running a separate scheduler for each stage. Our first designed scheduling mechanism takes this integrated scheme. For a stage (say A), it maintains a *readiness counter* for each task of its next stage (say B). When A calls the API `write()` to output a result data item into the *frame buffer*, the API internally conducts a series

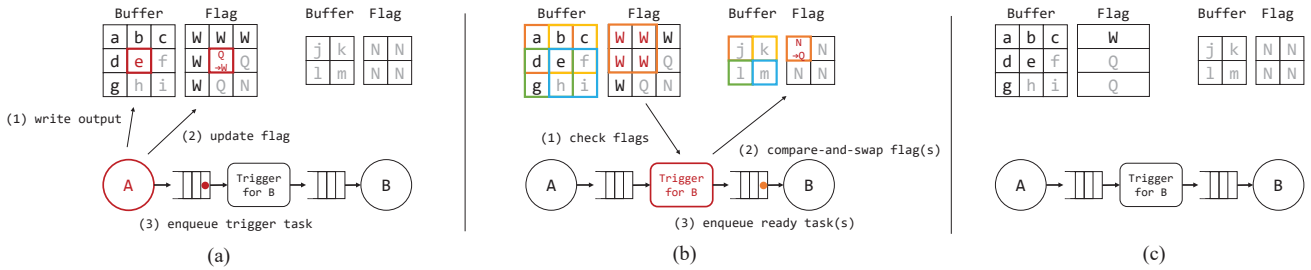

Figure 6: Illustration of event-driven decoupled task scheduling.

of operations to increment the *readiness counters* that correspond to the tasks of B that need to use that data item. A task is put into the task queue of B when its *readiness counter* reaches m, the number of data items needed by a task in B, as it indicates that the task is ready to run.

Fig. 5 gives an illustration of the counter-based scheduling. Fig. 5(a) shows the current state of the *frame buffer* where the size of the data range for a task of the next stage is 2×2. The values of the corresponding *readiness counters* are shown in Fig. 5(b). As the top left 2×2 tile is all filled with values, the first counter value is 4, indicating that the corresponding task is ready for the next stage to run.

This scheduling mechanism avoids the needs and associated overhead of a separate runtime scheduler. The lightweight scheduling is efficient when the task grain size is large and the total number of tasks is not. But if the task grain size is small and there are many tasks in the application, the scheduling operations inside `write()` could cause much delay to the computation of a stage as those operations are on the critical path of `write()` function. Checking the dependency from the side of the producing stage is complex: If the produced output data item belongs to multiple tasks in the succeeding stage(s), then it has to increment as many counters as the number of tasks.

4.2.2 Timer-Based Decoupled Task Scheduling. This design mitigates the problems of the former design by decoupling scheduling and computation. A separate module, *Trigger*, is implemented dedicated to tracking the status of data items and readiness of tasks. The module is implemented as a GPU kernel, periodically invoked such that it can run concurrently on GPU with the pipeline computations.

Our design is to associate a *flag buffer* with each stage, and use a three-state flag to track the status of each data item in the *frame buffer* of that stage. Let F be the flag in A's *flag buffer* that corresponds to a data item x in A's *frame buffer*. The value of F can be one of the following, indicating the different status of the task that produces x:

(1) `NotQueued`: The task has not been enqueued into A's task queue yet. This is the initial state of the flag.

(2) `Queued`: The task either has been enqueued into the task queue or is being processed.

(3) `Written`: The task has finished its job and x has been written into the *frame buffer*.

Each time when *Trigger* is launched, it checks all the flags to determine the readiness of each task for the next stage and enqueues those ready tasks only when the status of the tasks is `NotQueued`. In this design, the only scheduling-related work done by the API `write()` is to set the tri-state flag after it puts a data item into the *frame buffer*, making it much more efficient than in the first scheduling mechanism.

Although this decoupled design avoids the main drawbacks of the first design, it has its own weaknesses. First, the repeated invocations of the *Trigger* kernel incur overhead, both the launching overhead and the overhead caused by *Trigger*'s GPU resource usage. The overhead could be substantial when the tasks are produced quickly and *Trigger* needs to be called frequently. Finding the best invocation frequency is tricky as the pipeline may have dynamic behaviors and the producing and consuming rates of stages may change. Second, lots of work by *Trigger* could be useless. At each invocation, it checks all flags, but sometimes only a small portion of the flags have been updated since its previous launch.

4.2.3 Event-Driven Decoupled Task Scheduling. Our third design tries to mitigate the drawbacks of the second design by replacing timer-based triggering with event-based triggering. This *event-driven decoupled scheduling* is designed as follows. The scheduling module *Trigger* for a pipeline stage becomes a stage itself in that pipeline. The *Trigger* for B becomes a stage between A and B and it has its own task queue as the other stages in the pipeline do. When an output is written by A, it also enqueues an item into the task queue of *Trigger*. The item represents a request for *Trigger* to check the readiness of those tasks for stage B that are dependent on this output. If a task is indeed ready, *Trigger* inserts it into the task queue of stage B. This event-driven scheme helps avoid some of the main issues in the timer-based design presented earlier.

