VersaPipe: A Versatile Programming Framework for Pipelined Computing on GPU

Zhen Zheng
Tsinghua University
z-zheng14@mails.tsinghua.edu.cn

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

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Xipeng Shen
North Carolina State University
xshen5@ncsu.edu

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

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

ABSTRACT

Pipeline is an important programming pattern, while GPU, designed mostly for data-level parallel executions, lacks an efficient mechanism to support pipeline programming and executions. This paper provides a systematic examination of various existing pipeline execution models on GPU, and analyzes their strengths and weaknesses. To address their shortcomings, this paper then proposes three new execution models equipped with much improved controllability, including a hybrid model that is capable of getting the strengths of all. These insights ultimately lead to the development of a software programming framework named *VersaPipe*. With VersaPipe, users only need to write the operations for each pipeline stage. VersaPipe will then automatically assemble the stages into a hybrid execution model and configure it to achieve the best performance. Experiments on a set of pipeline benchmarks and a real-world face detection application show that VersaPipe produces up to 6.90× (2.88× on average) speedups over the original manual implementations.

CCS CONCEPTS

• **General and reference → Performance**; • **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization → Heterogeneous (hybrid) systems**;

KEYWORDS

GPU, Pipelined Computing

ACM Reference Format:
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 MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123978

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123978

1 INTRODUCTION

Pipeline parallel programming is commonly used to exert the full parallelism of some important applications, ranging from face detection to network packet processing, graph rendering, video encoding and decoding [11, 31, 34, 42, 46]. A pipeline program consists of a chain of processing stages, which have a producer-consumer relationship. The output of a stage is the input of the next stage. Figure 1 shows Reyes rendering pipeline [12], a popular image rendering algorithm in computer graphics. It uses one recursive stage and three other stages to render an image. In general, any programs that have multiple stages with the input data going through these stages in succession could be written in a pipeline fashion. Efficient supports of pipeline programming and execution are hence important for a broad range of workloads [5, 22, 26, 51].

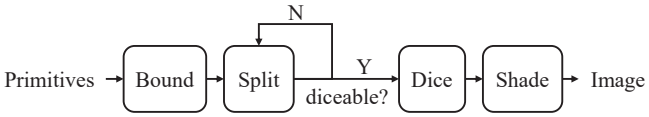


Figure 1: Reyes rendering: an example pipeline workload.

On the other hand, modern graphics processing unit (GPU) shows a great power in general-purpose computing. Thousands of cores make data-parallel programs gain a significant performance boost with GPU. While GPU presents powerful performance potential for a wide range of applications, there are some limitations for pipeline programs to make full use of GPU's power. With its basic programming model, each major stage of a pipeline is put into a separate GPU kernel, invoked one after one. It creates an ill-fit design for the pipeline nature of the problem. In the Reyes rendering pipeline, for instance, the recursive process would need a large number of kernel calls, which could introduce substantial overhead. And as stages are separated by kernel boundaries, no parallelism would be exploited across stages, which further hurts the performance. In the Reyes example, as the recursive depth varies for different data items in the pipeline, a few items that have deep recursive depth will cause the whole pipeline to stall.

How to make GPU best accelerate pipelined computations remains an open question. Some work has devoted manual efforts in developing some specific pipeline applications, such as graphic processing [34, 35, 37, 45] and packet processing [46]. They provide some useful insights, but do not offer a programming support to general pipeline problems. Some programming models have been proposed recently trying to simplify the development of efficient

pipeline applications on GPU. Most of them employ a Megakernel-based approach [3, 16], in which, the code for all stages of a pipeline is put into one large GPU kernel that branches to different sub-functions based on which stage the current data item needs to go through. At launching time, the method tries to launch as many blocks as can be concurrently scheduled on GPU and schedule the tasks through software managed task queues [10, 43, 44, 47].

Although these efforts have made good contributions to advance the state of the art, they have some serious limitations. Putting all stages into one kernel, Megakernel-based methods cause large register and shared memory pressure on GPU. As a result, many programs can have only a small number of threads running concurrently on GPU, leaving the massive parallelism of GPU largely untapped (detailed in Section 8).

An important observation made in this work is that, fundamentally, the prior efforts have been limited by the lack of software controllability of executions on GPU. GPU uses hardware schedulers to decide when and where (on which computing unit of a GPU) a GPU thread runs. All prior pipeline model designs have taken this limitation for granted, which has largely limited the design flexibilities.

In this paper, by combining two recent software techniques (persistent threads [3] and SM-centric method [50], explained later), we show that the limitation can be largely circumvented without hardware extensions. The software controllability released by that method opens up a much greater flexibility for the designs of pipeline execution models.

Leveraging the flexibility, we propose three new execution models with much improved configurability. They significantly expand the selection space of pipeline models. By examining the various models, we reveal their respective strengths and weaknesses. We then develop a software framework named *VersaPipe* to translate these insights and models into an enhanced pipeline programming system. VersaPipe consists of an API, a library that incorporates the various execution models and the scheduling mechanisms, and an auto-tuner that automatically finds the best configurations of a pipeline. With VersaPipe, users only need to write the operations for each pipeline stage. VersaPipe will then automatically assemble the stages into a hybrid execution model and configure it to achieve the best performance.

For its much improved configurability and much enriched execution model space, VersaPipe significantly advances the state of the art in supporting pipeline executions on GPU. Experiments on a set of pipeline benchmarks and a real-world face detection application show that VersaPipe produces up to 6.90× (2.88× on average) speedups over the original manual implementations.

Overall, this work makes the following major contributions.

- It provides a systematic examination of the space of possible execution models of pipeline computations on GPU. It analyzes the strengths and weaknesses of the various models.
- It introduces much improved software controllability into the design of pipeline execution models.
- It contributes three new configurable execution models for pipeline computations, which significantly expand the selection space of pipeline execution models.
- It proposes VersaPipe, a versatile programming framework that provides automatic support to create the combination of execution models that best suit a given pipeline problem.
- It evaluates VersaPipe on a set of pipeline workloads and compares its performance with several alternative methods. Results show that VersaPipe outperforms the state-of-art solutions significantly (by up to 1.66×), accelerates the original manual implementations by 2.88× on average (up to 6.9×).

Our approach targets NVIDIA GPUs, but the general principles behind the approach could be applied to any model based on an equivalent of NVIDIA's streaming multiprocessors, so long as the model provides a way to gain access to the id of the SM on which a thread is executing.

2 BACKGROUND

This section provides background on GPU that is relevant to this work. We use NVIDIA CUDA [30] terminology for the discussion.

Graphic processing unit (GPU) is widely used in a variety of general purpose computing domains now. It adopts a very wide Single Instruction Multiple Data (SIMD) architecture, where hundreds of units can process instructions simultaneously. Each unit in the SIMD of GPU is called Streaming Processor (SP). Each SP has its own PC and registers, and can access and process a different instruction in the kernel code. A GPU device has multiple clusters of SPs, called **Streaming Multiprocessors (SMs)**. Each SM has independent hardware resources of shared memory, L1 caches, and register files, while all SMs share L2 cache. For more efficient processing and data mapping, threads on GPU are grouped as **CTAs** (**Cooperative Thread Arrays**, also called **Thread Block**). Threads in the same CTA can communicate with each other via global memory, shared memory and barrier synchronization. The number of threads in a CTA is set by programs, but also limited by the architecture. All CTAs run independently and different CTAs can run concurrently.

GPU is a complex architecture in which resources are dynamically partitioned depending on the usage. Since an SM has a limited size of registers and shared memory, any of them can be a limiting factor that could result in under-utilization of GPU. When hardware resource usage of a thread increases, the total number of threads that can run concurrently on GPU decreases. Due to this nature of GPU architecture, capturing the workload concurrency into a kernel should be done carefully considering the resource usage of the kernel implementation, which is key to maximizing the throughput of the GPU.

On the other hand, some workloads may not have enough data-parallelism to fully utilize the GPU whereas it may have some task-parallelism as well. The conventional GPU execution model runs one kernel on all SMs, exploiting only data-parallelism. Although Concurrent Kernel Execution (CKE) can run more than one kernels at the same time, it is limited to the case in which the previous kernels do not fully utilize all the resources (SMs) in the GPU and the current kernel is not dependent on them. This allows for a limited form of task-parallel execution on GPU, but a task must be described in a separate kernel, which incurs overhead, makes GPU underutilized for kernels with small workload and usually cannot maintain good data locality as the binding of a task onto an SM cannot be controlled.

