

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

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

© 2023 Copyright held by the owner/author(s).

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 it 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.

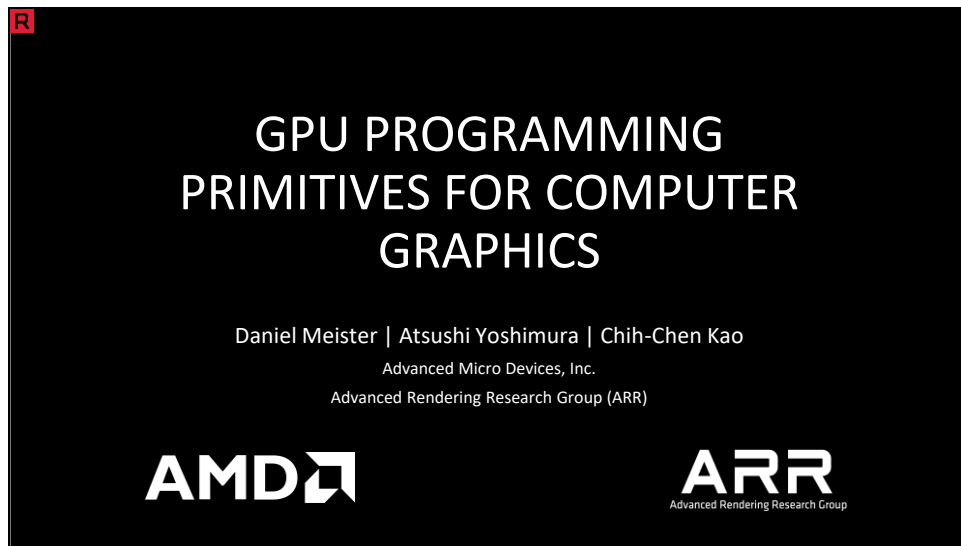
5.6 Code Optimization

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.

Slide 1



These are the notes for our course 'GPU Programming Primitives for Computer Graphics'.

Slide 2



In the introductory part, we briefly describe the course organization and course objectives. We also provide a minimal introduction to HIP that we use to illustrate the algorithms.

Slide 3



3

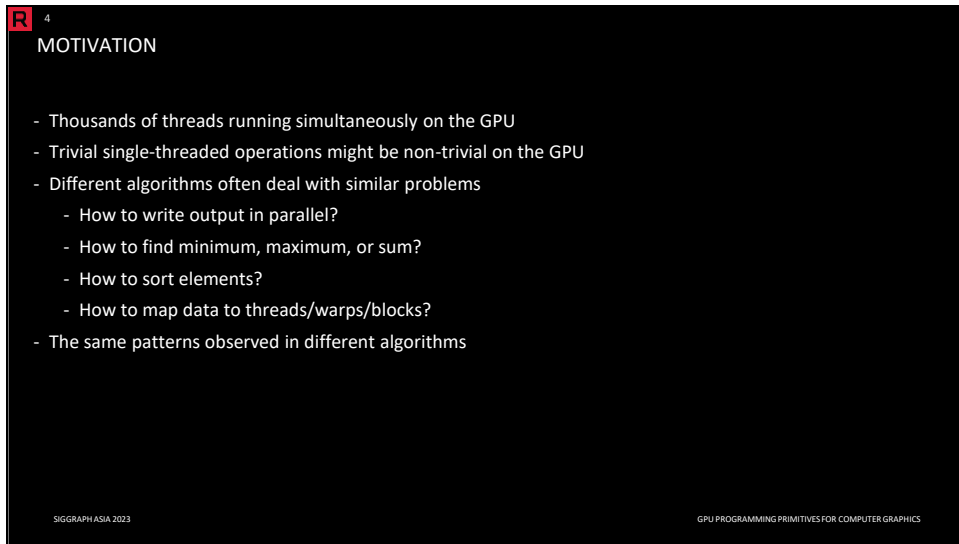
COURSE SYLLABUS

- Introduction (~15 min, Daniel)
- Parallel reduction and prefix scan (~25 min, Daniel)
- Programming primitives (~25 min, Daniel & Atsushi)
- Linear probing (~20 min, Atsushi)
- Radix sort (~15 min, Atsushi)
- Code optimization (~10 min, Atsushi)
- Q&A (~10 min)

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Our course consists of the following six sections and discussion, taking 105 minutes in total: introduction, parallel reduction and prefix scan, programming primitives, linear probing, radix sort, code optimization, and Q&A.

Slide 4



4

MOTIVATION

- Thousands of threads running simultaneously on the GPU
- Trivial single-threaded operations might be non-trivial on the GPU
- Different algorithms often deal with similar problems
 - How to write output in parallel?
 - How to find minimum, maximum, or sum?
 - How to sort elements?
 - How to map data to threads/warps/blocks?
- The same patterns observed in different algorithms

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Programming massively parallel systems such as GPUs is difficult due to running thousands of threads simultaneously. Many operations that are straightforward on the CPU are non-trivial on the GPU. We can observe that some parts of different algorithms resemble each other. In this course, we study these patterns and introduce how to handle the operations that are simple single-threaded but difficult on the GPU.



COURSE RESOURCES

- Course notes:
 - Presentation slide in PPT with animations
 - PDF with additional notes
- Code samples:
 - Buildable code presented in the slides
 - Performance comparison of different variants

<https://gpu-primitives-course.github.io>




Using this link or QR code, you can access the course webpage with course resources, including presentation slides, PDF, and sample code.

Slide 6

6

Q&A

- Feel free to send us any question through or after the course through the QR code
- We will answer reminded questions on the course page

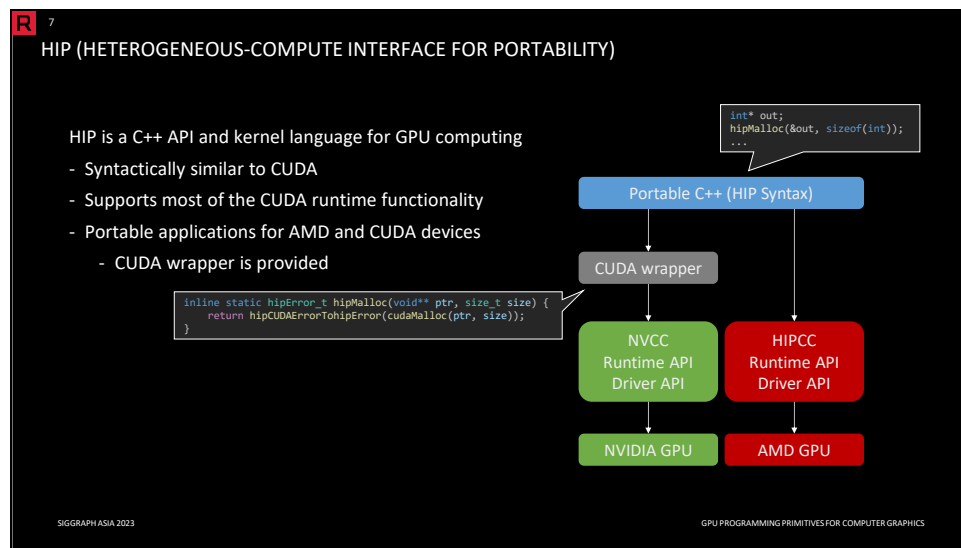


SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Using this link or QR code, you can access the course webpage with course resources, including presentation slides, PDF, and sample code.

Slide 7



CUDA is a widely supported GPU computing environment popular for scientific computations. A drawback is that CUDA is specific for Nvidia GPUs.

HIP is a C++ API and kernel language designed for GPU computing. It offers a syntax that closely resembles CUDA and supports a majority of the CUDA runtime functionality. It enables the development of portable applications for both AMD and CUDA devices. For the Nvidia path, the HIP header is only a wrapper around the CUDA, while for the AMD path, the program is directly compiled into the AMD device-specific code.

Therefore, we decided to use HIP/CUDA as a platform for algorithms presented in this course.

R 8

HIP RUNTIME API – KERNEL EXAMPLE

A counterpart to the CUDA runtime API

- Host calls are prefixed with *hip* instead of *cuda*

Kernel compatibility

- Same built-in variables
 - thread index, block index, and block size
- Functions specifiers such as `__global__` and `__device__`
- Kernel launch via `<<<...>>>` specifying the grid and block resolutions

```

#include <hip/hip_runtime.h>

__device__ int threadIndex()
{
    return threadIdx.x + blockIdx.x * blockDim.x;
}

__global__ void Kernel(int* out)
{
    int index = threadIndex();
    if (index == 0) *out = warpSize;
}


int main()
{
    int* out;
    hipMalloc(&out, sizeof(int));
    Kernel<<<1, 64>>>>(out);
    hipFree(out);
    return 0;
}
            
```

cudaMalloc(&out, sizeof(int));
 Kernel<<<1, 64>>>>(out);
 cudaFree(out);

SIGGRAPH ASIA 2023
GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The HIP runtime API is a counterpart to the CUDA runtime API, with host calls prefixed with ‘*hip**’ instead of ‘*cuda**’. The HIP device code is practically identical to the CUDA device code, providing the same built-in variables such as thread index, block index, or block size. Similarly, kernel functions are decorated with `__global__` and device functions with `__device__`. The kernels functions are launched via `<<<...>>>`, specifying the grid and block resolutions.