To see how the *Trigger* algorithm works, suppose that the unit input size of B is 2×2. The three-state flag introduced in Sec. 4.2.2 is also used in this scheme. When a task e of A writes the data item into the *output frame buffer*, *buffer(e)*, *flag(e)* turns to `Written` as shown in Fig. 6(a) and an item referring to *buffer(e)* is enqueued into the task queue of *Trigger* for B. Note that *flag(e)* had been changed from `NotQueued` to `Queued` by the *Trigger* for A, though it is not depicted in the figure. After *Trigger* dequeues the item, it checks the readiness of *all* tasks of stage B that are dependent on the data item in *buffer(e)*. For example, the tasks j, k, l, and m in Fig. 6(b) need to be checked as all of them require the data in *buffer(e)*. Suppose *Trigger* finds that *all* flags of the elements in the tile required by a task j of B is `Written`, it checks the flag of the task, *flag(j)*. If the flag is `NotQueued`, it sets it to `Queued`, and then puts the task j into the task queue of B. Setting the flag is done atomically, and if the value

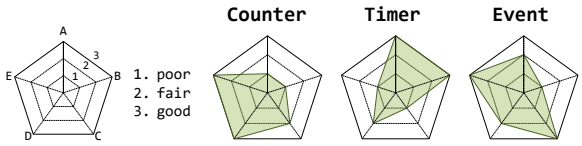

Figure 7: The strengths and weaknesses of each scheduler. Each metric (A,B,C,D,E) is explained in Sec. 4.2.4.

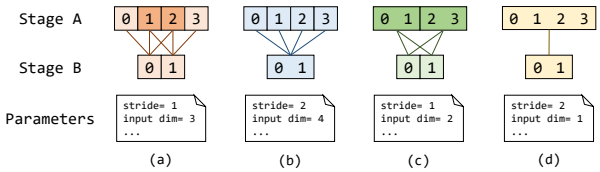

Figure 8: Examples of task grouping for a simple 1D stencil.

of the flag is `Queued`, it does not enqueue it again. If any element in the tile is not `Written`, *Trigger* simply does nothing; it might be tempting to think that in this case, *Trigger* should re-enqueue that task into its own task queue for future checking. That is actually not necessary because when that data item whose flag was not `Written` is written, all tasks that are dependent on it and have not yet been enqueued into stage *B* will be automatically checked again.

The event-driven feature of this mechanism helps it avoid the drawbacks of the second design, and its decoupling feature helps it avoid the shortcomings of the first design. In contrast to the integrated scheduling scheme, the decoupled scheduling scheme allows us to run *Triggers* on a dedicated SM(s), not on the same SMs that run the stages. Since *Triggers* are isolated in the dedicated SM(s), the large register usage of *Triggers* cannot affect the performance of the stages. As the previous two mechanisms, this scheme has its own weaknesses: The *Trigger* stage in the event-driven scheme could affect performance for the contentions in accessing the globally maintained task queues especially when task granularities are too small as every task invokes the *Trigger* stage at the end of its execution.

4.2.4 Comparison among Schedulers. Fig. 7 illustrates the characteristics of the proposed task schedulers. Each scheduler has its own strengths. The metrics are as follows.

A. Register usage This metric indicates the usage of register files for the scheduling routines. The counter-based integrated scheduler consumes much more register files as the scheduling code is integrated while the code for the other schedulers are detached. Note that large register usage does not necessarily degrade performance; it provides higher instruction-level parallelism (ILP) which may lead to better performance if thread-level parallelism (TLP) is already enough.

B. Invariance on the number of tasks The scheduling routine for the counter-based and event-driven schedulers is invoked as many times as the number of tasks. It could incur large overhead if the task granularity is small and there are many tasks. The timer-based scheduling is less influenced by the number of tasks as the *Trigger* is waked up periodically regardless of the number of tasks.

C. Invariance on the number of stages In case there are many stages, scheduling overhead of the timer-based scheduler increases since it employs as many *Triggers* as the number of stages. Without the optimal periods, there could be many meaningless invocations of *Trigger*.

D. SM-level parallelism The scheduling routines for the timer-based and event-driven schedulers are detached from the computation, and employ a dedicated SM to work, which may decrease the peak performance.

E. Simplicity of control This metric indicates how easy it is to achieve the optimal performance. Finding the best period for the timer-based scheduler is tricky.

The quantitative comparisons on the metrics among schedulers will be given in Sec. 5.1.

4.3 Task Grouping

The task grouping scheme in GOPipe centers around flexible yet efficient task re-definitions. It requires no user's interference at all. It does it by grouping adjacent unit tasks into a larger task based on reference-based task representations.

The dependence between tasks in stencil kernels is determined by the first location and the last location (e.g., the top-left and the bottom-right corners in a 2D stencil) of the dependent tasks within the input dimension in the producer stage(s). Thus, the inter-task dependence can be expressed by several parameters such as input dimension, stride, the task grain sizes of the producer and consumer stages, and so forth, as well as the location of the tasks in the frame buffer. GOPipe takes as input such parameters for unit tasks from the users. Fig. 8 illustrates an example with a simple 1D stencil. It consists of two pipeline stages *A* and *B* where stage *A* produces the input of stage *B*. The boxes represent the tasks and the lines represent the dependence: for example, a (unit) task of *B* depends on three (unit) tasks of *A* in Fig. 8(a). If task grouping is applied, GOPipe automatically derives new parameters for a coarse task based on the given unit task parameters. Fig. 8(b) shows the case where two unit tasks of *B* are grouped into a single task; the single task of *B* now depends on four tasks of *A*. The new stride, for instance, can be calculated by multiplying the unit stride, 1, and the task grain size of *B*, 2 (i.e., the number of grouped tasks), which is 2. On contrary, the input dimension of *B* becomes smaller in case the tasks of *A* are grouped. An extreme example is the coarsest configuration shown in Fig. 8(d) where the whole unit tasks in each of both stages form a single task. Only one dependence exists between two stages in this case (e.g., the input dimension is 1).