3 OVERVIEW OF VERSAPIPE

To help programmers deal with the complexities in developing efficient pipeline programs on GPU, we develop VersaPipe, a software framework that screens programmers from all the complexities, and automatically produces efficient pipeline models that best fit a given pipeline problem. Using VersaPipe, programmers only need to express the operations of each stage of the pipeline. VersaPipe will automatically identify the best way to assemble these stages into the most efficient pipeline GPU program. This section provides an overview of VersaPipe.

The framework of VersaPipe is shown in Figure 2, which consists of three components: an underlying efficient C++ template-based scheduling library, an auto-tuner, and a programming API intended to make VersaPipe easy to use.

The scheduling library (Section 5) is the main component of the framework, which translates the pipeline structure and functions from programming API into a combination of various execution models. It consists of several components shown in Figure 2. SM mapping controller maps each stage group onto target SMs. Block mapping controller maps thread blocks on SMs for the stage groups that use fine-grained pipeline (detailed in Section 4.2.2). Task scheduler defines the strategies of fetching tasks from each stage. Work queue library provides basic interfaces for queue operations.

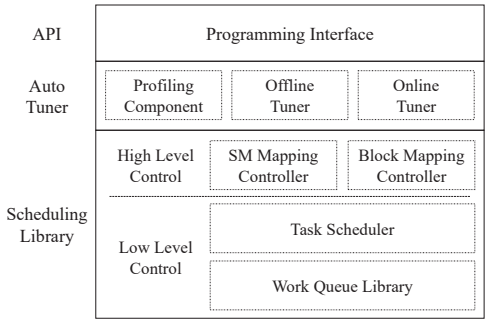


Figure 2: The framework of VersaPipe.

VersaPipe provides an auto-tuner (Section 7) which can automatically configure a combination of several possible execution models (explained in Section 4) to best suit a given pipeline. The auto-tuner consists of profiling component, offline tuner, and online tuner. Profiling component collects the workload characteristics of each stage. One main metric is the maximum number of blocks that can be launched concurrently on an SM for each stage. Based on the collected profiling information, offline tuner finds the most suitable execution model for each stage group, along with the SM binding strategy of each stage group and block mapping strategies for fine pipeline groups. Online tuner adjusts the configuration of VersaPipe dynamically while the pipeline program is running, to better utilize GPU resources.

As a programming framework, VersaPipe offers a simple API (explained in Section 6) to allow developers to program an arbitrary pipeline workload with ease and get high performance. Developers only need to write the function of each stage and specify the pipeline structures. All the other complex scheduling is done by the framework. The configuration for the scheduling can be set by the auto-tuner. VersaPipe is open-source and will be made available to the public.

We next explain each major component of VersaPipe.

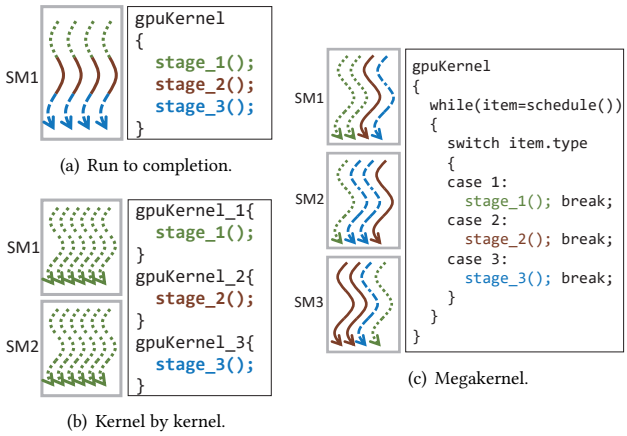
4 EXECUTION MODELS

We start with the comprehensive set of execution models it covers.

4.1 Existing Pipeline Execution Models on GPU

For pipeline programs, there are two major types of strategies to implement them on GPU. One is to use default programming models provided by underlying GPU platforms without adopting any software scheduling strategies. For this strategy, programming is relatively simple but task parallelism among different stages of a pipeline program cannot be exploited. Typical models include run-to-completion model and kernel-by-kernel model. The other is to schedule a pipeline program with a software scheduler on top of current programming models. With this strategy, more parallelism can be discovered from different stages. The most common model for this strategy is Megakernel [3].

Run to completion (RTC). RTC is a basic model to realize pipeline programs, which organizes all stages of a pipeline into a single kernel. All the pipeline logic controls are realized in this kernel. Figure 3(a) shows this model. For some regular pipeline programs (i.e., those without recursion or global synchronizations), it is easy to implement on GPU. Computations in each stage are scheduled using the default hardware scheduling mechanism of GPU.



(a) Run to completion.

(b) Kernel by kernel.

(c) Megakernel.

Figure 3: Execution models of *Run to completion, Kernel by kernel* and *Megakernel.*

Kernel by kernel (KBK). KBK is another common model to implement pipeline programs on GPU. In this model, multiple kernels are used to realize a pipeline program, where each kernel represents one or multiple stages. The pipeline logic control is implemented on the CPU side. Stages in different kernels communicate by global memory. Figure 3(b) shows this model. This is a simple model which can realize all kinds of pipeline computations, even for the ones with recursions or global synchronizations.

Megakernel. Megakernel organizes pipeline computations into a huge kernel and schedules each stage with a software scheduler. Figure 3(c) shows this model. A technique called persistent thread [3,

6, 16] is used to implement Megakernel. In the *persistent thread technique*, the kernel creates a number of threads that stay alive throughout the execution of a kernel function, usually through the use of a while loop in the kernel as Figure 3(c) illustrates. These threads continuously fetch new data items from the queue of each stage and execute the corresponding stage based on data items they fetched. These threads are called *persistent threads*. The method allows certain scheduling control of the tasks, but as all the threads are in a conventional kernel, the scheduling of them is still through the underlying hardware scheduler.

4.2 Enriched Set of Execution Models for Pipeline

Although there have been some previous studies on offering support for pipeline computations on GPU, they mainly focus on one type of execution model. There have not been systematic studies on the broad space of possible execution models for pipeline workloads on GPU. In this part, we firstly present the strengths and weaknesses of previous execution models in detail. Secondly, we propose two new execution models for pipeline computations through leveraging a software approach to circumvent the limitation of current hardware-based scheduling. Finally, we propose a hybrid execution model that combines the strengths of all previous execution models and the two newly proposed models.

4.2.1 Understanding Previous Pipeline Execution Models. To better guide the design and implementation of pipeline computations, this section qualitatively analyzes the main strengths and weaknesses of various execution models for pipeline computations. (Section 8 gives quantitative comparisons.)

RTC. In this model, since all stages are within one kernel, shared memory can be employed for data communication across different stages and data locality is relatively good.

Although this model is simple to implement, the weaknesses are obvious. First, hardware resources cannot be fully utilized for some pipeline computations. In this model, all the stages are within one kernel and hardware resources such as shared memory and registers are shared in different stages. Hence, if some stages in a pipeline consume too much hardware resource, it can lead to a low occupancy for this kernel, which further lowers the performance. Second, if global synchronization between stages is needed, it is infeasible to implement such a pipeline in the RTC model as global synchronizations are not supported in conventional GPU kernels. Finally, the instruction cache on GPU is relatively small; a kernel with a large code size can hence suffer, affecting the program performance.

KBK. In this model, as the whole pipeline is separated into several kernels, each kernel will consume fewer hardware resources and the occupancy of the GPU kernel is much larger than the RTC model. Moreover, these small kernels are more efficient for the instruction cache.

On the other hand, there are several problems with this model. First, in this model, there is an implicit synchronization between every two consecutive kernels as all dependent kernels get executed in sequential order. Thus a small number of long-running threads could severely delay the start of the next stage. Second, the model may cause frequent kernel launches, especially for a pipeline with recursion or loops. The launching overhead could be substantial [32]. Moreover, the CPU-side involvement that the model needs for coordinating the pipeline control adds further overhead.