Slide 9



9

HIP DRIVER API – KERNEL EXAMPLE

A counterpart to the CUDA driver API

- Host calls are prefixed with *hip* or *hiprtc* instead of *cu* and *nvrtc*
- Kernels compiled in runtime via *hiprtc* (similar to *nvrtc*) and launched via *hipModuleLaunchKernel*

```
const char* code = ...
const char* funcname = ...

hiprtcProgram prog;
hiprtcCreateProgram(
    &prog, code, "", 0, nullptr, nullptr);
hiprtcCompileProgram(prog, 0, nullptr);

size_t binarySize = 0;
hiprtcGetCodeSize(prog, &binarySize);

std::vector<std::byte> binary(binarySize);
hiprtcGetCode(prog, binary.data());

hipModule_t module;
hipModuleLoadData(&mod, cuModuleGetFunction(&func, ...

hipFunction_t func;
hipModuleGetFunction(&func, module, funcname);

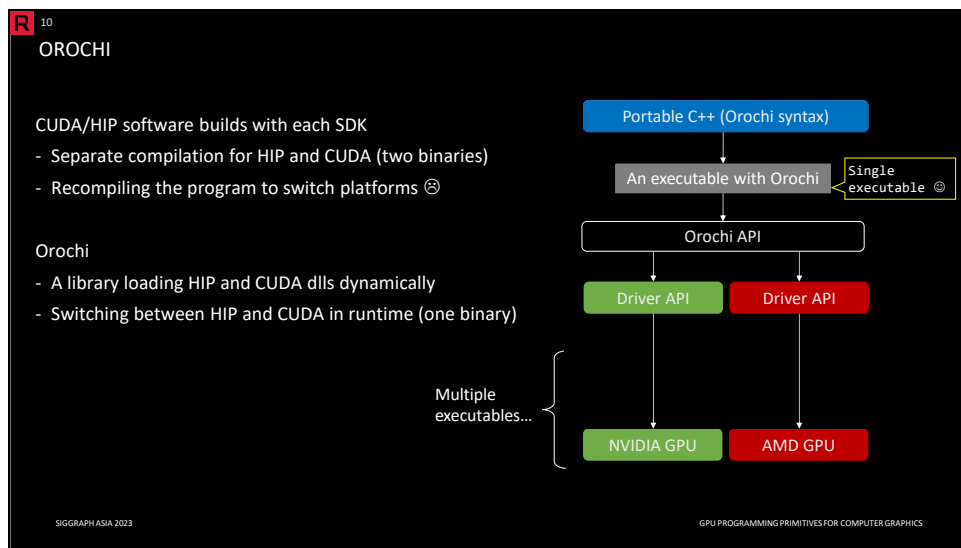
void* args[] = { &out };
hipModuleLaunchKernel(func, 1, 1, 1, 64, 1, 1, 0,
    0, reinterpret_cast<void**>(args), 0);
```

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Similarly, the HIP driver API is a counterpart to the CUDA driver API, with host calls prefixed with ‘*hip**’ or ‘*hiprtc**’ instead of ‘*cu**’ and ‘*nvrtc**’. In the example on the right, we can compile the kernel manually in runtime using the HIPRTC API.

Slide 10




The HIP code can be compiled for both Nvidia and AMD; however, it needs to be compiled for each platform separately. To switch the platforms, we need to recompile the code.

Orochi is a library loading HIP and CUDA APIs dynamically, allowing the user to switch platforms at runtime via a single (host) binary.

We decided to provide code samples written in Orochi for your convenience and simplicity of the sample code structure.

Slide 11

11

OROCHI – KERNEL EXAMPLE

The same as HIP driver API

- Host calls are prefixed with *oro* instead of *hip* or *cu*

```
const char* code = ...
const char* funcname = ...

orortcProgram prog;
orortcCreateProgram(
    &prog, code, "", 0, nullptr, nullptr);
orortcCompileProgram(prog, 0, nullptr);

size_t binarySize = 0;
orortcGetCodeSize(prog, &binarySize);

std::vector<std::byte> binary(binarySize);
orortcGetCode(prog, binary.data());

oroModule module;
oroModuleLoadData(&module, binary.data());

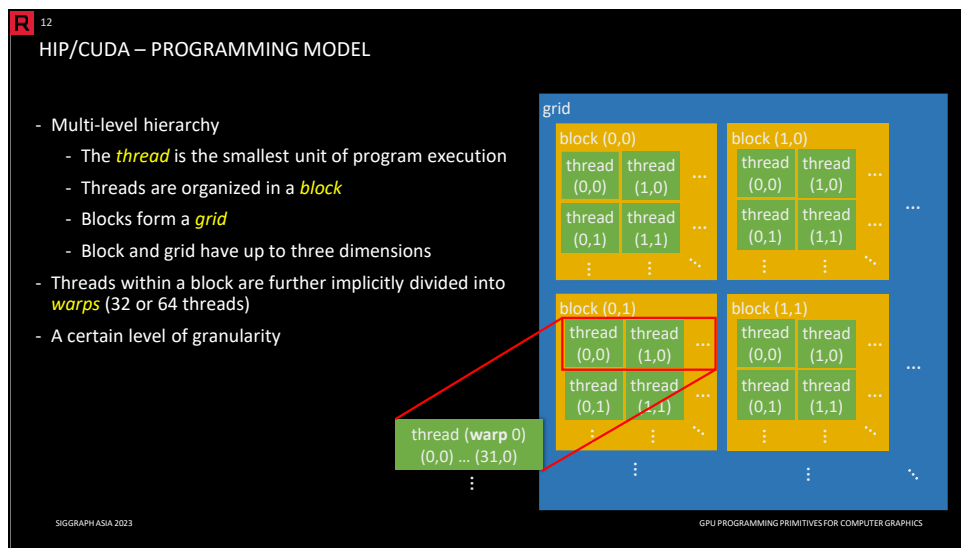
oroFunction func;
oroModuleGetFunction(&func, module, funcname);

void* args[] = { &out };
oroModuleLaunchKernel(func, 1, 1, 1, 64, 1, 1, 0,
    0, reinterpret_cast<void**>(args), 0);
```

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

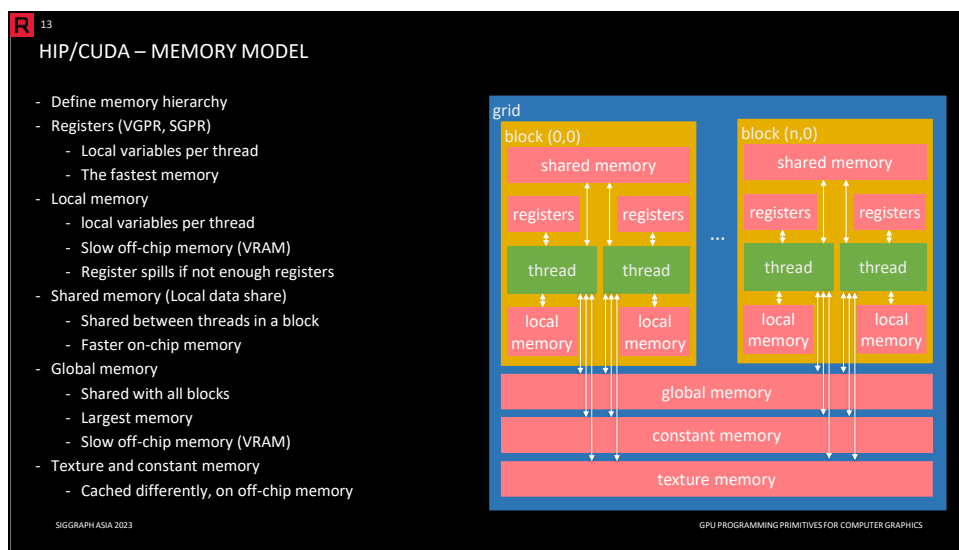
Orochi is practically the same as the HIP driver API; the host calls are prefixed with ‘oro*’ instead of ‘hip*’ / ‘cu*’.



HIP and CUDA use three models. The first model is the *programming model*, which defines how the threads are organized into a multi-level hierarchy. The thread is the smallest unit of parallelism. The threads are further organized into blocks of a fixed size (e.g., 128 or 256 threads), and blocks are organized into a grid. Both the blocks and the grid have up to three dimensions.

Technically speaking, there is one more level between threads and blocks. Threads within a block are implicitly grouped into *warps*, where each *warp* contains 32 or 64 threads. The warp was originally defined by the *execution model* (see the following slides) in the context of scheduling, but since HIP/CUDA introduced the *warp-level primitives* (that we will discuss later as well), we can exploit a priori knowledge of warps for the algorithm design.

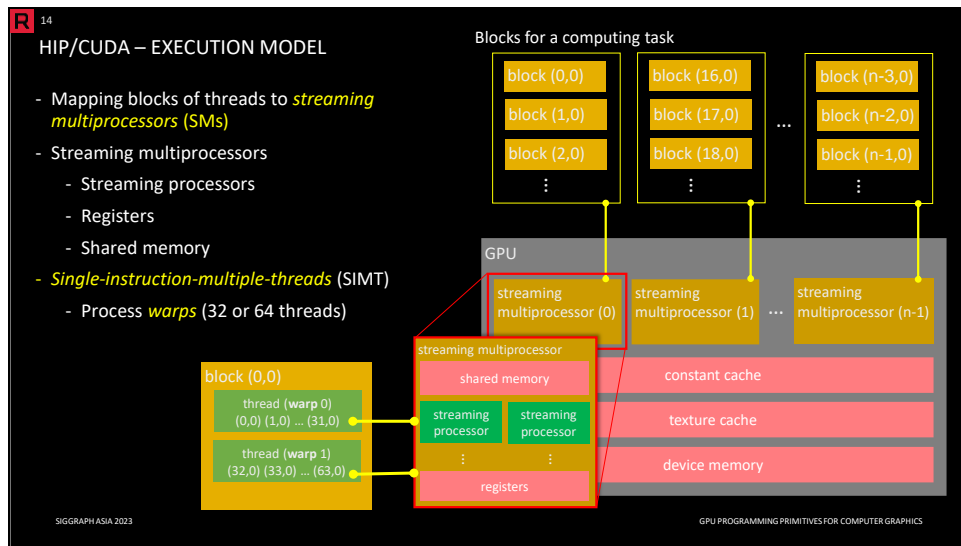
This hierarchical model allows us to choose the appropriate level of granularity when designing parallel algorithms. We are going to explain how to arrange the hierarchy for several algorithms.