The grouped task is treated the same as a single unit task. The scheduling routine will be invoked only once for a grouped task at the end of its computation by the GOPipe API (e.g., `write()`), and will update the counters or the flags based on the aforementioned parameters. For this, each auxiliary data structure such as the readiness counter or the flag for dependency tracking represents the state of the corresponding coarse task rather than a unit task. Suppose that the three tasks along x-axis in Fig. 6(c) are grouped. The *flag{a,b,c}* is then set as `Written` when all output of the grouped task is put into the *buffer*. *Trigger* also checks the readiness of all the next tasks that require at least one output in the grouped task (*j* and *k* in Fig. 6(c)).

A CTA dequeues a single (larger) task from the task queue, and threads in the CTA process the unit tasks that the dequeued task is comprised of. If the task grain size is larger than the number of threads in the CTA, the threads in the CTA iterate through a loop to process the task. On the contrary, if the task grain size is smaller than the number of threads, the CTA dequeues multiple tasks from the task queue to ensure that no thread is left idle. Since the computation of the task is implemented by calling the computation code of a unit task, GOPipe does not require any re-compilation, making the task granularity adjustment easy and efficient.

GOPipe allows for grouping adjacent tasks in the same stage along any axis or their combinations. The task grouping keeps the tasks in a stage in equal size to avoid introducing load imbalance. GOPipe's task grouping has a very little overhead, including just one indirection to the buffer for a grouped task, which is negligible in most cases.

4.4 Autotuner

While GOPipe allows applications to have arbitrary task granularity transparently without modifying the code, finding the appropriate granularity by manual efforts is still tedious due to its large design space. GOPipe provides an *autotuner* that automatically finds the good task grain size for each stage, as well as the suitable scheduling mechanism to use.

The auto-tuning tries each kind of the scheduling schemes. For each scheduling scheme, it searches for the appropriate granularities in an iterative manner. In each iteration, it finds the best task granularity for each stage one by one. As the granularity in the later stage could change the best granularity of a former stage, this process repeats until either convergence or a maximum number of iterations (3 in our experiments) are reached. The search for the best granularity for a stage starts from the largest size and decreases gradually. On sampled benchmarks (Sec 5), we observe that the auto-tuning method finds near-optimal values; the performance is above 98% of that produced by the granularities found by exhaustive search.

In addition, the autotuner inherits the autotuning feature included in the base pipeline framework [40] for searching for appropriate execution models. Although in most cases, persistent thread based execution models work well, there are cases where the simpler kernel-by-kernel execution model with global synchronization works better, especially when the kernel invocation is not very frequent or most of stages have sufficient data-parallelism. Therefore, GOPipe autotuner evaluates all possible execution models and returns the best configurations. Users are not required to take any additional efforts.

5 EVALUATION

To test the efficacy of GOPipe, we evaluate it with six real-world pipeline applications, which cover different stencil dependencies. The machine is equipped with NVIDIA GTX 1080Ti GPU running CUDA 8.0 and Intel Xeon E5-2630 CPU. We study the performance in both batch and streaming settings; the former has a set of inputs ready to process by the pipeline, while the latter has a stream of inputs flowing in continuously. For performance measurement, we repeat each measurement 50 times; as the observed variance is negligible (<0.9%), we report the average as the reported performance. We in addition report the productivity benefits.

Table 1: Pipeline applications used for evaluation. (KBK: Kernel-by-kernel, MK: Megakernel, CP: Coarse pipeline)

Application	Abbr.	Execution model	Scheduler
HotSpot	HS	MK	Counter
Cell	CL	MK, CP	Counter
Image Pyramid	IP	MK, CP	Event
Face Detection	FD	MK, CP	Counter
Local Laplacian Filters	LF	KBK, MK, CP	Event
Box Filter	BF	KBK	N/A

5.1 Results in the Batch Setting

In the batch setting, a given collection of several inputs are fed into each of the pipeline benchmarks. We compare the performance to two baselines. One is *VersaPipe* [40], the state-of-the-art framework that supports true pipeline executions on GPU; the other is *Halide* [29], an efficient DSL(Domain Specific Languages)-based framework for image processing on heterogeneous systems, which uses GPU as a monolithic accelerator. Table 1 lists various pipeline applications used in the experiment including execution models and schedulers used for each application in GOPipe. The autotuning took from 36 minutes (`Boxfilter`) to 75 minutes (`Local Laplacian Filters`).

The implementations of `Image Pyramid` and `Face Detection` on VersaPipe are from the authors of VersaPipe; the task granularities were determined and hard coded by the previous authors. They used a whole image as the task granularity for most stages except the final stage of `Face Detection`, which uses tasks of the finest granularity as no inter-task dependence exists after the last stage. The other four programs on VersaPipe are implemented by us. They use the finest task granularity as it gives 2-3× better performance than using an image as the granularity.

It is important to note that although Halide includes an auto-scheduler for tuning loop schedules [1], the lead author of that work told us that the GPU support of the schedules is not ready to use. We hence worked with Halide authors to manually tune the schedules of the benchmarks. The schedules of `HotSpot` and `Local Laplacian Filters` used in our evaluation on Halide are provided by the authors of Halide, and the schedule of `Face Detection` is also verified by them[2]. We manually tuned GPU schedules for other applications and report the best performance we have achieved. The evaluated schedules, including function inlining, are *vectorize, unroll, reorder, split, fuse, compute_at, compute_root*, and some combinations of them.