Megakernel. In this model, as different stages are put into one single kernel with persistent thread technique, it can avoid the need of global synchronizations between different pipeline stages, allowing more parallelism among stages to be exploited. As this model only has one kernel, shared memory can also be used for data communication across different stages. Using the Megakernel model, different stages in the Reyes rendering pipeline can be executed concurrently and the stage that cannot fully utilize GPU resources by itself, like `Split`, has little impact on the whole program.

This model still suffers from some limitations. The fundamental shortcomings of the Megakernel design stem from the lack of controllability. Although the persistent thread technique can decide what stage of the pipeline a thread helps with at each time, it cannot determine where (or on which SMs) a stage runs, what stages shall be put together onto one SM, which ones shall avoid co-run together, and how the GPU resources shall be allocated to the different stages. Lack of such controllability can significantly hurt the program performance for some pipeline computations. Furthermore, putting all stages into a single kernel, Megakernel-based models cause large register and shared memory pressure on GPU. For instance, each thread of the Reyes program in Megakernel uses 255 registers and each SM can only launch 1 thread blocks. In contrast, for the KBK model, 5 blocks can be launched on one SM for the *Shade* stage. In addition, like the RTC model, Megakernel has a large code size, which is not efficient for instruction cache.

4.2.2 Exploring New Pipeline Execution Models. Due to the limitations of the aforementioned existing models, a substantial room is left for improvement (as much as several folds) of the computing efficiency of pipeline computations, as Section 8 will detail.

This section describes two new primary execution models we have developed. They are designed to achieve better controllability of task and thread scheduling such that better mappings can be created between pipeline stages and SMs, and hence lead to better resource utilizations.

The new models leverage the combination of persistent threads [3] and another technique called *SM-centric mechanism* [50]. *SM-centric mechanism* is a way to enable precise controls of the mapping of the tasks (a task is the set of operations needed to process a data item in a pipeline stage) of a GPU kerenl to the GPU SMs. It is through transforming the kernel code such that each GPU thread decides which tasks to process based on the ID of the SM on which the thread runs. More specifically, it uses a method called *SM-based task selection* to guarantee task execution is based on a specific SM and enable the binding between tasks and SMs, and uses a *filling-retreating scheme* to offer a flexible control of the amount of active threads on an SM. (See [50] for details.) Persistent threads, on the other hand, can help control which stages of a pipeline application map to which threads, their execution order, and enable cooperations across CTAs. Therefore, through combining these two techniques, we can implement a precise control of the mapping from pipeline stages to SMs. In this subsection, we describe two new execution models for pipeline applications based on this idea, *coarse pipeline* and *fine pipeline*.

Coarse Pipeline. We first propose a coarse pipeline execution model based on the SM-centric technique. Figure 4 shows this model. In this model, each stage is implemented in a single kernel like the KBK model, but with persistent threads. It puts each of the kernels into a CUDA stream, such that through these streams, the kernels can get launched at the same time on a GPU. One of its main features is that through persistent threads and the SM-centric scheme, it ensures that each stage is bound to one or more SMs and each SM can only run a single stage. As different stages are executed on different SMs, they can concurrently run without interfering with each other. With the coarse pipeline model, we can allocate SMs for each stage based on their actual requirements.

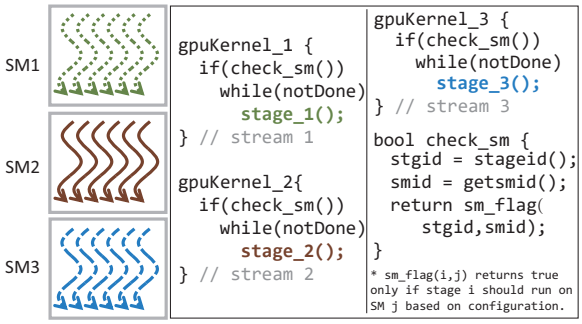


Figure 4: Execution model of coarse pipeline. Each stage is bound to specific SMs exclusively. All stages can execute concurrently.

In coarse pipeline, no global synchronization between stages is needed and different stages can execute concurrently. Each stage maintains an input task queue, which serves as a buffer for data communication between stages. A task of a previous stage fetches input data from its queue, executes this stage, and finally puts output data into the queue of the next stage, which is repeated until all stages finish. The avoidance of global synchronization helps it avoid the associated overhead that RTC and KBK suffer. Another advantage of coarse pipeline is that because stages are put into different kernels, the hardware usage for each kernel is reduced such that more threads can be launched concurrently compared to Megakernel. For example, in the Reyes rendering pipeline, the first and the third stage use 111 and 61 registers respectively with coarse pipeline, while 255 with Megakernel. In addition, smaller kernels can lead to better usage for instruction cache on GPU. A disadvantage of this model is that it misses the cases, in which, multiple kernels co-run on an SM could be better than they run separately: They, for instance, either have complementary resource needs or share some data that can benefit from the on-chip (shared) data cache.

Fine Pipeline. In coarse pipeline, as SM is a basic assignment unit for each stage, SMs cannot be fully utilized for some pipelines. A simple example is that a stage has limited workload and can use only a half of an SM; there would be a half of the SM resource wasted if this stage is assigned to the SM exclusively. To address this limitation, we further propose a fine pipeline execution model, as shown in Figure 5. Like coarse pipeline, different stages are still implemented in different kernels with persistent threads and communicate with each other by a task queue. Pipeline stages in fine pipeline are still bound to specific SMs. The difference is that in fine pipeline, one SM can accommodate more than one stage. It makes it possible for multiple stages to fill up under-utilized SMs. In fine pipeline, thread block is a basic unit for stage mapping. With the SM-centric technique, the count of thread blocks on a specific SM can be controlled for a stage, but the total count of thread blocks executed on an SM is limited by available hardware resources.

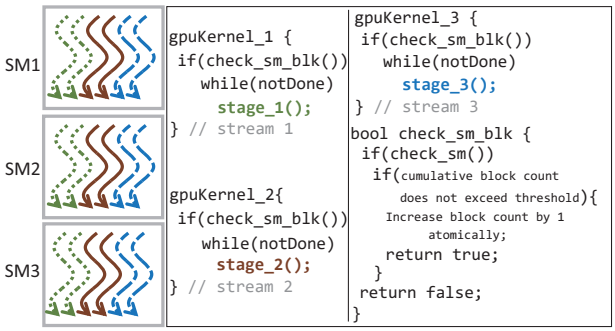


Figure 5: Execution model of fine pipeline. Each stage is assigned to several blocks on several SMs and each SM can execute several stages.

In fine pipeline, as different stages can be executed on the same SM, the on-chip cache can be better utilized between different stages. Fine pipeline also allows fine-grained hardware resource allocation and hence offers more flexibility than coarse pipeline does. A better load balance can be achieved by filling up an SM with several stages consuming different hardware resources. The main drawback is that the configuration of fine-grained block mapping is tricky. It is not easy to get the optimal performance from a large number of optional configurations.

4.2.3 Hybrid Pipeline. We have examined the space of execution models for pipeline computations and analyzed the strengths and weaknesses of each model and also proposed two new execution models. We give 7 metrics to summarize the strengths and weaknesses qualitatively.

Applicability The metric of applicability indicates whether a given execution model is practical for a variety of pipeline computations, such as pipelines with recursion or global synchronizations.

Task parallelism This metric reflects whether a given execution model can exploit parallelism across different pipeline stages, which is important for a pipeline program to fully utilize GPU resources.

Hardware usage This metric denotes hardware usage for a given execution model, such as register files and shared memory, which can affect the number of threads that can be concurrently executed on a GPU.

Load balance This metric denotes whether it is easy to balance workload among GPU streaming multiprocessors or threads.

Data locality This metric denotes whether the data access in an execution model has good locality.

Code footprint This metric denotes the code size of a pipeline program in a given execution model, which affects the efficiency of instruction cache.

Simplicity control This metric means whether an execution model is easy to be configured to reach its optimal performance, which reflects programming efforts.