The *memory model* defines the memory hierarchy.

- *Registers* (also known as VGPR or SGPR) is a fast per-thread on-chip memory used for storing local variables of the thread.
- *Local memory* stores local variables that do not fit into registers (i.e., register spilling). Local memory is an off-chip memory, and thus it is significantly slower than registers.
- *Shared memory* (also known as *local data share* – LDS) is a fast on-chip memory shared between threads in the block, providing an efficient way of communication between threads in the block. It is slightly slower than registers but significantly faster than off-chip memory.
- *Global memory* is large memory but very slow (taking hundreds of clock cycles per IO operation) off-chip memory, typically stored in VRAM.
- *Constant* and *texture memory* are off-chip types of memory optimized for read-only accesses. Texture memory is suitable for storing image data, exploiting 2D spatial coherency.

Slide 14

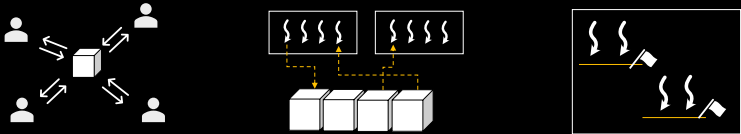


The *execution model* defines how blocks of threads

Streaming multiprocessors consist of streaming processors, registers, and shared memory; all SMs share device memory and caches. The threads are scheduled and executed on a streaming multiprocessor in *warps* (i.e., groups of 32 or 64 threads); this model is also known as *single-instruction-multiple-data (SIMT)*.

HIP KEY COMPONENTS

- Atomic operations
 - Guarantee correctness even in a highly parallel environment
- Shared memory
 - Temporal memory for sharing data in the same block
- Warp-level primitives
 - Low-level control of warp execution
 - Inter-warp communication between threads



The diagrams illustrate the three key components of HIP. The first diagram shows four threads (represented by person icons) interacting with a central shared memory block (represented by a cube). The second diagram shows two warps (represented by boxes with wavy lines) interacting with a shared memory block (represented by a cube). The third diagram shows a single warp (represented by a box with wavy lines) interacting with a shared memory block (represented by a cube).

SIGGRAPH ASIA 2023

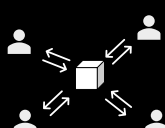
GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The algorithms we cover in this course heavily depend on three essential components provided by HIP/CUDA.

- Atomic operations ensure the correctness of basic arithmetic operations within a highly parallel environment.
- Shared memory acts as temporary storage for exchanging information among threads within the same block.
- Warp-level primitives enable control over warp execution and facilitate communication between threads within a warp.

ATOMIC OPERATIONS

- Even a trivial operation may consist of multiple instructions
- Concurrent execution by multiple threads may lead to undesirable results (race conditions).
 - $X += Y$
- Addition consists of three steps:
 - Reading a value of X
 - Adding Y to this value
 - Writing the result back to X
- Atomic operations (or atomics) guarantee that the operation is correctly processed in parallel execution
 - `atomicAdd()`, `atomicMin()`, `atomicMax()`, `atomicExch()`, `atomicCAS()`, etc.



```

global __void DotProductKernel(int size, float* x,
float* y, float* out)
{
    if (index < size) {
        atomicAdd(out, x[index] * y[index]);
    }
}
            
```

Multiple threads may process simultaneously

SIGGRAPH ASIA 2023
GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Even a trivial operation may consist of multiple instructions. Concurrent execution by multiple threads may lead to undesired results as the instructions of threads are executed in arbitrary order (i.e., race conditions). For example, addition consists of three steps: reading the original value, adding the value to the original value, and writing the new value.

To prevent race conditions, we can employ built-in atomic operations that guarantee that the operation is correctly processed in parallel execution. The HIP provides a couple of atomic operations such as `atomicAdd()`, `atomicMin()`, `atomicMax()`, `atomicExch()`, `atomicCAS()`, and others. Atomic operations allow communication between threads of different blocks.

The code on the right side calculates a dot product of two vectors (represented as two arrays). Each thread multiplies the corresponding vector components fetched from the arrays. To accumulate the sum of partial products, we employ `atomicAdd()` to prevent data races. Note that this is an illustrative example showing the capability of the operations. We will introduce later more efficient algorithms.

17
SHARED MEMORY

Memory shared among the threads in the same **block**

- Can be used for accumulators, communicating between threads
- **`__syncthreads()`** guarantees IO operations have been completed in the **block** by synchronizing threads
 - Some threads in a block may go forward than others

write data

`__syncthreads()`

read data

A block

SIGGRAPH ASIA 2023

```
constexpr int BLOCK_SIZE = 64;

__global__ void DotProductKernel(int size, float* x,
float* y, float* out)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    __shared__ float smem[BLOCK_SIZE];

    float val = 0.0f;
    if (index < size)
        val = x[index] * y[index];

    smem[threadIdx.x] = val;
    __syncthreads();

    if (threadIdx.x == 0)
    {
        float sum = 0.0f;
        for (int i = 0; i < blockDim.x; ++i)
            sum += smem[i];
        atomicAdd(out, sum)
    }
}
```

Shared memory

Write & sync

Read data

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS


The shared memory is a memory that is shared among threads in the same block, declared with `__shared__`. It is very fast, almost as fast as registers, if there are no bank conflicts (we will talk about them later). We typically use shared memory in combination with `__syncthreads`, which allows threads in the block to be synchronized (also known as a barrier). This is important, especially for memory accesses, to make sure that the data has been written before any further computation.

On the right side, we use shared memory to compute the dot product. Each thread stores the corresponding partial products in the shared memory. We use `__syncthreads` to make sure that all data have been written. The first thread then sums the individual products sequentially. This is not the optimal way, and we will show later how to do it more efficiently.

WARP-LEVEL PRIMITIVES

Efficient operations within a warp

- `__shfl*()`: Allows threads to read local registers of another thread in the same warp (no need for shared memory!)
- `__ballot()`: Binary voting within the warp
- `__any()` and `__all()`: Logic quantifiers for the warp



`x`

`= __ballot(x)`
`= 0x95`

The voting results are packed as a bitmask and returned to threads.

```
__global__ void DotProductKernel (int size, float* x,
float* y, float* out)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int laneIndex = threadIdx.x & (warpSize - 1);

    float val = 0.0f;
    if (index < size)
        val = x[index];

    float sum = 0.0f;
    for (int i = 0; i < warpSize; ++i)
        sum += __shfl(val, i);
    if (laneIndex == 0)
        atomicAdd(out, sum);
}
```

Still sequential work ☹️

Read data from a specific thread

⇒ Need more parallelism for further optimization

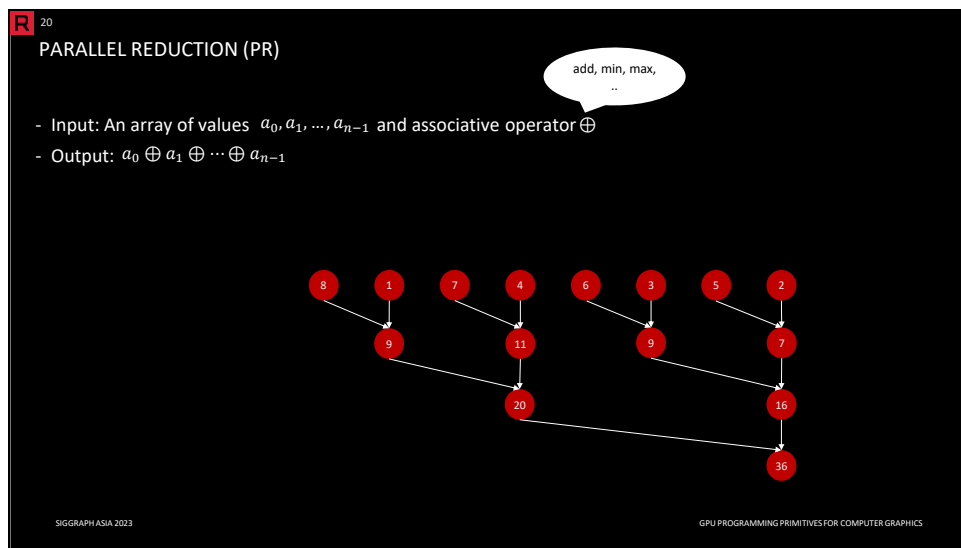
SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

HIP/CUDA provides *warp-wide level primitives* that facilitate efficient communication within a warp. The *shuffle* instruction allows reading registers of other threads in the warp without the need for shared memory. The *ballot* instruction returns an integer, where each bit indicates a predicate of each thread. The *any* and *all* instructions implement logic quantifiers. We will employ these instructions a lot in the following section.