5.1.1 Benefit over VersaPipe. In our experiments, we observe that the main performance benefit of GOPipe over VersaPipe are from better task granularity and efficient scheduling. VersaPipe provides enqueue and dequeue APIs to enable task-parallel execution and scheduling on GPUs, but contains no task scheduler or the capability to resolve inter-task dependencies; users need to write their own code. Consequently, the programmer's choices about task granularities are limited and often far from the optimal due to the lack of an

[2]Our thanks to the Halide authors for their great help.

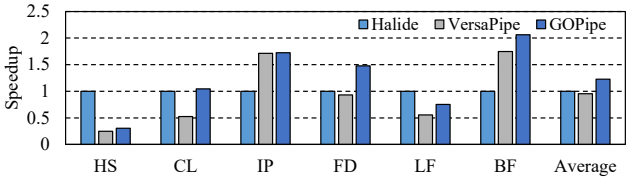

Figure 9: Performance comparison of GOPipe with VersaPipe and Halide. The exact times are reported in Sec. 5.1.3.

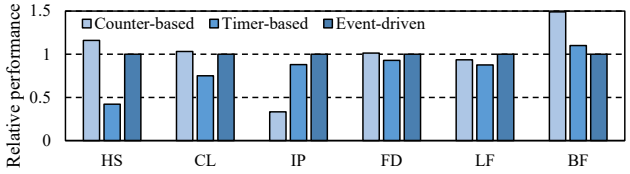

Figure 10: Relative performance among GOPipe schedulers with the batch setting used in Sec. 5.1 (Higher is better).

efficient dependency tracking. For example, `Image Pyramid` and `Face Detection` in VersaPipe only have the coarsest task granularity in most stages, which results in a limited data-parallelism, and also inferior performance. We also find that even if the same granularity is employed, VersaPipe cannot show its full potential unless the programmer carefully resolves the dependencies among different tasks (Sec. 5.3).

5.1.2 Comparison with Halide. The two major merits of Halide over GOPipe are the use of shared memory and function fusion, or inlining. The former is not supported in current GOPipe but the latter is similar to the RTC (Run to completion) execution model where the producer and consumer are merged into a single kernel. As explained in [40], RTC model has a tradeoff between good locality and large register usage. In VersaPipe and GOPipe, RTC is only supported when the producer and consumer has one-to-one dependency, while Halide can fuse the functions with stencil dependency at the expense of redundant computation.

GOPipe can employ persistent threads for each stage and schedules them dynamically so that it can execute the appropriate stage(s) regardless of the producing-consuming rate between stages. Even if a producing or consuming stage has dynamic workloads and the rate changes, GOPipe can adapt to this variation. Moreover, any stage that is ready can be executed whenever there are enough resources. In contrast, Halide supports only a fixed producing-consuming rate among fused stages and the stages that are not fused cannot be executed at the same time even if there are idle resources in the GPU.

In terms of productivity, users can describe algorithm and schedule independently in Halide, which makes it easy to explore various schedules. However, it is quite hard to find a good schedule due to the limitation of the auto-scheduler for GPU, which seriously limits the productivity. The programmer, sometimes, needs to write more than 50 lines of code for schedule only for a 2-line computation. Also, both algorithm and schedule codes need to be manually modified if the pipeline structure changes. In GOPipe, once users describe the computation of each stage with the finest task granularity, GOPipe will automatically adapt the algorithm with the best configuration computed by the autotuner based on the computation and the information about pipeline structures. Users do not need to optimize their code or modify the computation code manually even if they want to change the pipeline structure such as the order or the number of instances of a stage in the pipeline.

5.1.3 Detailed results. As shown in Fig. 9, GOPipe outperforms Halide on four out of the six benchmarks, and VersaPipe in every case. The average speedups are 1.23× and 1.39× over the two prior work respectively. The execution time here is end-to-end time, including the memory transfer time between host and device and the kernel launch overhead.

Fig. 10 shows the comparison among schedulers. We found that no scheduler is always the best and the choice of the scheduler can lead to a large performance difference depending on the applications. Detailed comparisons will be given in the discussions on each application. With auto-tuned granularities, the scheduling overhead of the counter-based integrated scheduler is less than 3% of the total cycles of the computation of the tasks. For the two decoupled schedulers, we assigned 3 SMs out of 28 SMs for *Trigger*. That loss of SM-level parallelism turns out to influence performance only slightly. The decoupling of the computation and the scheduling could actually lead to better performance with the increased SM occupancy thanks to the reduced register consumption. For instance, if both the computation and *Trigger* stages of the event-driven scheduler are assigned to the same SMs without decoupling, the occupancy decreases to 53% in `Face Detection`. As finding the best periods of the timer-based scheduler is tricky, we let *Triggers* have the shortest periods so that no SM for *Trigger* goes idle.

We now give detailed discussions on each of the applications.

HotSpot & Cell `HotSpot` [16] iteratively solves partial differential equations for each cell, which corresponds to a 5-point stencil. `Cell` (Cellular automaton) has a 3-dimensional grid. Similar to `HotSpot`, it continuously checks whether each cell is either live or dead depending on the previous status of neighbor cells (i.e., $3 \times 3 \times 3$ stencil). The implementations are based on Rodinia [10]. We evaluate a 512×512 and a $512 \times 512 \times 16$ grids for `HotSpot` and `Cell`, respectively. In GOPipe, a recursive iteration in a stage is implemented as an instance of the stage. GOPipe takes 18ms for 500 iterations of `HotSpot` and 72ms for 100 iterations of `Cell`, outperforming VersaPipe by 1.22× and 1.99×.

On both applications, the counter-based scheduler is chosen as the best scheduler since the dependency patterns are simple; they require only a few atomic operations.