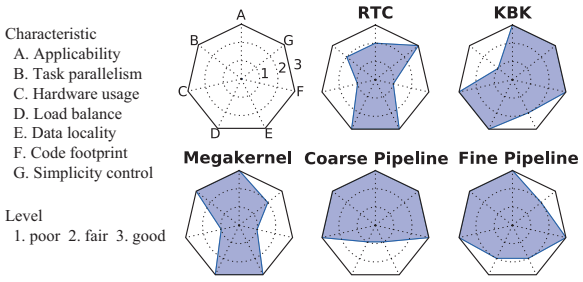


Figure 6: Characteristics of each pipeline model.

Figure 6 shows the strengths and weaknesses of each pipeline execution model we have discussed. We can see that each model has strengths and weaknesses so that no single model can outperform the other models in all metrics. In a large pipeline program, different stages may have different workload characteristics and the pipeline structure itself may be complex so that there is no single execution model meeting all the requirements. It is desirable to mix different pipeline execution models in a complex pipeline application and combine the strengths of each model. To this end, we propose a new execution model, called hybrid pipeline, which combines all these execution models to meet the needs of various kinds of pipeline programs.

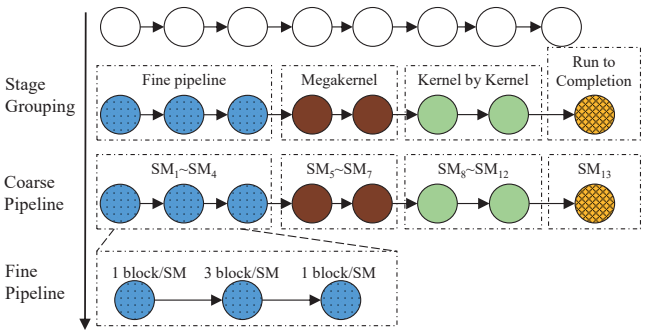


Figure 7: An example of hybrid pipeline model.

In hybrid pipeline model, we partition stages of a pipeline program into several stage groups based on the workload characterization of each stage. Each stage group adopts a suitable execution model based on its features. The execution model between stage groups is a coarse pipeline model. That is, each stage group is bound to several SMs and each SM is dedicated to execute the functions of this stage group. For stage groups that need to be executed in a fine pipeline model, a mapping strategy of the thread blocks they contain should be decided. Figure 7 uses a simple example to illustrate how the hybrid model works. This pipeline program consists of 8 stages. First, 8 stages are partitioned into 4 groups and a specific execution model is selected for each group. Second, each group is assigned to several SMs based on their resource requirements. The final step is to decide how many thread blocks shall be used on an SM for each of the stages in a group that uses the fine pipeline model.

5 VERSAPIPE LIBRARY

VersaPipe library realizes the combination of execution models with two levels of scheduling controls, high-level and low-level controls. The high-level control puts stages into different kernels and do SM assignment for them, which corresponds to *SM Mapping Controller* and *Block Mapping Controller*. The low-level control performs task scheduling inside each kernel, which corresponds to *Task Scheduler* and *Work Queue Library*, as shown in Figure 2.

In VersaPipe, we firstly perform the high-level control among all stages of a pipeline program. With the configuration generated by the auto-tuner, the stages are partitioned into several stage groups, with each group assigned with a specific execution model. For Megakernel and RTC groups, all stages are put into one kernel. Stages in coarse or fine pipeline or KBK groups will be put in separate kernels. Except the kernels in KBK groups, we put each kernel into a separate CUDA stream so that they can execute concurrently with different hardware resources. After grouping, we leverage SM-centric technique to implement SM and block mapping. *SM Mapping Controller* binds all these kernels onto target SMs. In addition, for fine pipeline groups, *Block Mapping Controller* maps each stage to specific blocks on corresponding SMs.

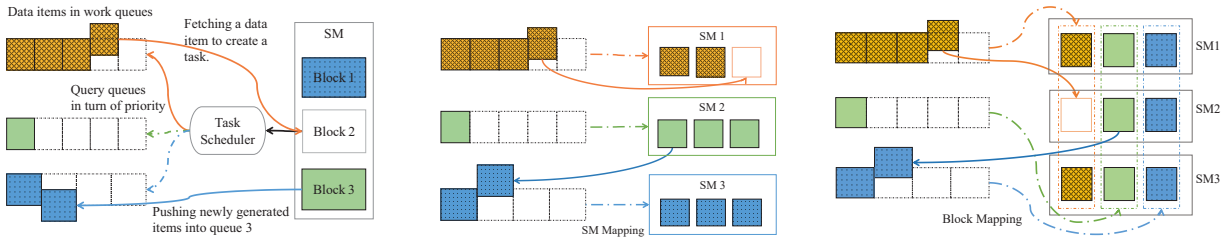
The implementation of SM mapping uses a scheme employed in the SM-centric mechanism. The kernel code is set up such that at the beginning of each kernel, VersaPipe launches enough thread blocks that can run concurrently on each SM. In the kernel code, each thread block first checks whether it should execute on its current SM according to the SM mapping configuration. If not, that block will exit. Thus only the blocks that should be assigned to this SM will remain.

The implementation of block mapping is similar with SM mapping. The main difference is that, in addition to SM checking, each kernel also checks block mapping status. Each stage on each SM maintains a block counter whose initial value is 0. After SM checking, *Block Mapping Controller* will check whether the current block counter for this stage on this SM has exceeded the specified block count. If true, this block will exit. Otherwise this block counter will increase by one.

The basic concepts of the low-level control in VersaPipe framework are listed below.

- **Data Item** A basic data unit to be processed for each stage. One data item can be processed by either one thread or multiple threads on GPU.
- **Task** An instance that executes a data item in a stage. This is the basic processing unit during execution.
- **Work Queue** A buffer that is used for data communication between stages. A previous stage pushes data items into a queue and the next stage pulls them from it, and executes them.
- **Task Scheduler** Determining the strategy of fetching data items from each work queue. Task scheduler guarantees load balance of the program.

In the low-level control, both RTC and KBK models schedule tasks with basic GPU hardware scheduler, while Megakernel, coarse



(a) Megakernel. (b) Coarse pipeline. (c) Fine pipeline.

Figure 8: Mechanisms of low-level control in VersaPipe for Megakernel, coarse pipeline, and fine pipeline.

pipeline, and fine pipeline adopt additional software scheduling strategies. The main body of the software-scheduled kernel is a loop using persistent threads. Each block of the kernel fetches data items from a work queue and executes tasks continually. A task scheduler is in charge of querying queues of all corresponding stages in a specific order. After a block executes its tasks, it pushes results into the queue of the target stage and tries to fetch and execute new tasks again. Task parallelism is realized as different blocks may pull data items from different queues and execute tasks of different stages at the same time. Figure 8 shows the mechanism of the low-level controls for Megakernel, coarse pipeline, and fine pipeline. A pipeline program with three stages maintains three work queues in each model. In Megakernel, thread blocks fetch tasks through a task scheduler, while in coarse/fine pipeline, each SM and also each thread block are bound with specific queues, from which blocks fetch tasks directly.

With the aforementioned mechanisms and methods, the execution models in each stage group can be realized, and all stage groups communicate with each other using work queues. The combined execution models for the whole pipeline can be hence materialized when the configurations are given to the VersaPipe framework.

6 API

VersaPipe provides a C++ template based API to implement pipeline programs efficiently. Developers can easily implement pipeline programs in VersaPipe framework. The implementation of a pipeline program contains two steps: defining each stage of the pipeline by extending a base class, and inserting initial elements into the pipeline. There is an optional step, pipeline execution model configuration, which can be provided automatically by the auto-tuner in VersaPipe.

Figure 9 shows a simple example in VersaPipe framework for a 3-stage pipeline. Due to the limitation of space, we show the code of the first stage only, which is a recursive stage. In that stage, each data item is multiplied by 2. If the value reaches a threshold, the data item is enqueued for stage 2 to process; otherwise, it goes through stage 1 again.