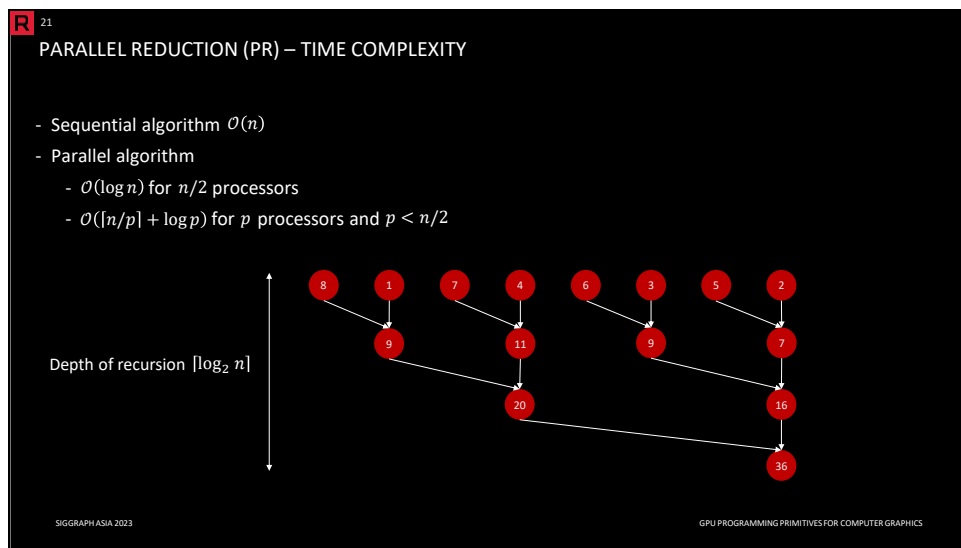
Note that in HIP, the warp-level primitives implicitly use the synchronized versions with a full mask to handle independent thread scheduling.



We look into the details of two pivotal algorithms: parallel reduction and parallel prefix scan. These two algorithms are typically covered in parallel programming courses. As both algorithms serve as building blocks for more advanced techniques, we pay extra attention to them.



The reduce operation takes a binary associative operator op and an ordered sequence of n elements, returning a value obtained by iteratively applying operator op on all elements in the sequence.



The sequential algorithm of the reduction is straightforward. We scan the sequence element by element and simultaneously update the partial result of processed elements. The time complexity of the sequential algorithm is $O(n)$.

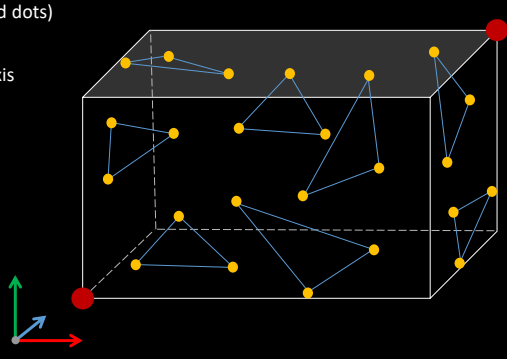
The parallel algorithm is based on the divide and conquer paradigm. The sequence is recursively divided into halves until a single element is left. In interior nodes of a recursion tree, partial results are merged using the operator.

The depth of the recursion tree is $\lceil \log_2(n) \rceil$, and hence the time complexity for $n/2$ processors is $O(\log n)$. The time complexity for p processors such that $p < n/2$ is $O(\lceil n/p \rceil + \log p)$. Each processor requires $\lceil n/p \rceil$ time steps, and merging partial results of each processor takes another $\log_2(p)$ steps.

22
PARALLEL REDUCTION (PR) – AXIS-ALIGNED BOUNDING BOX

Axis-aligned bounding box (AABB)

- Defined by minimum and maximum (red dots)
- Can be computed by parallel reduction
 - Minimum and maximum for each axis
 - Six parallel reductions in 3D



The diagram illustrates an Axis-Aligned Bounding Box (AABB) for a triangle soup. A 3D wireframe box is shown, with its minimum and maximum corner points marked by red dots. Inside the box, several triangles are represented by blue lines connecting yellow dots (vertices). A 3D coordinate system is shown at the bottom left, with red, green, and blue axes.

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Typically, the operator is addition, minimum, or maximum. In the context of computer graphics, it can be, for example, an axis-aligned bounding box of a triangle soup. The bounding box is defined by two points: minimum and maximum (the red dots). Hence, the computation can be decomposed into $2 \times 3 = 6$ parallel reductions in 3D.

PARALLEL REDUCTION (PR) – IMPLEMENTATION

Block-wise reduction with shared memory

- Shared memory for intermediate computations

```

template <typename T>
__device__ T ReduceSumBlock(T val, T* smem)
{
    smem[threadIdx.x] = val;
    __syncthreads();

    for (int i = 1; i < blockDim.x; i *= 2)
    {
        if (threadIdx.x < (threadIdx.x ^ i))
            smem[threadIdx.x] += smem[threadIdx.x ^ i];
        __syncthreads();
    }

    return smem[0];
}
  
```

Shared Memory

Make pairs

Any associative operator

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Parallel reduction (PR) can be implemented using one parallel loop. This is a block-wise variant using shared memory with block-wise barriers. The figure illustrates the steps of the algorithm. There are several ways to do this, but we present the *xor* approach.

The thread pairs in each iteration are determined by *xor* operation: each thread applies *xor* to its own index and i to determine the other thread's index. The distance between paired items is given by i , and it gets doubled after each iteration. For example, for the leftmost item [0], it forms a pair [0,1] in the first iteration; it forms a pair with [0,3] in the second iteration; and finally, it forms a pair [0,7] in the last iteration. The same rule applies to other elements. Only the smaller thread in the pair process to avoid shared memory data race.

Note that any associative operator can be used, although this example uses the plus operator for the sake of simplicity.

PARALLEL REDUCTION (PR) – IMPLEMENTATION

Warp-wise reduction with shuffle

- Directly reading registers of other threads

```

template <typename T>
__device__ T ReduceSumWarp(T val)
{
    for (int i = 1; i < warpSize; i *= 2)
    {
        val += __shfl_xor(val, i);
    }
    return val;
}
  
```

Diagram illustrating the warp-wise reduction process across three iterations (i = 001, i = 010, i = 100) for a warp of 8 threads (lanes 0-7). The diagram shows the state of the warp at each iteration, with arrows indicating the shuffle operation performed by each thread.

Iteration i = 001:

Lane	0	1	2	3	4	5	6	7
Initial	000	001	010	011	100	101	110	111
After Shuffle	[0-1]	[0-1]	[2-3]	[2-3]	[4-5]	[4-5]	[6-7]	[6-7]

Iteration i = 010:

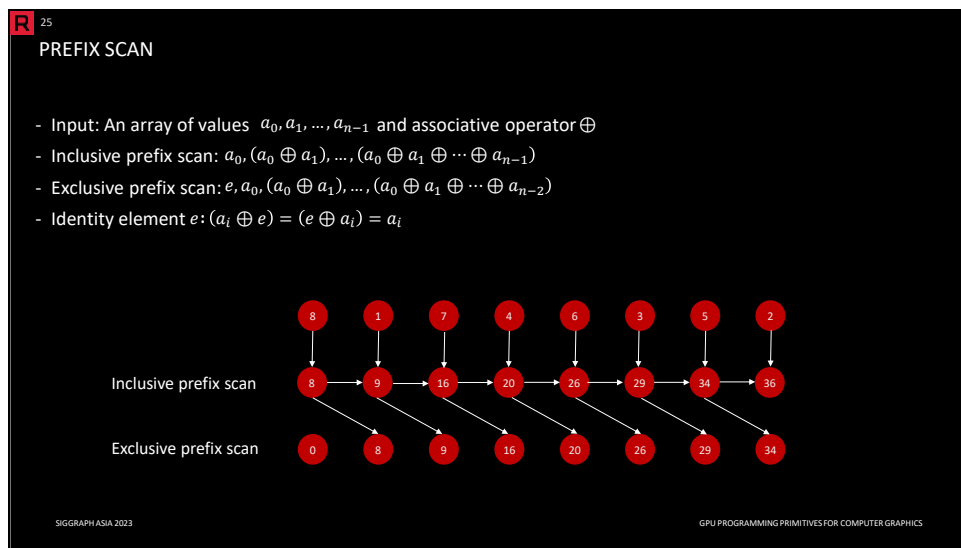
Lane	0	1	2	3	4	5	6	7
Initial	000	001	010	011	100	101	110	111
After Shuffle	[0-3]	[0-3]	[0-3]	[0-3]	[4-7]	[4-7]	[4-7]	[4-7]

Iteration i = 100:

Lane	0	1	2	3	4	5	6	7
Initial	000	001	010	011	100	101	110	111
After Shuffle	[0-7]	[0-7]	[0-7]	[0-7]	[0-7]	[0-7]	[0-7]	[0-7]

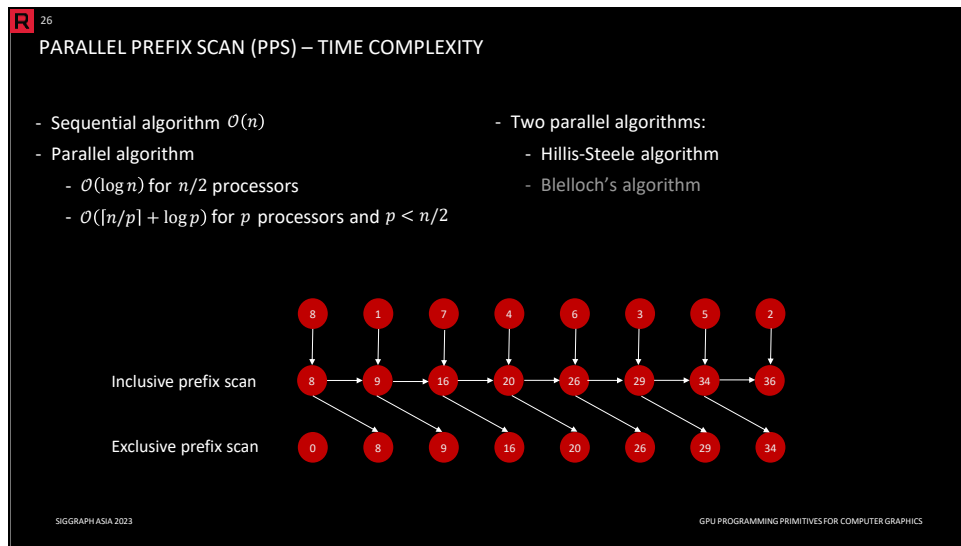
SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The warp-wise variant is practically the same as the block-wise one. The difference is that the values are directly acquired from registers of other threads via the shuffle instruction.