Fig. 11(c) shows the execution times of `Cell` on various task grain sizes. In Fig. 11, all stages have the same relative task grain size to each *stage output dimension*. For instance, the grain size of 1/64 on `Cell` may represent that $512 \times 8 \times 16$ cells in each iteration comprise one task as the *stage output dimension* is $512 \times 512 \times 16$. Although the best scheduler is the counter-based one, the efficiency of the timer-based scheduler improves as the task grain size decreases. This is because scheduling overhead of the other two schedulers is proportional to the number of tasks while the number of *Trigger* invocations in the timer-based scheduler is invariant on the task grain size. Applications with many tasks would benefit from the timer-based scheduler. On the other hand, the timer-based scheduler

becomes inferior when the number of iterations is large as shown in Fig. 11(top). Since each iteration is implemented as a pipeline stage in GOPipe, there is large overhead in maintaining the hundreds of stages of *Trigger* (See Sec. 4.2.4).

In Halide, the code for four iterations is inlined into one kernel, in which the intermediate values of the inlined iterations are stored in shared memory. Based on the scheduler provided by the authors of Halide, we did further optimization as we found that execution time of Halide increases superlinearly with the number of iterations: it took 10ms for 500 iterations and 237ms for 3,000 iterations on `HotSpot`. We divided the total number of iterations into several subsets of iterations and placed a *Buffer* between the subsets to prevent the propagation of some dependencies across iterations. As a result, the running times of Halide for `HotSpot` become 6ms for 500 iterations and 48ms for 3,000 iterations. Without our manual optimization, GOPipe would be faster than Halide for 3,000 iterations, as Halide would take 107ms to finish.

Image Pyramid `Image Pyramid` [2] is a widely used technique in many image processing applications. The purpose of `Image Pyramid` is to provide a set of images of various sizes. Our implementation is based on a prior study [24]. The pipeline consists of three stages: `Grayscale`, `Histogram Equalize`, and `Resize`. The `Resize` stage requires a 2×2 tile to produce one output pixel.

For a given UHD ($3840 \times 2160 \times 3$) input image, 6 resized images are produced when the scale factor is 2. The `Histogram Equalize` stage uses only one CTA due to insufficient parallelism. With the conventional data-parallel model, the rest of the GPU would remain idle unless a large batch is used, which would increase the latency. Also, there are considerable load imbalance as each `Resize` stage processes an image of different size; one produces a 1920×1080 image while another produces a 60×33 image. Due to the load imbalance, the performance is degraded with the large granularity in VersaPipe. On the contrary, GOPipe can distribute the different workload efficiently with finer grain task scheduling.

As a result, the experiment with ten UHD images shows that GOPipe with the event-driven scheduler shortens the processing time to 83ms, a 1.72× speedup over Halide as shown in Fig. 9. The counter-based and timer-based schedulers take 167ms and 91ms, respectively.

Face Detection `Face Detection` finds out the location of faces in a given image. The base implementation came from an earlier work [24], which employs an LBP (Local Binary Pattern) approach. All stages, except the second stage, have stencil patterns. The unit input dimension in face detection ranges from 3×1 to 24×24.

`Face Detection` consists of five stages as Fig. 1 shows. Since, in order to detect faces of arbitrary sizes, it resizes an image recursively until the relative size of a search window reaches the pre-configured size, the number of actual stages is much larger (17 for a $1920 \times 1080 \times 3$ resolution image). The `Scanning` stage adopts a cascaded classification [38] which consists of several weak classifiers. The number of the weak classifiers to be processed depends on the input. Halide is not suited to such a dynamic behavior since it mainly aims at optimizing static affine computation. In Halide, all the weak classifiers need to be processed regardless of the input.

GOPipe with counter-based scheduler takes 28.4ms in processing ten $1920 \times 1080 \times 3$ images. The other two schedulers have execution times of 29.9ms and 28.5ms, respectively, as shown in Fig. 10. The

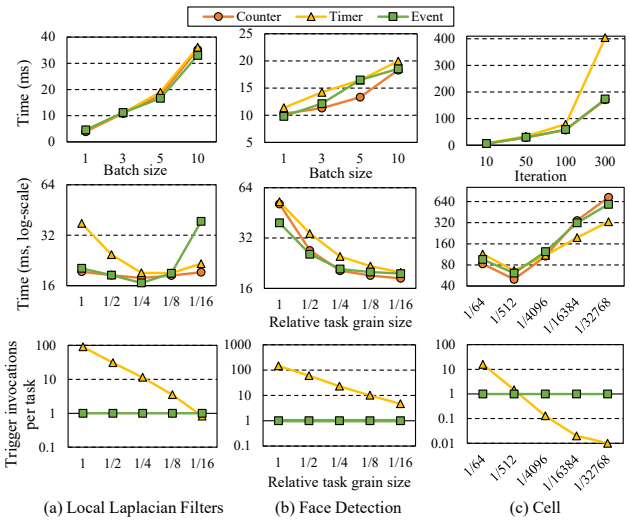

Figure 11: Execution time for various batch sizes or iterations (top), and for various task granularity (middle). The number of *Trigger* invocations per task for the decoupled schedulers (bottom).

counter, timer, and event-based schedulers consume 88, 56, and 80 registers per thread respectively. Each stage of `Face Detection` has various task grain sizes: from 280 to 2M. Fig. 9 shows that GOPipe outperforms VersaPipe and Halide implementations by 1.60×, 1.49×, respectively.

In auto-tuning, the data sensitivity of `Face Detection` may take effect. To figure out the impact, we additionally experimented with a different input that has 10× more faces than the dataset for tuning, and the performance difference turned out to be less than 10%.