To define a stage, developers need to create a class by inheriting a built-in class `BaseStage`. As Figure 9 illustrates, there are 3 main fields to implement: defining the type of a data item in the stage, indicating the number of threads for processing a data item, and defining the `execute` function of the stage. The `execute` function is used by the threads of that stage to process each data item in the input queue of that stage. The parameters of this function are the pointer to the data item fetched from the queue and the ID of the current thread. The function of *enqueue<StageClassName>(itemVal)* is used to enqueue a data item (*itemVal*) to stage *StageClassName*. Developers call the function of *insertIntoQueue()* to push initial data items into the input queue of the target stage. The parameters of this function are an array of the initial data items and its size.

```
class Stage_1 : public BaseStage {
    typedef int DataItemType;   // Element type is int
    int threadNum = 1;          // Each task has 1 thread

    void __device__ execute(
                DataItemType * data, int threadid) {
        int val = *data;    // Data fetched from queue
        val *= 2;
        if (val >= THRESHOLD)
            enqueue<Stage_2>(val);
        else
            enqueue<Stage_1>(val);
    }
};

…; // Definition of Stage_2 and Stage_3

// push 10 initial data items into queue
VersaPipe::insertIntoQueue<Stage_1>(initItems, 10);
```

Figure 9: Programming example in VersaPipe framework.

These simple interfaces screen programmers from concerning the complexities in implementing and optimizing the interactions and performance interplays among different stages of a pipeline. If we implement stage 1 with the original GPU interfaces, the recursive pipeline structure would require either dynamic-parallelism, whose recursive depth is limited, or multiple kernel calls controlled by CPU. Both methods need much higher programming efforts than the use of the VersaPipe API. Programs written with VersaPipe look like sequential programs in most cases. Porting existing pipeline programs with VersaPipe needs only few changes.

It is worth noting that the definition of data items determines the granularity of a task for a stage, and hence affects the efficiency of the program. Sometimes, combining several basic data elements into a composite type can reduce the number of data items to be processed as well as the needed queuing operations. For example, in *CFD* application (described in Section 8), combining 1024 elements into one composite data item yields much better performance than using a single data item as an queuing unit.

7 VERSAPIPE AUTOTUNER AND RUNTIME ADAPTATION

A configuration for a pipeline application consists of the appropriate stage grouping, SM mapping, as well as block mapping for fine pipeline as Figure 7 illustrates. The search space for the optimal configuration is usually large as it is the product of all the possibilities of these three aspects. Different configurations can affect performance heavily; it is hence necessary to provide a method to help developers explore the huge selection space systematically.

We provide an auto-tuner tool to do that. The auto-tuner consists of an offline part and an online part. The former generates an initial configuration of the pipeline, and the latter supports online adaptive mechanisms to adjust the configuration.

```
autotuning() {
  G = set of all possible stage grouping strategies
  for g in G:
    S = set of all possible SM mapping strategies for g
    for s in S:
      B = set of all possible block mapping strategies for s
      // prune for B: 1. each stage has a maximum count
      // of blocks on an SM; 2. a stage will have the same
      // number of blocks on each of assigned SMs
      for b in B:
        config = {g, s, b}
        // set timeout 'mintime' and execute program
        newtime = timeoutexec(mintime, config)
        if newtime < mintime:
          minconfig = config
          mintime = newtime
}
```

Figure 10: Pseudo-code of the offline auto-tuner.

The offline auto-tuner first searches all possible grouping strategies. To limit the searching space, a stage can only be grouped with its neighbouring stages. It then explores all possible models for each group: RTC, KBK, Megakernel, and fine pipeline. Also, all possible SM mappings are attempted, as well as all possible block mappings for fine pipeline groups. Some constraints are added to prune the searching space of block mapping. One is the upper bound of block counts for each stage on each SM, which is the maximum number of blocks that can run concurrently on each SM for each stage. Another constraint we add is that, in a fine pipeline stage group, a stage will have the same number of blocks running on each of its assigned SMs. With these constraints, the searching space is limited to a much smaller size. Finally, offline auto-tuner executes the pipeline program with all the configurations and measures the execution times. A timeout value is set to limit the execution time of each run and this value is updated when a run completes its execution with shorter time. The configuration with the shortest execution time is chosen as an initial hybrid pipeline model of the program. Figure 10 is the pseudo-code of the offline auto-tuner. The metrics required by the tuner is the maximum count of blocks that can run on a SM for each stage, which is collected by the *Profiling Component*.

The online adaptation is enabled through a coordination between the kernels and the host thread. The adaptation does not adjust stage groupings as most pipeline applications in practice have fixed quite stable behavior patterns and inter-stage relations. The adaptation is mainly about dynamically adjusting SM mappings and block assignments. When a thread block does not have any task to execute, the threads raise a flag on the pinned host memory and exit. At seeing the flag, the host thread notices the availability of some extra computing resource on GPU. It launches new kernels to fill the underutilized SMs. During the process, it chooses the stage group with the most data items stalled in its queues. Such dynamic adjustments incur some changes to the initial SM mappings and block assignments. The use of queues make the adjustments easy and safe to do.

8 EVALUATION

In this section, we evaluate our proposed VersaPipe framework using a variety of real pipeline applications with different pipeline structures. To demonstrate the benefits of the VersaPipe framework over previous work, we compare it with the basic pipeline execution models, such as RTC and KBK, which are the original versions for those applications. We also compare our method with a state-of-the-art Megakernel model, called Whippletree [44].

Table 1: Various pipeline applications used for evaluating VersaPipe framework.

Applications	Description	Stage Count	Pipeline Structure	Workload Pattern
Pyramid	Image Pyramid	3	Recursion	Dynamic
Face Detection	LBP Face Detection	5	Recursion	Dynamic
Reyes	Reyes Rendering	3	Recursion	Dynamic
CFD	CFD Solver	3	Loop	Static
Rasterization	Image Rasterization	3	Linear	Dynamic
LDPC	LDPC Decoder	4	Loop	Static

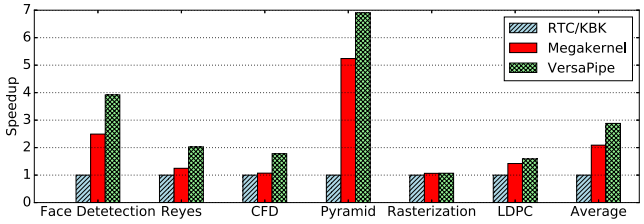
8.1 Pipeline Applications

We evaluate VersaPipe with a diverse set of pipeline applications shown in Table 1. `Image Pyramid` and `Face Detection` are implemented based on the work of Oh et al. [31]. `Reyes` and `Rasterization` are ported from the source codes released by Patney et al. [35]. `CFD` is from Rodinia benchmark [8]. `LDPC` is implemented based on an open source version [18].

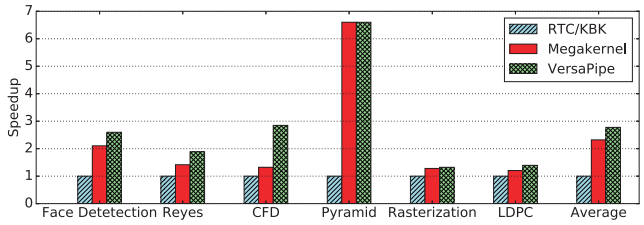
The main features for a pipeline program include stage count, pipeline structure, workload patterns, and hardware usage. Table 1 shows these features for evaluated pipeline applications. The stage count ranges from 3 to 5. We classify the pipeline structure by linear, loop, and recursion. Linear pipelines are the simplest structures. Recursion pipelines have recursive executions and loop pipelines have iterations over different stages. The workload patterns are either static or dynamic, regarding whether the amount of workload of each task in a stage changes on different inputs. Together, these applications provide a diverse set for evaluating the performance and applicability of VersaPipe. We first provide an overview of the overall results, and then analyze the results of each application in detail.

8.2 Overall Results

We experiment on two models of GPU, K20c GPU with 13 SMs and GTX 1080 GPU with 20 SMs. The CUDA version is 7.5. The CPU is Intel Xeon E5-2620. Figure 11 shows the overall results for all the applications in the basic pipeline execution models used in the original benchmarks (RTC or KBK), Megakernel, and VersaPipe. All timing results are the average of five repeated runs and no



(a) Speedup on K20c.



(b) Speedup on GTX 1080.

Figure 11: Overall results in different models. Speedup is normalized by the execution times of the basic models used in the original benchmark (either RTC or KBK). (8 images used for Face Detection and Pyramid)

Table 2: Experiment results on K20c.