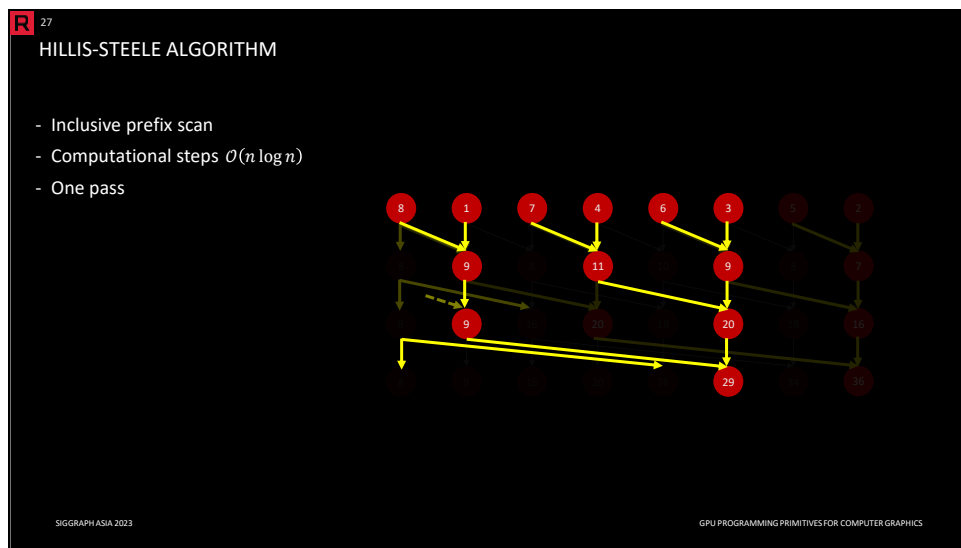


There are two types of prefix scans: *inclusive* and *exclusive*. Both types take a binary associative operator op and an ordered sequence of n elements. The prefix scan returns an ordered sequence, where the i -th element is a reduction of the input sequence up to the i -element, which is either *included* or *excluded*. Note that in practice, we typically use exclusive prefix scan.

The exclusive prefix scan can be constructed from the inclusive prefix scan by removing the last element and inserting the identity element at the beginning. The inclusive prefix scan can be constructed from the exclusive prefix scan by removing the identity element and inserting the sum of the last element of the input sequence and the last element of the exclusive prefix scan at the end. The sequential algorithm is trivial; we scan the input sequence element by element and simultaneously write the partial results of the output sequence.



The time complexity of the sequential algorithm is $O(n)$. The time complexity of the parallel algorithm is the same as parallel reduction. There are two parallel prefix scan algorithms: the Hillis-Steele algorithm and Blelloch's algorithm. We explain the Hillis-Steele algorithm in detail. You can find details of the Blelloch's algorithm in the supplementary material.



The Hillis-Steele algorithm implicitly computes inclusive prefix scan in a single pass with $O(n \log n)$ computational steps. The algorithm works iteratively by adding preceding values to the succeeding ones. In each iteration, a thread adds a value located to the left by a given *offset* to its own value (if such a value exists). The *offset* is initially set to 1 and doubled after each iteration. Intuitively, the algorithm reduces the preceding elements for each entry in the output prefix scan. In the example above, we can reconstruct the computational tree that represents these reductions. Missing branches are assumed to be zero (or an identity element in general).

HILLIS-STEELE ALGORITHM – IMPLEMENTATION

Block-wise prefix scan with **shared memory**

- Shared memory for intermediate computations

```

template <typename T>
__device__ T ScanBlock_HillisSteele(T val, T* smem)
{
    smem[threadIdx.x] = val;
    __syncthreads();

    for (int offset = 1; offset < blockDim.x; offset *= 2)
    {
        if (threadIdx.x - offset >= 0)
            val += smem[threadIdx.x - offset];

        __syncthreads();
        smem[threadIdx.x] = val;
        __syncthreads();
    }

    return smem[threadIdx.x];
}

```

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The Hillis-Steele algorithm can be implemented in one parallel loop. This is a block-wise variant using shared memory. The input values are initially loaded to the shared memory. Each thread keeps the current value of the prefix scan also in registers. In each iteration, the thread loads a value of another thread (given by the current *offset*) from shared memory and adds it to its own value stored in registers. Before we write the updated value back to shared memory, we use a block-wise barrier to prevent race conditions. The offset gets doubled after each iteration. On the right side, you can see a figure illustrating the offset behavior in different iterations.

HILLIS-STEELE ALGORITHM – IMPLEMENTATION

Warp-wise prefix scan with **shuffle**

- Directly reading registers of other threads

```

template <typename T>
__device__ T ScanWarp_HillisSteele (T val)
{
    int laneIndex = threadIdx.x & (warpSize - 1);
    for (int offset = 1; offset < warpSize; offset *= 2)
    {
        T paired = __shfl_up(val, offset);
        if (laneIndex - offset >= 0)
            val += paired;
    }
    return val;
    }
  
```

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The warp-wise variant is practically the same as the block-wise one. The difference is that the values are directly acquired from registers of other threads via the shuffle instruction. Note that the `__shfl_up` instruction reads the variable of a thread with the lane index lower (given by the offset) than the caller's lane index.

WARP-WISE BINARY PPS – IMPLEMENTATION

- A special case of prefix scan with binary values (0 or 1)
 - `__ballot`: bits indicating how threads voted
 - `__popc`: the number of bits set to one

```

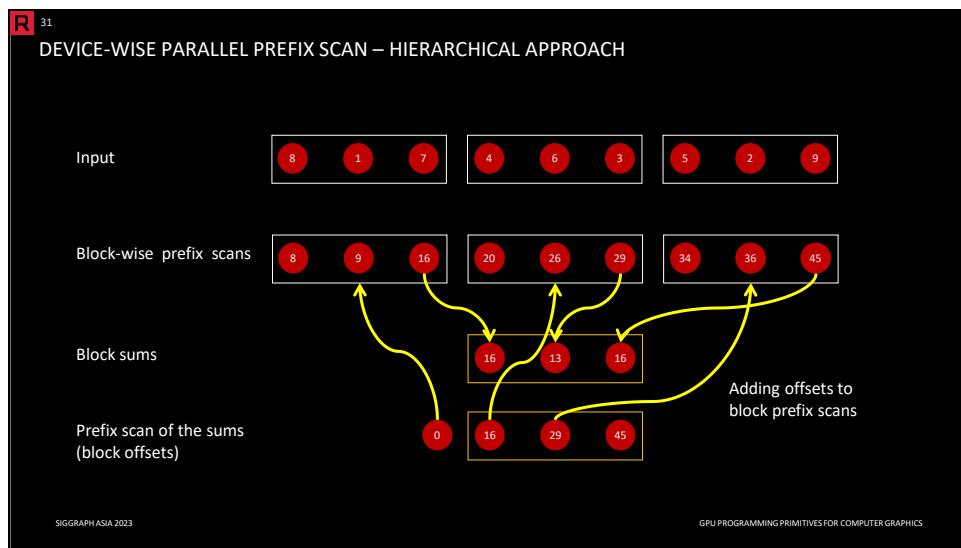
device__ int ScanWarpBinary(bool x)
{
    int laneIndex = threadIdx.x & (warpSize - 1);
    uint64_t ballot = __ballot(x);
    return __popc11(ballot & ((1ull << laneIndex) - 1));
}
  
```

`laneIndex=0 -> 00000000`
`laneIndex=1 -> 00000001`
`laneIndex=2 -> 00000011`
`laneIndex=3 -> 00000111`
`laneIndex=4 -> 00001111`
 ...

`ballot(x) = 1 0 0 1 0 1 0 1`
 Threads:
 [0] 0 = **POPC** (~~1~~ 0 0 1 0 1 0 1)
 [1] 1 = **POPC** (~~1~~ 0 0 1 0 1 0 1)
 [2] 1 = **POPC** (~~1~~ 0 0 1 0 1 0 1)
 [3] 2 = **POPC** (~~1~~ 0 0 1 0 1 0 1)
 [4] 2 = **POPC** (~~1~~ 0 0 1 0 1 0 1)
 ...

SIGGRAPH ASIA 2023
 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

If the values of the prefix scan are binary (0 or 1), we can implement warp-wise prefix scan efficiently via the *ballot* and *popcount* instructions. The *ballot* instruction returns an integer where each bit represents the vote (0 or 1) of the corresponding thread in the warp. The *popcount* instruction calculates the number of bits that are set to one. To obtain the prefix scan value for a specific thread, we need to first mask out the higher bits, corresponding to threads with higher lane indices, before we use the popcount instruction.

