GPU Programming Primitives for Computer Graphics

Daniel Meister
daniel.meister@amd.com
Advanced Micro Devices, Inc.
Japan

Atsushi Yoshimura
atsushi.yoshimura@amd.com
Advanced Micro Devices, Inc.
Japan

Chih-Chen Kao
chihchen.kao@amd.com
Advanced Micro Devices, Inc.
Germany

SYNOPSIS

Various parallel algorithms can be decomposed into programming primitives that share similar patterns. This course focuses on studying these programming primitives and their applicability in computer graphics, specifically in the context of massively parallel processing on GPUs. The course begins by establishing a theoretical foundation, followed by practical examples and real-world applications. We explain two pivotal algorithms: parallel reduction and parallel prefix scan in detail, discussing their variants and different implementations. Afterward, we provide a collection of more advanced techniques and tricks applicable across various domains. At the end of the course, we also briefly discuss code optimization.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; **Massively parallel algorithms**.

KEYWORDS

Computer graphics, GPGPU, parallel computing

ACM Reference Format:
Daniel Meister, Atsushi Yoshimura, and Chih-Chen Kao. 2023. GPU Programming Primitives for Computer Graphics. In *SIGGRAPH Asia 2023 Courses (SA Courses '23), December 12-15, 2023.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3610538.3614632

1 INTRODUCTION

As parallel architectures continue to advance, the demand for parallel algorithms naturally grows. Computer scientists have been looking for inspiration in well-studied sequential algorithms. Surprisingly, it turned out that trivial operations that are straightforward when performed sequentially can become challenging when executed in parallel. This difficulty is particularly pronounced in massively parallel systems like GPUs, where thousands of threads run concurrently. There are several frequently-used patterns to address this issue. For example, a compare and swap (CAS) instruction and shared memory on the GPU are used for resolving data races efficiently. These components often require intrusive changes to the algorithm themselves. Thus, we organize them as a collection of programming primitives for computer graphics problems as design patterns on the GPU.

2 COURSE RATIONALE

Although general-purpose computing on GPUs (GPGPU) is significantly more difficult than traditional single-threaded programming, many existing courses cover only basic concepts of parallel computing. However, mastering the design and efficient implementation of parallel algorithms requires years of experience. Recently, there have been no comprehensive courses focusing on the algorithmic aspects of GPGPU.

3 INTENDED AUDIENCE

The course is designed for developers and researchers interested in GPGPU and computer graphics. The course assumes basic knowledge of GPGPU APIs, such as OpenCL, HIP, or CUDA. Although the course primarily targets an intermediate audience, even more experienced programmers can acquire new knowledge and insights.

4 PEDAGOGICAL INTENTIONS AND METHODS

Our focus is primarily on high-level concepts, prioritizing an understanding of the underlying principles rather than solely optimizing code performance. We start with motivation and high-level description of each technique, followed by a minimal code snippet illustrating the key idea of the technique. We plan to conclude the course with a Q&A session to address any lingering questions. We also provide a repository with a full version of the code presented in the slides.

SIGGRAPH Asia 2023 Courses (SA Courses '23), December 12-15, 2023, 2023.

ACM ISBN 979-8-4007-0309-6/23/12.
https://doi.org/10.1145/3610538.3614632

5 DETAILED DESCRIPTION

5.1 Introduction

We introduce the outline and main objectives of the course. We also provide a brief introduction to HIP (Heterogeneous-Compute Interface for Portability), that we use for illustrating the discussed algorithms.

5.2 Parallel Reduction and Prefix Scan

We review two important parallel programming primitives: parallel reduction and parallel prefix scan (PPS) [Blelloch 1990]. We provide an analysis of the theoretical properties of these algorithms, accompanied by an extensive discussion of diverse implementations and their respective advantages and disadvantages. We use both algorithms as building blocks for more complex techniques in the following sections.

5.3 Programming Primitives

We present a collection of more advanced techniques that are widely applicable. We start with a very simple yet ubiquitous operation (e.g., how to efficiently output data in parallel.) In computer graphics, we arrange spatial data into hierarchical structures. During the construction, we output elementary blocks, such as nodes or cells, in parallel. Another example is the task queue when we have to write new tasks to the output buffer.

The task queue itself can be considered as another primitive as it is a very general concept. We introduce a *waterfall scheme* that can be used to implement a simplified general task queue, assuming a fixed number of tasks. We present two algorithms relying on the waterfall scheme that can be implemented in a single kernel launch: device-wise parallel prefix scan and top-down traversal of the hierarchical structure, where we have to deal with parent-child dependencies. To make is complete, we add bottom-up traversal [Karras 2012] as another technique, reducing the information from leaves up to the root.

More complex algorithms require global synchronization, which is typically realized as separate kernel launches that are implicitly synchronized. However, it might be beneficial to fuse multiple kernels into a single one to decrease the kernel launch overhead and memory accesses. The global synchronization inside the kernel can be implemented through the concept of *persistent threads* [Gupta et al. 2012] that allows global synchronization, which is otherwise not possible.

For some algorithms, we need a per-thread auxiliary buffer (e.g., a buffer for the stack). Local arrays are allocated per thread, but they cause significant register pressure. On the other hand, allocating a global buffer for all scheduled threads may be too costly as only a fraction of threads are executed concurrently. One solution is to use persistent threads, but it might be difficult to change the scheduling in complex frameworks. We present a technique that allocates data only for the concurrent threads and dynamically assigns the buffers to the launched threads to address this issue.

5.4 Linear Probing

Linear probing is an algorithm to build a hash table. Since it uses open addressing, an array is used as the storage for elements. Insertion is done by linearly searching from the home location that is defined by a hash function. Thanks to the search linearity, parallel insertion can be supported on the GPU with CAS. We first show the basic algorithm of linear probing and introduce bidirectional linear probing as another variant of linear probing for better performance in case the load factor of the hash table is high [van der Vegt 2011].

5.5 Radix Sort

Radix sort is one of the efficient sorting algorithms that is suitable to run on GPUs due to the parallel nature [Harada and Howes 2011; Merrill and Grimshaw 2011]. Its design leverages offset-counting instead of comparison. In this section, we will introduce the concept of the parallel radix sort with a working example. Specifically, we will demonstrate how the aforementioned techniques, such as the parallel prefix sum, could be applied, as well as the design considerations in order to optimize the algorithm.

5.6 Code Optimization

We provide a couple of basic general recommendations to improve code efficiency, irrespective of the underlying architecture. In particular, we focus on coalesced memory accesses to the global memory, bank conflict in the shared memory, thread divergence, and occupancy.

REFERENCES

Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A Study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar).* 1–14. https://doi.org/10.1109/InPar.2012.6339596

Takahiro Harada and Lee W. Howes. 2011. Introduction to GPU Radix Sort.

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Paris, France) *(EGGH-HPG'12).* Eurographics Association, Goslar, DEU, 33–37.

Duane Merrill and Andrew Grimshaw. 2011. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters* 21, 2 (2011), 245–272. https://doi.org/10.1142/S0129626411000187

Steven van der Vegt. 2011. A Concurrent Bidirectional Linear Probing Algorithm Towards a Concurrent Compact Hash Table.