(32 images used for Pyramid and Face Detection)

Program	Execution Time(ms)			Longest Stage	
	RTC/KBK	Megakernel	VersaPipe	time(ms)	itemSz
Pyramid	14.41	1.59	1.37	0.80	12B
Detection	18.27	9.09	5.38	5.29	16B
Reyes	15.6	12.5	7.7	4.02	272B
CFD	5820	5430	3270	2970	12B
Raster	32.8	30.8	30.7	30.6	4B
LDPC	560	394	352	185	12B

significant variances are observed. The execution time is measured on CPU side, starting from the beginning of the first kernel and ending just after the last kernel finishes. It includes all the kernel executions, CPU controls, and memory copies covered in that entire code range. All results have been normalized by the performance of the basic models, which are either RTC or KBK, depending on the original implementation for each application. The block dimension for Megakernel, coarse pipeline, and fine pipeline are all 256 in our experiments.

The input of `Image Pyramid` and `Face Detection` is 8 HD (1280 × 720) images. The rendering result of `Reyes` is a teapot in a resolution of 1280 × 720 pixels. The input of CFD program is the `missile` data set provided by Rodinia benchmark. The `Rasterization` application rasterizes 100 cubes and outputs pictures in 1024 × 768 resolutions. In `LDPC`, we set both the number of frames and iterations as 100. The performance of each application also changes with different inputs, and we will discuss them in detail in Subsection 8.5.

In general, the results on both K20c and GTX 1080 show that for all the applications, VersaPipe has a better performance than the basic models and Megakernel. On K20c, VersaPipe has achieved

up to 6.90× (2.88× on average) better performance than the basic models and 1.66× (1.38× on average) better performance than Megakernel. All versions of the kernels run much faster on GTX 1080. For instance, the baseline versions of the kernels finish in 11-44% of the time they take on K20c. That reflects the benefits of the extra resource and more advanced designs of GTX 1080. The benefits of VersaPipe remains. It still significantly outperforms both the baseline and the Megakernel versions, yielding 2.7× and 1.2× speedups over them.

8.3 Detailed Results

Image Pyramid Image Pyramid is a widely used approach in image processing and computer vision domains to find an object of unknown size. It produces a set of down-sampled images from the given image [1, 4, 28]. The practical use of Image Pyramid ranges from real-time applications such as robot vision [41] to video analytics with surveillance cameras or satellites [13]. We construct a typical Image Pyramid with 3 stages as shown in Figure 12, where the `Resize` stage has a recursive execution. Note that, the amount of workload in Image Pyramid decreases as it reaches a high level of the down-sampling process (i.e., getting close to the top of the pyramid of samples), which causes the GPU to be under-utilized in the basic KBK model. For example, the workload size and the computation time for an HD input in the `Resize` stage vary by up to 239 times and 4.94 times respectively as the resized image becomes smaller. However, the most severe performance bottleneck in Image Pyramid is `Histogram equalizaiton` which has a limited degree of parallelism and contains a serial portion that cannot be parallelized. It can be executed only with 256 threads in a block. As a result, most of SMs are idle in `Histogram equalizaiton` and it takes 96.1% (13.85 ms) of the time in Image Pyramid. In contrast to this execution in KBK model, VersaPipe can make a full use of the GPU resources by effectively combining task-parallelism and data-parallelism.

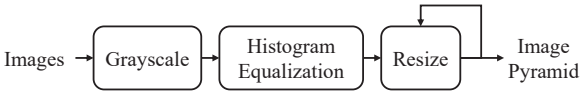


Figure 12: The pipeline structure of Image Pyramid.

Figure 13 shows the experimental results of Image Pyramid for processing different numbers of input images in different models. KBK with Stream is an improved model over KBK. In this model, different images are processed in multiple GPU streams, which alleviates the limited parallelism in KBK model. VersaPipe outperforms both execution models significantly since it avoids large kernel launch overhead and achieves good load balance. VersaPipe shows a much better performance than Megakernel does for all the cases, thanks to its effective exploitations of both the coarse pipeline and the fine pipeline models.

In VersaPipe, we can get a proper configuration for the hybrid execution model with the VersaPipe auto-tuner. Image Pyramid stages are separated into two groups to get benefits with coarse pipeline model. One is for `Grayscale` with 4 SMs and the other is for `Histogram equalization` and `Resize` with 9 SMs. These stages launch 6, 2, and 2 thread blocks on each SM, respectively. Although the maximum numbers of blocks in an SM for `Histogram equalization` and `Resize` are originally 3 and 4, fine pipeline

model allows them to share the hardware resources and makes it feasible to execute 4 blocks (i.e., 2 blocks each) all the time. As a result, VersaPipe maintains a total of 60 blocks, while Megakernel only 39 blocks. When the input size is small (less than 5 images), the performance difference of the different models is less prominent due to the limited amount of work to do; in those cases, the performance is in general less of a concern.

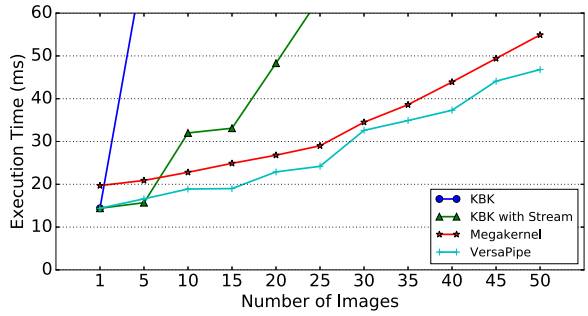


Figure 13: Performance results of Image Pyramid.

Face Detection Face detection is a real-life application, which tries to detect faces from a given image. In this paper, we use a face detection application which adopts Local Binary Pattern (LBP) based classification [2].

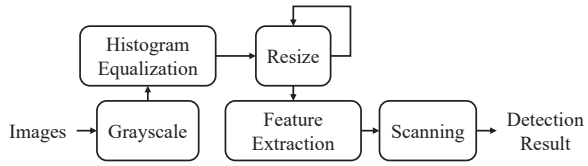


Figure 14: The pipeline structure of face detection.

There are 5 stages in this pipeline with a recursive structure as shown in Figure 14, where `Feature extraction` and `Scanning` are processed after Image Pyramid. In the `Scanning` stage, the threads whose assigned searching windows contain no face will terminate earlier than others with faces [48], which could result in severe load-imbalance between threads. Thus we define a search window as a basic data item in VersaPipe to achieve a good load-balance.

VersaPipe gives 3.4× speedup over KBK. One of the main reasons for this improvement is from `Histogram equalization`, which has limited degree of parallelism and cannot fully utilize GPU resources in KBK. Note that, in KBK, we try to minimize performance degradation due to this lack of parallelism, by dividing `Histogram equalization` into three separate kernels, where two kernels are executed in pixel-level parallelism. Nevertheless, KBK still shows worse results than our framework.

Compared to Megakernel, VersaPipe achieves better performance in all cases. First, VersaPipe reduces register usage which limits the maximum number of active thread blocks in Megakernel. Each thread in Megakernel consumes 87 registers, while in VersaPipe, each kernel by a thread consumes 56, 69, 56, 61, 37 registers respectively. Megakernel can only launch 2 concurrent blocks in an SM, while VersaPipe can launch at least 3, or at most 6 blocks for different stages concurrently. Also, with fine pipeline model, an SM can execute different kernels concurrently. Since most tasks are memory-intensive, cache affects the performance significantly. For this case, the fine pipeline model provides large performance enhancement by utilizing L1 cache more efficiently with good data locality.

Reyes Rendering Reyes rendering was proposed by Cook et al. [12], which is widely used in modern renders, like Pixal's Renderman [36]. Our implementation is based on the source code released in [35], in which `bound` and `split` are implemented as one stage, referred to as `Split`. The pipeline structure is shown in Figure 1. This pipeline is recursive and has dynamic workload generation between neighbouring stages, which makes it impractical to deal with RTC model. We implement `Reyes` with KBK, Megakernel, and VersaPipe. Table 2 shows VersaPipe has a 2.03× speedup over KBK and a 1.62× speedup over Megakernel.