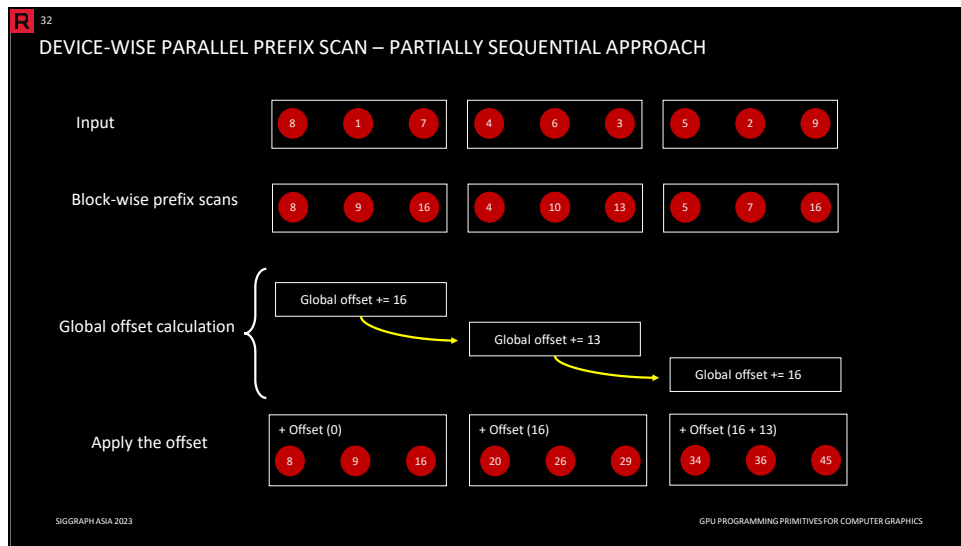


We introduced block-wise and warp-wise prefix scan algorithms. However, the input is typically larger than the block or warp. There are two approaches to how we can compute the (global) device-wise prefix scan from (partial) block-wise prefix scans.

The problem is that for each block, we need to compute its *offset*, which is the sum of all previous items. The key observation is that already each block has computed the sum of its items, i.e., the last entry in the block-wise prefix scan.

One solution is a hierarchical approach. First, we compute the prefix scan in each block, then we compute the prefix scan of block sums to get the offset for each block. If the number of blocks is larger than the block size, we have to use more than one level, requiring multiple kernel launches and additional buffers.

Slide 32



Another solution is to use a partially sequential approach. As before, we compute block-wise prefix scans for each block. To compute the global offset for each block, we process blocks sequentially. Each block is waiting until the previous block updates the global offset. This approach can be implemented in a single kernel launch.

33

DEVICE-WISE PARALLEL PREFIX SCAN – IMPLEMENTATION

- Hierarchical approach
 - Prefix scan of block sums
 - Multiple kernel launches
 - For large inputs we need more than two levels
- Sequential approach
 - Wait for the block offset using atomic counter
 - Add the block sum obtaining the block offset
 - Add the block offset and block values to obtain the global prefix scan
 - **Only one kernel launch ☺**
- Both approaches can be implemented as an in-place algorithm

Device-wise prefix scan

```

template <typename T>
__device__ T ScanDevice (T val, T* smem, T* sum, int*
counter)
{
    val = ScanBlock(val, smem);
    __shared__ T offset;
    if (threadIdx.x == blockDim.x - 1)
    {
        while (atomicAdd(counter, 0) < blockIdx.x);
        __threadfence();

        offset = *sum;
        *sum += val;

        __threadfence();
        atomicAdd(counter, 1);
    }
    __syncthreads();
    return offset + val;
}

```

Parallel execution

Synchronization for sequential execution

Parallel execution

SIGGRAPH ASIA 2023
GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Unlike parallel reduction, where we can use an atomic operation for each block to get the final result, the implementation of device-wise parallel prefix scan is more complicated. The hierarchical approach requires multiple kernel launches, and even a one-level hierarchy might not be enough.

The sequential approach can be implemented via a *waterfall scheme*, which we will discuss later in detail. It allows us to compute the device-wise prefix scan in a single kernel launch regardless of the input size. We use two atomic counters: *sum* and *counter*. The last thread in each block spins until *counter* is equal to its block index, then it adds its sum to *sum*, obtaining the offset for all threads in the corresponding block. Finally, it atomically increases *counter*, letting the next block be processed.

Note that as we do not have to store intermediate prefix sum results in global memory, both approaches can be implemented as in-place algorithms, avoiding unnecessary memory allocations.

Slide 34



This section presents a collection of more advanced techniques widely applicable across different areas.

PARALLEL ENQUEUEING

- Writing an output is one of the most common tasks in parallel computing
 - Task queue, tree constructions, etc.
 - Non-trivial if not all threads want to write
- Naïve solution is to use atomic add to get the offset
- Better solution is to use warp-wise with atomic add
- Device-wise prefix scan is not necessary
 - Block-wise or warp-wise are sufficient

Naïve solution with atomic add

```

global __void EnqueueNaiveKernel(const int* input,
int* output, int* counter)
{
    int value = ...

    bool enqueue = /* ANY
if (enqueue)
    output[atomicAdd(counter, 1)] = value;
    
```

Per thread @

Input buffer

Output buffer

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Writing output is a basic task in parallel computing, frequently encountered in various domains. In computer graphics, spatial data are often organized into hierarchical structures. During the construction, we output elementary blocks such as nodes or cells in parallel. Another example is a task queue, where we need to enqueue new tasks. While a naive approach using atomic add to determine the offset (as shown on the right side) is simple, this method can introduce significant overhead. A more efficient alternative is to employ the warp-wise (or block-wise) prefix scan with atomic add, which offers improved performance compared to a device-wise prefix scan that may be unnecessarily wasteful.

PARALLEL ENQUEUEING – IMPLEMENTATION

Binary warp-wise prefix scan with atomic add and shuffle

```

__global__ void EnqueueBinaryKernel(int size, const
int* input, int* output, int* counter)
{
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int laneIndex = threadIdx.x & (warpSize - 1);

    int val = 0;
    if (index < size) val = input[index];

    bool enqueue = /* ANY CONDITION HERE */;
    int warpScan = ScanWarpBinary(enqueue);

    int warpOffset = 0;
    if (laneIndex == warpSize - 1)
        warpOffset = atomicAdd(
            counter, warpScan + enqueue);
    warpOffset = __shfl(warpOffset, warpSize - 1);

    if (index < size && enqueue)
        output[warpOffset + warpScan] = val;
}

```

Include the current value of the last lane

Warp-wise prefix scan with atomic add and shuffle

```

__global__ void EnqueueKernel(int size, const int*
input, int* output)
{
    int val0 = input[threadIdx.x];
    int val1 = input[threadIdx.x + 1];

    bool enqueue0 = /* ANY CONDITION HERE */;
    bool enqueue1 = /* ANY CONDITION HERE */;
    int enqueuedCount = enqueue0 + enqueue1;

    int warpScan =
        ScanWarp(enqueuedCount) - enqueuedCount;

    int warpOffset = 0;
    if (laneIndex == warpSize - 1) warpOffset =
        atomicAdd(counter, warpScan + enqueuedCount);
    warpOffset = __shfl(warpOffset, warpSize - 1);

    int offset = warpOffset + warpScan;
    if (index0 < size && enqueue0)
        output[offset++] = val0;
    if (index1 < size && enqueue1)
        output[offset] = val1;
}

```

Each threads outputs up to two items

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

On the left side, you can see an implementation employing the binary warp-wise prefix scan. On the right side, each thread outputs up to two items. In this case, we use the (general) warp-wise prefix scan that can handle arbitrary values.

PARALLEL ENQUEUEING – COMPLEMENT

- Sometimes we want to output data to either one of buffer or to another one
- We can use just one prefix scan and its complement instead of two prefix scans

$$\sum_{j=0}^i (1 - a_j) = i - \sum_{j=0}^i a_j$$

i-th element

Binary warp-wise prefix scan with its complement

```

_global__ void EnqueueComplementKernel(int size, const int*
input, int* output0, int* output1, int* counters)
{
    int index = threadIdx.x + blockDim.x * blockDim.x;
    int laneIndex = threadIdx.x & (warpSize - 1);

    int val = 0;
    if (index < size) val = input[index];
    int complWarpScan = ScanWarpBinary(!enqueue);
    bool enqueue = /* ANY CONDITION HERE */;
    int warpScan = ScanWarpBinary(enqueue);
    int complWarpScan = laneIndex - warpScan;

    int warpOffset = /* THE SAME AS BEFORE */;

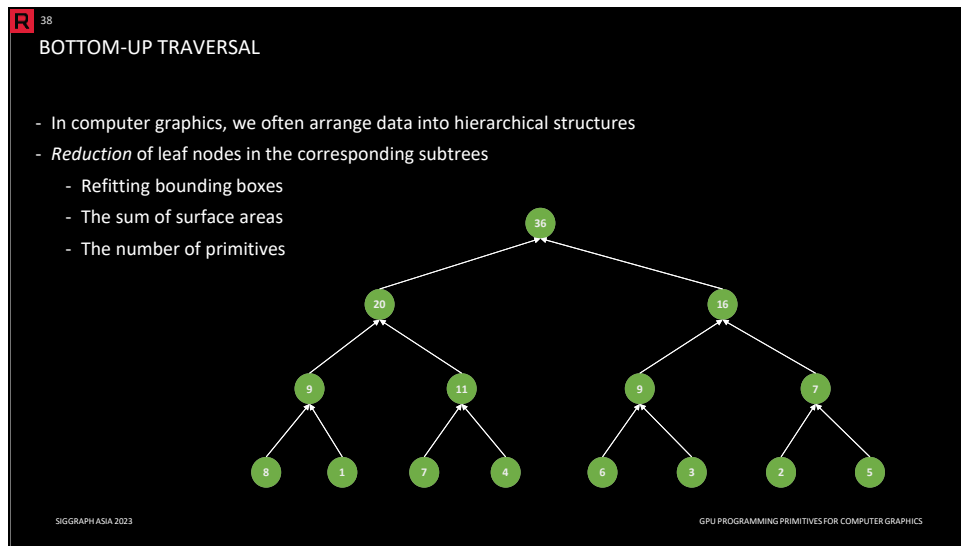
    int complWarpOffset = 0;
    if (laneIndex == warpSize - 1) complWarpOffset =
        atomicAdd(&counters[1], complWarpScan + !enqueue);
    complWarpOffset = __shfl(complWarpOffset, warpSize - 1);

    if (index < size)
    {
        if (enqueue) output0[warpOffset + warpScan] = val;
        else output1[complWarpOffset + complWarpScan] = val;
    }
}