Local Laplacian Filters `Local Laplacian Filters` [25] builds a Laplacian pyramid to accomplish an edge-aware image processing. For 960×512 size of image, it deploys 37 pipeline stages with a complex topology.

The workload of a task in each stage of `Local Laplacian Filters` is usually tiny; a task may operate one addition and three global memory accesses. For such cases, Halide can show its strength for its function fusion optimizations, while the scheduling overhead in GOPipe is relatively large; Halide takes 8ms for this application.

The determined execution model in GOPipe is a hybrid model as shown in Table 1. As some stages have sufficient data-parallelism while the computation is simple, those can benefit from KBK execution without dynamic scheduling. Out of 37 stages, 8 stages run with the KBK model and the others adopt the Megakernel model. The numbers of SMs assigned to these models are 15 and 10, respectively. The remaining 3 SMs are assigned to the event-driven scheduler, which runs with a coarse pipeline model. GOPipe takes 11ms which is slower than Halide but is 1.36× faster than VersaPipe.

While GOPipe cannot automatically fuse stages, we have observed that some simple manual fusions in Local Laplacian Filters on GOPipe can improve the performance shown in Fig. 9 by 1.26×, narrowing the gap with Halide (8.4 ms vs. 8.0ms).

The event-driven scheduler is chosen as the best by the auto-tuner (Fig. 10). As shown in Fig. 11(a), however, the counter-based

scheduler could have been the best if task grain size was smaller. It benefits from higher ILP since TLP is limited due to the small batch size.

Box Filter `Box Filter` [22] is an image convolution application that smooths an image by converting each pixel value to the average value of adjacent pixels. Its stencil size is equal to the filter size. In this paper, the implementation of box filter consists of four stages that convolve the image with two different filter sizes using separable filters: 4×4 and 16×16.

For ten 3840×2160 images, GOPipe outperforms VersaPipe and Halide by 1.18× and 2.06×, respectively. The running time of GOPipe is 52ms. The autotuner chooses KBK as the execution model and 63K elements as the granularity.

5.2 Streaming Scenario

This part evaluates `Face Detection` when inputs arrive in a steady stream, which is a common usage scenario for such applications.

Setup The objective is to maximize throughput TH while achieving a response time R shorter than a timing constraint D. One input image arrives in every Gms. The response time is defined as the longest time span between the arrival of an image and the attainment of its processed result. In order to achieve a high throughput, an input is first put into a buffer, and the whole buffer gets processed at once on the GPU if the number of images in the buffer reaches a threshold N. Such a scheme is common in streaming processing for GPU as the performance of both data transfer and GPU processing favor a set rather than one individual image. The response time R for a given batch size N is then defined as follows:

$$R(N) = T_{wait}(N) + T_{process}(N) + T_{transfer}(N) \quad (1)$$

where T_{wait} and $T_{process}$ represent required time for forming a batch (i.e., $T_{wait}(N) = (N-1) \times G$) and processing time on GPU, and $T_{transfer}(N)$ is the time to transfer N images.

Results To provide a comprehensive comparison, we manually modified the program to derive three other versions for the benchmark on VersaPipe. Including the default VersaPipe version (VP-C), we now have four version; they differ in task granularities, the finest granularity (VP-F), quarter of an image (VP-Q), half of an image (VP-H).

Fig. 12 reports the experimental results for a spectrum of streaming settings. GOPipe achieves the best performance overall, showing the benefits of its adaptive granularity and scheduler selection for meeting the needs of various settings. In contrast, no version of the VersaPipe implementation fits all. The task granularity of GOPipe for each stage varies from 1×2 to 1920×1080 in the scenarios shown in Fig. 12.

Fig. 12(a) shows the result for $1920 \times 1080 \times 3$ (FHD) images when the response time constraint D changes. VP-C could not achieve a good throughput for the limited data-parallelism. Moreover, it misses the time constraint as required response time becomes shorter. GOPipe, for the adaptive granularity support, shows the best performance for all cases.

Fig. 12(b) and 12(c) show the results when the time constraint D is fixed to 80ms while the interval G changes for FHD and UHD input image sizes. The shorter time interval with the fixed deadline gives much room for forming a batch, yielding a higher throughput. Even though the implementation in the largest grain could have the largest throughput if the batch size becomes large enough, it cannot work in this kind of streaming scenario where response deadline exists. Fig. 12(d) and 12(e) show results when image resolution varies. The time constraints are 40ms and 30ms, and the input arrival rates are 30 FPS and 120 FPS, respectively.

5.3 Programming Productivity

Fig. 13(a) shows a pseudo-code for the `Resize` stage of `Face Detection` application in VersaPipe with a case where the task granularity corresponds to 1/9 of image. The implementation assumes that the succeeding stages (`Feature Extraction` and another instance of `Resize`) are defined to have the same granularity.

The tricky part is in checking readiness of the inputs for the task that is to be scheduled (line 9-19). It should be described properly for each given grain size and dependency between stages. The shown implementation assumes that each task for two succeeding stages can be enqueued when all adjacent patches (1/9 image each) are prepared.

The code may not need modification if the size of patch is large enough to meet the assumption. However, the assumption breaks when the granularity becomes smaller and a set of adjacent patches no longer contains enough data for the next stage. Code changes would also need to be done if the following stages do not have the same granularity. Further, The code entwined with the specification of the topology of the pipeline (line 15-16). Much of the scheduling code would need to be modified if there is a change in the topology.