The KBK implementation contains 16 kernel calls. Memory copies and recursive control on CPU introduce large overhead. The workload in `Split` are not able to fully utilize the GPU, but implicit synchronizations between kernels make task parallelism between stages impossible. With auto-tuning, VersaPipe combines fine pipeline and Megakernel. CPU-GPU switching is eliminated and task parallelism is also introduced to fully utilize the GPU.

Megakernel outperforms KBK, but performs worse than VersaPipe as Megakernel consumes too many registers in the single kernel. Megakernel consumes 255 registers per thread, while the 3 kernels in VersaPipe consume 111, 255, 61 registers respectively. The high register usage comes from `Dice`, while `Split` and `Shade` use less. In VersaPipe, `Split` and `Dice` compose a stage group with fine pipeline model, while `Shade` composes another group with Megakernel model. By separating into different kernels, threads executing in `Split` and `Shade` consume fewer registers and then GPU can launch more blocks concurrently. `Dice` consumes too many registers that one SM can only execute one block. Although the remaining hardware resources are not enough for a `Dice` block, it is sufficient for a `Split` block, enabling each SM to execute one block for `Split` and another block for `Dice` in VersaPipe. Therefore, `Split` and `Dice` can make full use of an SM with fine pipeline model. Finally, there are 35 blocks launched concurrently in VersaPipe, while the count for Megakernel is only 13.

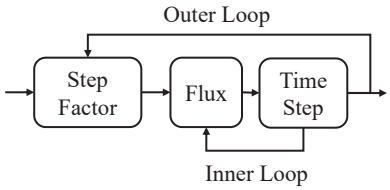


Figure 15: The pipeline structure of CFD.

CFD CFD (Computational Fluid Dynamics) [17] is an important branch in fluid mechanics. We use the implementation in the Rodinia benchmark [8], whose pipeline structure is shown in Figure 15. There are 3 stages and 2 iterations in this pipeline. The count of inner iterations is 3 and the count of outer iterations is 2,000. The execution time is 5.82 seconds, 5.43 seconds, and 3.27 seconds for KBK, Megakernel, and VersaPipe, respectively.

KBK makes 14,000 times of kernel calls. Frequent switching between CPU and GPU introduces large overhead. Furthermore, the occupancy of `Flux` is only 37.5%. Without task parallelism, the

GPU utilization is very low. With fine pipeline model, VersaPipe eliminates CPU-GPU switching overhead by reducing the count of kernel launches to 3 and utilizes GPU better by introducing task parallelism.

In Megakernel, each SM can only execute 2 blocks as `Flux` consumes too many registers. VersaPipe separates 3 stages into 3 kernels such that threads executing `Step Factor` or `Time Step` consume fewer registers. With VersaPipe, the block count that can be executed concurrently on each SM are 4 for `Step Factor`, 2 for `Compute Flux`, and 3 for `Time Step`. Thus, VersaPipe achieves much higher throughput by launching more blocks.

As Figures 11 shows, VersaPipe gives a higher speedup on GTX 1080 than on K20c for CFD. It is because as kernels execute faster on GTX 1080, the launching overhead of the baseline version accounts more.

Rasterization Rasterization is the most commonly used rendering technique for producing real-time 3D graphics on a computer. We implement rasterization based on the source code released in [35]. As shown in Figure 16, the pipeline includes 3 stages. The workloads of each stage changes dynamically based on different inputs.

Figure 16: The pipeline structure of rasterization.

The basic version is implemented in both KBK and the mixing of KBK and RTC models. In the mixing of KBK and RTC model, `Clip` and `Interpolate` are in the same kernel with RTC model, and `Shade` is in the other kernel. The execution time for KBK is 33.8ms, while it is 32.8ms for the mixing of KBK and RTC. It is 30.8ms in Megakernel, and 30.7ms in VersaPipe.

KBK presents the worst performance as it has the large overhead of CPU-GPU switching and memory copy, and GPU cannot be fully utilized under dynamic workload without task parallelism. By merging kernels of `Clip` and `Interpolate`, the mixing of KBK and RTC achieves better performance thanks to better data locality, fewer global synchronization, and fewer CPU-GPU switching. Both models exploit task parallelism of different stages and the execution time of the first 2 stages are overlapped with the last stage. The hybrid models in VersaPipe create a better overlap.

LDPC LDPC (Low-Density Parity-Check) [15] is a linear error correction program that transmits a message over noisy transmission channels. We realize it based on an open source implementation [18], which is in KBK model. There are 4 stages in LDPC decoding pipeline, as shown in Figure 17. There are 2 iterations in this pipeline.

In KBK, the execution time of 4 kernels themselves is 495ms, which means that the total overhead for kernel launch, CPU management, and memory copy between CPU and GPU is 65ms. Furthermore, the occupancies of the 4 kernels in KBK are all around 40%, which is because the workload is too small to fully utilize GPU without task-level parallelism. VersaPipe separates four stages into three groups, `Initialize` and `C2V` are in a fine pipeline group, and the other two stages are in two other groups with Megakernel model. VersaPipe eliminates the overhead of CPU control and memory copy, and utilizes GPU better with task parallelism between different stages. Megakernel suffers from high register usage so that each SM can only execute 4 thread blocks. On the contrary, in VersaPipe, one SM can execute 5 blocks for `C2V` or `V2C`, or 4 blocks for `Initialize` or `ProbVar`. A total of 56 blocks can be executed concurrently in VersaPipe, while it is 52 blocks in Megakernel.

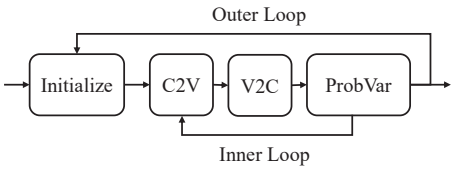


Figure 17: The pipeline structure of LDPC.

8.4 Dynamic Parallelism

Dynamic Parallelism (DP) allows threads running on GPU to launch subkernels. We use `Reyes` to compare DP with VersaPipe. When a new data item is generated in a stage, a new subkernel is launched to process it. Results show that the execution times with DP (110.6ms on K20c and 45.2ms on GTX 1080) are over 10 times longer than VersaPipe due to the large launching overhead of DP. That echoes the observations in previous studies [9, 14, 49]. It is possible that with some careful coding or advanced optimizations, the subkernel launching overhead could be reduced. But it takes much non-intuitive coding or optimization. Moreover, pipeline computations require some careful resource allocations to different pipeline stages. It remains unclear how that can be done effectively when DP is used. Finally, DP is subject to some hardware limit (e.g., on nesting depth), which could limit its applicability.

8.5 Overhead Analysis and Insights

We summarize the main reasons for the benefits of VersaPipe, and some insights on opportunities for further improvement by analyzing the main overhead of VersaPipe.

As the experiments results have shown, the benefits of VersaPipe mainly come from its reduced register usage in the kernels, fewer kernel launches, and the improved configurations of the execution models for a given application. The low register usage allows VersaPipe to launch more threads concurrently than Megakernel does. Having fewer kernel launches reduces the launching overhead. Better configurations allows VersaPipe to more fully utilize GPU resources. We found that data locality plays relatively a modest role except on Face Detection; Megakernel and VersaPipe show similar L1 and L2 performance on others.

There are two possible aspects for further improvement. The first is queue overhead. For each program, we have measured the execution time of each single stage of the pipeline while employing the same number of basic blocks as in the configurations used by VersaPipe. No queuing time is included. The second rightmost column in Table 2 reports the time taken by the longest stage, which could be regarded as the performance of VersaPipe if there was no queuing or other runtime overhead (`LDPC` is an exception because its versaPipe version folds some CPU computations into the kernel and is hence not comparable to the longest stage). From Table 2, we can see that the overhead is 10% or less on `Face Detection`, `CFD`, and `Rasterization`. It is more visible on *Pyramid*, because the

kernel is very short. Its significant overhead shows on Reyes, which features the largest data item structure in the queue as shown in the rightmost column of Table 2. More efficient queue schemes (e.g., distributed queues or hardware-based queues) could help. Methods that reduce data item size in the queues could also be beneficial.

Another potential opportunity for improvement is better load balance. Our measurement shows that even with VersaPipe, the times of stages still differ a lot. For instance, on Reyes, the longest stage takes about one third more time than the shortest stage does. Relaxing some restrictions to the search scheme of the autotuner could possibly yield configurations giving a better load balance.