```

SIGGRAPH ASIA 2023
GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

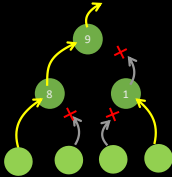
Sometimes we want to output data either to one buffer or to another. On the left side, our goal is to separate green and red elements into two output buffers. We can use one prefix scan and its complement instead of two separate prefix scans. On the right side, you can see the implementation of a simple example, separating input numbers. In practice, we can employ this approach, for example, in the context of a hierarchical structure construction where we produce either leaf nodes or internal nodes.



In computer graphics, we often arrange data into hierarchical structures. A common operation is a *reduction* of leaf nodes in the corresponding subtrees for each internal node, e.g., bounding boxes or the sum of surface areas. This is very similar to parallel reduction; however, in this case, the tree structure is explicit.

39 **BOTTOM-UP TRAVERSAL - IMPLEMENTATION**

- **Counters** in interior nodes (initialized to 0) and **parent links**
- Each thread is assigned to a leaf node proceeding up to the root
- In internal node, the thread atomically increment the counter
- Allowing only the last thread processes the internal node



Summing up values in all the leaf nodes

```

__global__ void BottomUpTraversalKernel(int size, const Node* nodes,
const Leaf* leaves, int* sums, int* counters)
{
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    if (index >= size) return;

    const Leaf& leaf = leaves[index];
    index = leaf.m_parentAddr;

    while (index >= 0 && atomicAdd(&counters[index], 1) > 0)
    {
        __threadfence();

        const Node& node = nodes[index];

        int sum = 0;
        if (node.leftIsLeaf())
            sum += leaves[node.getLeftAddr()].m_value;
        else
            sum += sums[node.getLeftAddr()];

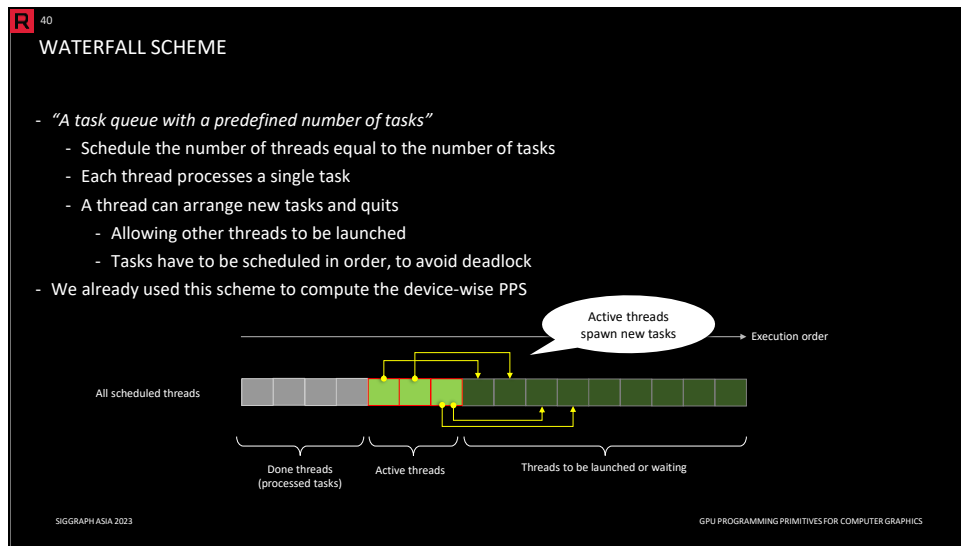
        if (node.rightIsLeaf())
            sum += leaves[node.getRightAddr()].m_value;
        else
            sum += sums[node.getRightAddr()];

        sums[index] = sum;
        index = node.m_parentAddr;
        __threadfence();
    }
}

```

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

To implement this, we need counters in interior nodes (initialized to 0) and parent links. Each thread is assigned to a leaf node proceeding up to the root. In an internal node, the thread atomically increments the counter. Only the last thread processes the internal node (the second thread in the case of binary trees). In the example on the right side, we are summing up values in all leaf nodes. As we modify *sums*, we have to use a memory fence to make sure that the changes are visible to threads in other blocks.



A *waterfall scheme* is a task queue with a predefined number of tasks. We schedule the number of threads equal to the number of tasks. Each thread processes a single task; the thread spawns new tasks and quits, allowing other threads to be launched. Note that tasks have to be scheduled in order to avoid deadlock.

WATERFALL SCHEME - DEVICE-WISE PARALLEL PREFIX SCAN

- Compute the device-wise PPS from block-wise PPSs
- Waterfall scheme for the global offset calculation
- A thread in a block adds its sum the global offset obtaining the offset for its elements
- It increments the block counter letting the next block compute its offset

Block-wise prefix scans (parallel)

Global Offset (serial)

Global offset += 16

Global offset += 13

Global offset += 16

Apply the offset (parallel)

+ Offset (0)

+ Offset (16)

+ Offset (29)

Device-wise prefix scan

```
template <typename T>
__device__ T ScanDevice(T val, T* smem, T* sum, int* counter)
{
    val = ScanBlock(val, smem);
    __shared__ T offset;
    if (threadIdx.x == blockDim.x - 1)
    {
        while (atomicAdd(counter, 0) < blockIdx.x);
        __threadfence();
        offset = *sum;
        *sum += val;
        __threadfence();
        atomicAdd(counter, 1);
    }
    __syncthreads();
    return offset + val;
}
```

Waiting for the previous block to finish the task

Global offset calculation

Schedule the next block

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

We already used the waterfall scheme to compute device-wise prefix scan, which you can see on the right side.

To compute the device-wise Parallel Prefix Scan (PPS) from block-wise PPSs, only a in each block participates in the computation. Since there is only one active task at a time, the task queue can be represented by a single counter. The thread starts its work when the counter value is equal to its block index. The thread reads the offset and adds the block sum to the offset. After adding the sum, the thread atomically increments the counter, signaling that the next block can be processed.

WATERFALL SCHEME – TOP-DOWN TREE BUILD

- The top-down construction of a binary tree via the waterfall scheme
 - Each thread processes a single node
 - Each thread waits until its node is ready to process
 - A node marks children as *ready* after it is processed

```

global __void BuildTree(int nodeCount, bool* readyStates, Node* nodes)
{
    int index = threadIdx.x + blockDim.x * blockIdx.x;

    bool done = index >= nodeCount;
    while (!__all(done))
    {
        __threadfence();

        bool ready = done ? false : readyStates[index];
        if (!ready) continue;

        Node& node = nodes[index];
        if (node.isLeaf())
        {
            node = build a leaf
        }
        else
        {
            nodes[node.left] = build left child
            nodes[node.right] = build right child
            __threadfence();
            readyStates[node.left] = true;
            readyStates[node.right] = true;
        }
        done = true;
    }
}
  
```

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Besides the bottom-up traversal of the hierarchical structure, we sometimes need the top-down traversal that can be implemented via the waterfall scheme. In the example above, we use this approach to build a binary tree in a top-down fashion.

The thread index is the same as the processed node index. The task queue contains only binary values, indicating whether the task is ready or not yet. Other input information about a particular node is stored in the node structure itself. In the beginning, there is only one task corresponding to the root node. Each task may produce two new tasks.

PERSISTENT THREADS

- An algorithm can be implemented as several kernels
- Separated kernel launches are implicitly globally synchronized (a.k.a. global barrier) in a stream
 - Easy to guarantee that all the previous tasks have been done
- It might be beneficial to fuse multiple kernels into one kernel
 - Eliminate the kernel launch overhead
 - Reduce memory accesses
- **Fused kernel may have deadlock**
 - Not guaranteed that all threads are running simultaneously

Separated Kernels

A Fused Kernel - A deadlock case ☹️

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Separate kernel launches are implicitly globally synchronized (a.k.a. global barrier) in a stream. It might be beneficial to fuse multiple kernels into one kernel to decrease the kernel launch overhead and reduce memory access. Global synchronization in a single kernel typically leads to a deadlock, as it is not guaranteed that all threads are running simultaneously.