In contrast, the code in GOPipe shown in Fig. 13(b) is much shorter and simpler as no tracking or scheduling needs to be coded by the programmer. As a result, the face detection application is expressed in 80 lines of code in GOPipe while VersaPipe needs 333 lines for the same task granularity, which is obtained from GOPipe autotuner. We find that the manual scheduling on VersaPipe resulted in not only less productivity but also less efficiency: GOPipe shows 1.16× still faster performance. The large code footprint and many conditional branches of manual scheduling in VersaPipe for complex granularities takes many registers and degrades the occupancy.

6 RELATED WORK

Recently, some domain specific frameworks are developed for pipeline computations on GPU. For instance, Patney et al. [27] design a framework for graphics pipelines, and optimize both load balance and data locality on GPU. Some systems [33, 36, 40] use persistent thread on GPU [3, 14, 32] to improve the performance of scheduling pipeline computations. Juggler [7] resolves dependencies among tasks dynamically in a persistent thread execution but only considers one-to-one matched dependence. On the other hand, there have been many studies on task-based programming models on CPUs, which employs dependence-aware dynamic task schedulers similar to our work [5, 6, 9, 37]. However, the distinctive characteristics of GPU make it a different problem. For instances, GPUs support neither interrupts among cores nor the preemption of a kernel, which can lead to deadlocks as a CTA that would release the lock cannot be scheduled. Our proposed scheduling algorithms are designed to address these complexities.

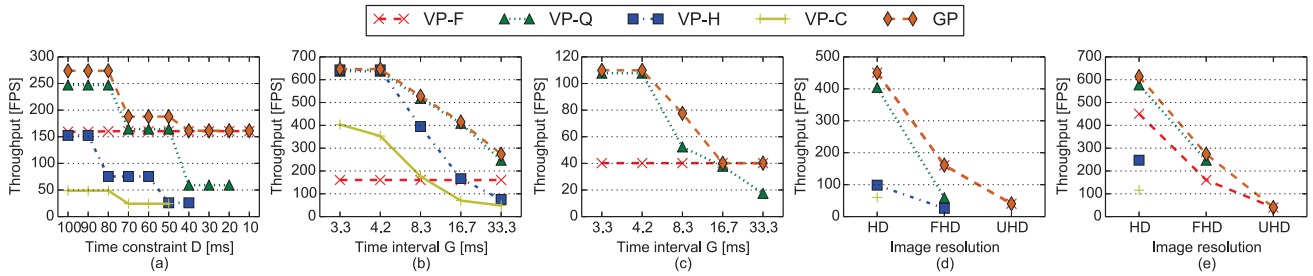

Figure 12: Throughput comparison over various settings. VersaPipe (VP) with four different task granularities are compared with GOPipe (GP). The settings that miss the constraints are not denoted in the graph.

```
1:  int flag[N_S][9] = {0, …};
2:  __device__ void Resize::execute(TaskItem data) {
3:    for (int i=threadIdx.x;i<data.width/3*data.height/3;i+=blockDim.x) {
4:      int y = (i*data.idx / 3)  *  data.width;
5:      int x = (i*data.idx%3) % data.width;
6:      data.dst[y][x] = interpolate(data.src, x, y, x+1, y+1);
7:    }
8:    __syncthreads(); // Tedious task scheduling routine begins below
9:    for (int i=-1; i<=1; i++) {
10:     for (int j=-1; j<=1; j++) {
11:       int2 n = saturate_cast<0,2>(id%3+i, id/3+j);
12:       int next_idx = n.y*3 + n.x;
13:       int readiness = atomicAdd(&flag[data.stage_id][next_idx], 1);
14:       if (readiness == 9-1) {
15:         if (data.stage_id != 18) enqueue<Resize>(…);
16:         enqueue<FeatureExtraction>(…);
17:       }
18:     }
19:   }
20: }
```

(a)

```
1:  struct TaskItem { int input_dimension[], int stride[], … };
2:  void Resize::execute(TaskItem data) {
3:    unsigned char res = interpolate(data.src, 0, 0, 1, 1);
4:    write(res, data);
5:  }
```

(b)

Figure 13: User-code comparison between VersaPipe (a) and GOPipe (b).

None of previous work address the task granularity problem for pipelined stencil programs. Some studies [4, 21] introduce the concept of task granularity but in a KBK model, where there is no need for resolving dependency between stages as they are implicitly resolved at the kernel boundaries. Task granularity should be easily parameterizable in such a case.

There are some recent systems optimizing image processing pipeline with stencil dependency. In addition to Halide [29] and PolyMage [23], Ravishankar et al. [30] propose a DSL for image pipeline target both CPU and GPU, whose compiler deals with GPU workload offloading and memory management. However, these systems do not support dynamic scheduling in the pipelined computation on GPU. Chugh et al. [12] propose an FPGA backend for PolyMage.

Kim et al. [18] and Liao et al. [20] propose hardware-based approaches to optimizing pipelined execution of dependent kernels, in which the dependency check is implemented in hardware. Tzeng et al. [35] gives a careful study on scheduling of image processing programs on GPU. It proposes a counter-based design to track dependency between tasks, which shares some commonality with our counter-based integrated scheduling. As we show, the scheduling is subject to some major shortcomings for pipelined stencil programs. They also propose a static scheduler that assumes the execution order among thread blocks, in which case, threads can be idle waiting for their execution order and should be careful in planning the static schedule to avoid deadlocks. Two scheduling algorithms in GOPipe decouple computation and scheduling in an efficient way, hiding the tedious and error-prone burden arising from the optimization with task granularity re-definition and the need for synchronization among fine grain tasks. Their work studies scheduling designs but does not provide programming frameworks, and gives no treatment to granularity selection or adaptation. A recent work [41] studies efficient communication among pipeline stages that run across the boundary between CPU and GPU. It provides an efficient communication library, HiWayLib, but does not deal with the selection of task granularities in pipeline applications.