9 RELATED WORK

There is a body of work on supporting task-level parallelism on heterogeneous systems. For instance, Halide [38] is a domain-specific language proposed to support the development of efficient graph processing applications. It treats a GPU as a single processing unit in the system and deploys only one GPU kernel on it each time. In contrast, our work focuses on how to best utilize the computing resource on a GPU to support multiple stages of a pipeline on a GPU.

Some work has concentrated on some particular pipeline applications and explored how to manually tune them to achieve high performance. Examples include the work on ray tracing [3, 24, 33]. They provide no general programming frameworks for pipeline applications.

Efficient work queue implementation is critical for high performance Megakernel. Access from massive parallel threads may cause severe contention and the work queue may become the main performance bottleneck. For distributed queues, access contention can be reduced but load balance will be another problem. Recent studies [7, 10, 47] have proposed efficient solutions, like work stealing and donation, to implement distributed queues with good load balance.

Steinberger et al. [44] propose a framework based on Megakernel to schedule pipeline programs with dynamic, irregular workloads on GPU. The main limitation for this model is high hardware resource usage as the previous sections show.

Wu et al. [50] study SM-Centric technique. This method offers a solution to enable program-level spatial scheduling on GPU. Kernels can be mapped to specific SMs with this method. It provides some opportunities for optimizing GPU programs with scheduling strategies. Lin et al. [27] study the preemption on GPU and realize a light-weight context switching method. Kim et al. [23] propose some hardware extensions to GPU to dynamically check dependencies and overlap different kernels.

Another orthogonal research direction is to improve hardware schedulers on GPU to reduce execution inefficiency, examples include a large warp architecture [29], a two-level warp scheduler [19, 20], instruction pattern based scheduling [25], cache conscious scheduling [39, 40], and thread block scheduler [21]. These techniques improve execution efficiency of a GPU kernel, but are orthogonal to the execution model configurations of pipeline stages.

10 CONCLUSION

In this paper, we provide a systematic examination of various pipeline execution models on GPU, and analyze their strengths and weaknesses. We further propose three new execution models equipped with much improved controllability, including a hybrid model that is capable of getting the strengths of all. We then develop a software framework named *VersaPipe* to translate these insights and models into an enhanced pipeline programming system. With VersaPipe, users only need to write the operations for each pipeline stage. VersaPipe will then automatically assemble the stages into a hybrid execution model and configure it to achieve the best performance. Experiments on a set of pipeline benchmarks and a real-world face detection application show that VersaPipe produces up to 6.90× (2.88× on average) speedups over the original manual implementations. VersaPipe is open-source and will be made available to the public.

ACKNOWLEDGMENTS

The authors would like to thank Skip Booth, John Marshall, and Robbie King for discussions at the early stage of this work. The feedback from the anonymous reviewers are helpful for improving the final version of the paper. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. 1455404, 1455733 (CAREER), and 1525609, and Cisco. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF or Cisco. In China, this work is partially supported by NSFC Project No.61232008, National Key Research and Development Program 2016YFB0200100, and Tsinghua University Initiative Scientific Research Program. This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(No. 2015R1C1A1A01055212). Jidong Zhai, Xipeng Shen, and Youngmin Yi are corresponding authors of the paper.

REFERENCES

[1] Edward Adelson, Charles Anderson, James Bergen, Peter Burt, and Joan Ogden. 1984. Pyramid Methods in Image Processing. *RCA Engineer* 29, 6 (1984), 33–41.
[2] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. 2006. Face Description with Local Binary Patterns: Application to Face Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 12 (2006), 2037–2041.
[3] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Conference on High PERFORMANCE Graphics*. 145–149.
[4] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded Up Robust Features. In *European Conference on Computer Vision*. Springer, 404–417.
[5] Christian Bienia and Kai Li. 2010. Characteristics of Workloads Using the Pipeline Programming Model. In *International Symposium on Computer Architecture*. Springer, 161–171.
[6] Michael Boyer, David Tarjan, Scott Acton, and Kevin Skadron. 2009. Accelerating Leukocyte Tracking Using CUDA: A Case Study in Leveraging Manycore Coprocessors. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–12.
[7] Daniel Cederman and Philippas Tsigas. 2008. On Dynamic Load Balancing on Graphics Processors. In *Eurographics/acm SIGGRAPH Conference on Graphics Hardware 2008, Sarajevo, Bosnia and Herzegovina*. 57–64.
[8] 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 *IEEE International Symposium on Workload Characterization*. IEEE, 44–54.
[9] Guoyang Chen and Xipeng Shen. 2015. Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*.

[10] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. 2010. Dynamic Load Balancing on Single-and Multi-GPU Systems. In *IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–12.

[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] Robert L Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. In *ACM SIGGRAPH Computer Graphics*, Vol. 21. ACM, 95–102.

[13] Carl Doersch, Saurabh Singh, Abhinav Gupta, Josef Sivic, and Alexei A Efros. 2015. What Makes Paris Look Like Paris? *Commun. ACM* 58, 12 (2015), 103–110.

[14] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. 2016. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–12.

[15] Robert Gallager. 1962. Low-density Parity-check Codes. *IRE Transactions on Information Theory* 8, 1 (1962), 21–28.

[16] 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.

[17] John L Hess and A_M O Smith. 1967. Calculation of Potential Flow about Arbitrary Bodies. *Progress in Aerospace Sciences* 8 (1967), 1–138.

[18] Jiwei Liang. 2016. LDPC OOK Decoder. https://github.com/BibbyLiang/LDPC-OOK-Decoder-on-GPU. (2016).

[19] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 395–406.

[20] Adwait Jog, Onur Kayiran, Asit Mishra, Mahmut Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*.

[21] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. 2013. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*.

[22] Brucek Khailany, William Dally, Ujval 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.

[23] 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*.

[24] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *High-Performance Graphics Conference*. 137–143.

[25] Minseok Lee, Gwangsun Kim, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2016. iPAWS: Instruction-issue Pattern-based Adaptive Warp Scheduling for GPGPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 370–381.

[26] 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.

[27] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 77.

[28] David G Lowe. 2004. Distinctive Image Features from Scale-invariant Keypoints. *International journal of computer vision* 60, 2 (2004), 91–110.

[29] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO*.

[30] NVIDIA Corporation. 2016. NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html. (2016).

[31] 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.

[32] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1–19.

[33] Steven Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David Mcallister, Morgan Mcguire, Keith Morley, and Austin Robison. 2010. OptiX: A General Purpose Ray Tracing Engine. *Acm Transactions on Graphics* 29, 4 (2010), 157–166.

[34] Anjul Patney and John D Owens. 2008. Real-time Reyes: Programmable Pipelines and Research Challenges. *ACM SIGGRAPH Asia 2008 Course Notes* (2008).

[35] 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.

[36] Pixar. 2016. Pixar's RenderMan. https://renderman.pixar.com/view/renderman. (2016).

[37] Timothy Purcell, Ian Buck, William Mark, and Pat Hanrahan. 2002. Ray Tracing on Programmable Graphics Hardware. In *ACM Transactions on Graphics (TOG)*, Vol. 21. ACM, 703–712.

[38] 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.

[39] T. Rogers, M. O'Connor, and T. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the International Symposium on Microarchitecture*.

[40] T. Rogers, M. O'Connor, and T. Aamodt. 2013. Divergence-aware Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture*.

[41] Keigo Shirai, Hirokazu Madokoro, Satoshi Takahashi, and Kazuhito Sato. 2014. Parallel Implementation of Saliency Maps for Real-time Robot Vision. In *Control, Automation and Systems (ICCAS), 2014 14th International Conference on*. IEEE, 1046–1051.

[42] 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.

[43] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: Dynamic Scheduling on GPUs. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 161.

[44] 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.

[45] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. 2009. GRAMPS: A Programming Model for Graphics Pipelines. *ACM Transactions on Graphics (TOG)* 28, 1 (2009), 4.

[46] 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.

[47] 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.

[48] 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.

[49] Wang, Jin and Yalamanchili, Sudhakar. 2014. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 51–60.

[50] 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*. ACM, 119–130.

[51] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2017. Understanding Co-running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2017), 905–918.