7 CONCLUSION

We propose the first *granularity-oblivious* GPU pipeline programming framework named GOPipe. GOPipe features an efficient dynamic scheduler for efficiently tracking and resolving task dependencies, enabling automatically adaptation to different task granularities. Experimental results show that GOPipe delivers averagely 1.39× speedup over existing pipeline programming frameworks. It meanwhile eases pipeline program development on GPU by automatically adapting task granularity. (GOPipe will be released to public upon publication.)

ACKNOWLEDGMENTS

This work was supported by Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education (No. 2018R1D1A1B07050463). In China, this work is partially supported by the National Key R&D Program of China (2017YFB1003103), Beijing Natural Science Foundation (4202031), Beijing Academy of Artificial Intelligence (BAAI). Youngmin Yi, Xipeng Shen, and Jidong Zhai are corresponding authors of the paper.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 121.

[2] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. 1984. Pyramid methods in image processing. *RCA engineer* 29, 6 (1984), 33–41.

[3] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Conference on High PERFORMANCE Graphics*. 145–149.

[4] Prithayan Barua, Jun Shirako, and Vivek Sarkar. 2018. Cost-driven thread coarsening for GPU kernels. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 32.

[5] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2009. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *ACM sigplan notices* 44, 4 (2009), 219–228.

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

[7] 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 the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 54–67.

[8] Christian Bienia and Kai Li. 2010. Characteristics of workloads using the pipeline programming model. In *International Symposium on Computer Architecture*. Springer, 161–171.

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

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 44–54.

[11] Nagai-Man Cheung, Xiaopeng Fan, Oscar C Au, and Man-Cheung Kung. 2010. Video coding on multicore graphics processors. *IEEE Signal Processing Magazine* 27, 2 (2010), 79–89.

[12] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 327–338.

[13] Robert L Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. In *ACM SIGGRAPH Computer Graphics*, Vol. 21. ACM, 95–102.

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

[15] M Harris and K Perelygin. 2017. Cooperative groups: Flexible CUDA thread programming.

[16] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. 2006. HotSpot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14, 5 (2006), 501–513.

[17] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. 2001. Imagine: Media processing with streams. *IEEE micro* 21, 2 (2001), 35–46.

[18] Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. 2016. Automatically Exploiting Implicit Pipeline Parallelism from Multiple Dependent Kernels for GPUs. In *International Conference on Parallel Architectures and Compilation*.

[19] Kai Li and Jeffrey F Naughton. 2000. Multiprocessor main memory transaction processing. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*. IEEE Computer Society Press, 177–187.

[20] Wei-Cheng Liao, Yuan-Ming Chang, Shao-Chung Wang, Chun-Chieh Yang, Jenq-Kuen Lee, and Yuan-Shin Hwang. 2018. Scheduling Methods to Optimize Dependent Programs for GPU Architecture. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, 13.

[21] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 455–466.

[22] MJ McDonnell. 1981. Box-filtering techniques. *Computer Graphics and Image Processing* 17, 1 (1981), 65–70.

[23] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 429–443.

[24] Chanyoung Oh, Saehanseul Yi, and Youngmin Yi. 2015. Real-time face detection in Full HD images exploiting both embedded CPU and GPU. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 1–6.

[25] Sylvain Paris, Samuel W Hasinoff, and Jan Kautz. 2011. Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid. *ACM Trans. Graph.* 30, 4 (2011), 68–1.

[26] Anjul Patney and John D Owens. 2008. Real-time Reyes: Programmable pipelines and research challenges. *ACM SIGGRAPH Asia 2008 Course Notes* (2008).

[27] Anjul Patney, Stanley Tzeng, Kerry A. Seitz, and John D. Owens. 2015. Piko: a framework for authoring programmable graphics pipelines. *Acm Transactions on Graphics* 34, 4 (2015), 1–13.

[28] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–25.

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[30] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. 2015. Forma: A DSL for image processing applications to target GPUs and multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. ACM, 109–120.

[31] Changhe Song, Yunsong Li, and Bormin Huang. 2011. A GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images. *IEEE Journal of selected topics in applied earth observations and remote sensing* 4, 3 (2011), 683–690.

[32] Tyler Sorensen, Alastair F Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable inter-workgroup barrier synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 39–58.

[33] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *Acm Transactions on Graphics* 33, 6 (2014), 1–11.

[34] Weibin Sun and Robert Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*. IEEE Press, 25–36.

[35] Stanley Tzeng, Brandon Lloyd, and John D Owens. 2012. A GPU task-parallel model with dependency resolution. *Computer* 8 (2012), 34–41.

[36] Stanley Tzeng, Anjul Patney, and John D Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 29–37.

[37] Hans Vandierendonck, George Tzenakis, and Dimitrios S Nikolopoulos. 2011. A unified scheduler for recursive and task dataflow parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 1–11.

[38] Paul Viola and Michael Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, Vol. 1. IEEE, I–511.

[39] Feng Zhang, Jidong Zhai, Bingsheng He, and Shuhao Zhang. 2016. Understanding Co-running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel & Distributed Systems* (2016), 1–1.

[40] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, 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*. ACM, 587–599.

[41] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2019. HiWayLib: A Software Framework for Enabling High Performance Communications for Heterogeneous Pipeline Computations. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 153–166.