PERSISTENT THREADS

- Launch the maximum number of threads that can run simultaneously on the device to prevent deadlock
 - The threads are persistently running
- We can use the occupancy API to determine the number of persistent threads
- The global barrier by spin waiting is safely used for already processed tasks

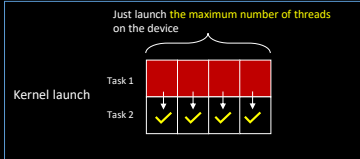
```

oroDeviceProp prop;
oroGetDeviceProperties(&prop, device);

int blockCount;
oroOccupancyMaxActiveBlocksPerMultiprocessor(&blockCount, func, BLOCK_SIZE, 0);

int nPersistentThreads = prop.multiProcessorCount * BLOCK_SIZE * blockCount;
            
```

Fused kernel – Persistent threads



```

- Task 1
if (laneIndex == 0)
{
    atomicAdd(counter, 1);
    while (atomicAdd(counter, 0) < numberOfTask1);
}
__syncthreads();
__threadfence();

- Task 2
            
```

Count finished warps and can wait properly

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The solution is to launch the maximum number of threads that can run simultaneously on the device. As there are no inactive tasks that we are waiting for anymore, it is possible to use active spinning. This concept is known as persistent threads.

Determining the number of persistent threads is relatively difficult as it depends on a particular HW architecture and occupancy. Luckily, we can use the occupancy API to query the number of active blocks on a multiprocessor. The number of blocks is further multiplied by the warp size and the number of multiprocessors (which can also be queried) to get the number of persistent threads.

PARALLEL POOL ALLOCATOR

- In some situations, we may need a per-thread/per-block buffer
 - Registers and Shared memory may not be large enough
 - Stack memory, hash table, etc.
- Allocating a global buffer for all scheduled threads may be wasteful as only a fraction of threads are active at the time ☹️
- Persistent threads can reduce scheduled threads, but it might be difficult to change scheduling in some situations
- `malloc()` in a kernel might be too costly ☹️

Buffer for each thread

➔ **Allocate a buffer only for the active threads and assign it dynamically to active threads**

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

In some situations, we need a per-thread buffer (e.g., stack). Local arrays are allocated per thread, but they are hard to control as they may cause significant register pressure. Allocating a global buffer for all scheduled threads may be wasteful as only a fraction of threads are active at a time.

Persistent threads can reduce scheduled threads, but it might be difficult to change scheduling in some situations. Another option is to use dynamic allocation in the device code via *malloc*; however, depending on the implementation, it might be too costly.

Instead, we can allocate a buffer only for the active threads and assign it dynamically to active threads.

PARALLEL POOL ALLOCATOR

Acquire a lock based on a hash of the warp index

Fixed-size buffers

Hash (warp index)

Other warps fail to acquire

Try the next one

Linearly search for a free buffer

Warp

Parallel pool allocator (warp level)

```

int warpIndex = ...
int laneIndex = ...
int indexOfBuffer = INVALID_INDEX;

int iterator = hash(warpIndex) % numberOfBuffers;
while (bufferIndex == INVALID_INDEX)
{
    if (laneIndex == 0)
    {
        if (atomicCAS(&locks[iterator], 0, 1) == 0)
            bufferIndex = iterator; // success!
        iterator = (iterator + 1) % numberOfBuffers;
        bufferIndex = __shfl(bufferIndex, 0);
    }
    __threadfence();
}

int* buffer = getBufferPointer(bufferIndex);

... Do some awesome work here with the buffer ...

__threadfence();
__syncwarp();
if (laneIndex == 0)
    atomicExch(&locks[bufferIndex], 0);

```

Acquire a lock

Release the lock

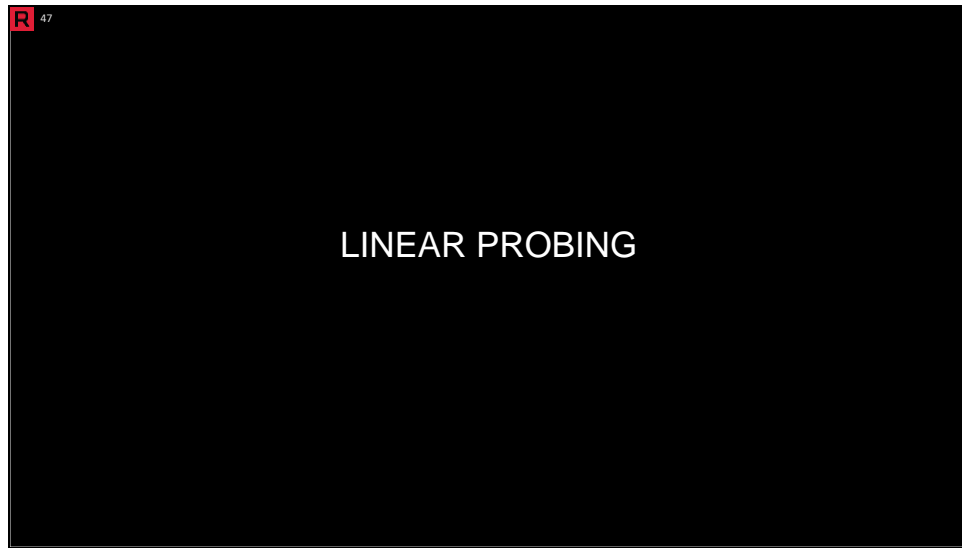
SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

We first allocate a pre-defined number of buffers, enough for the active threads. The number of buffers should be roughly the same as the number of persistent threads. It is independent of the total number of scheduled warps (no deadlock); however, too few buffers would make threads idle, and too many buffers would be wasteful.

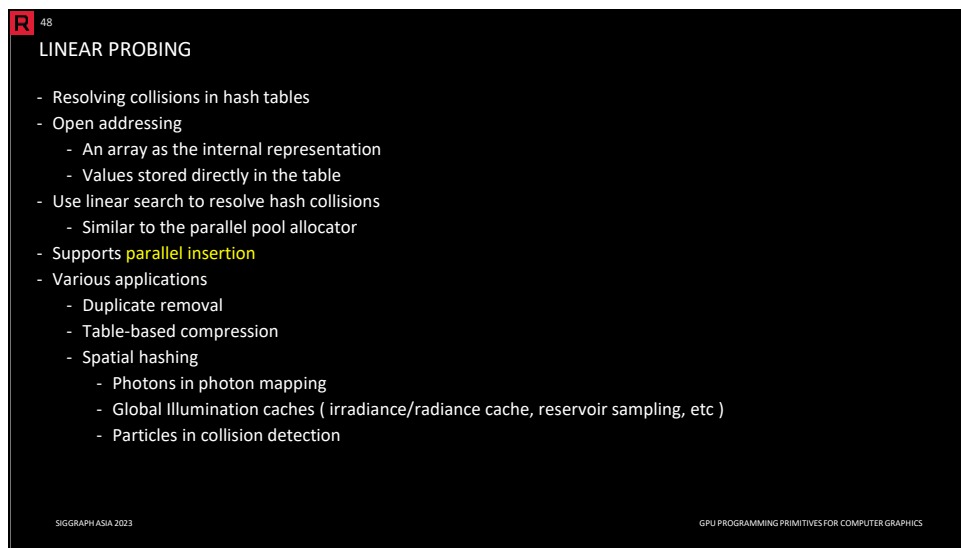
When a warp reaches a point that needs some allocation, it tries to acquire a lock of one of the buffers. It may fail in case the buffer is used by another warp. In that case, it tries the next one and so on, linearly searching for the first buffer that is free. This can be implemented as a spinlock with atomic *compare-and-swap* (*atomicCAS*). The first thread in the warp tries to acquire the lock, broadcasting the result to other threads in the warp. After the work is done, the buffer can be freed by releasing the lock via the atomic exchange (*atomicExch*). In this example, we assign the buffer per warp; we can also implement the assignment per block.

We will discuss the spinlock and this kind of linear search in detail in the following section.

Slide 47



In this section, we take a look into details of another primitive known as *linear probing*.



48

LINEAR PROBING

- Resolving collisions in hash tables
- Open addressing
 - An array as the internal representation
 - Values stored directly in the table
- Use linear search to resolve hash collisions
 - Similar to the parallel pool allocator
- Supports **parallel insertion**
- Various applications
 - Duplicate removal
 - Table-based compression
 - Spatial hashing
 - Photons in photon mapping
 - Global Illumination caches (irradiance/radiance cache, reservoir sampling, etc)
 - Particles in collision detection

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Linear probing is an algorithm for resolving collisions in hash tables that are represented by arrays. It is a form of *open addressing*, where values are directly stored in the hash table, in comparison with *chaining* that stores chains of values with the same hash that are typically represented as linked lists. As the name indicates, the search algorithm is linear from the hashed location of the key. An advantage is that insertion can be implemented in parallel thanks to the data structure simplicity, which is suitable for the GPU.

Various applications benefit from the use of linear probing, including duplicate removal, table-based compression, and spatial hashing. In computer graphics, techniques such as photon mapping, global illumination caching, and collision detection heavily rely on spatial hashing.

LINEAR PROBING - INSERTION

1. Allocate storage as an array with size N
2. Calculate hash for input X
3. Determine the home location based on $hash(X)$
 - The mapping from hash value to home location is arbitrary but $hash(X) \% N$ is an option
4. Insert the value if it is empty. Or return if X is already existing
5. Otherwise, try 4. Try the next location

home = $hash(X) \% N$

Linear search

home = $hash(X) \% N$

Linear search

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The core algorithm of linear probing works as follows.

First, we allocate storage of N items. We stick with a fixed size for the sake of simplicity. The following steps are calculating a hash of the value and looking up the home location. Each value has a home location. The mapping from hash to home location is arbitrary, but the simplest one is $hash(X) \% N$. Then, if the home location is empty, we insert the value; or if the value is already present, we are done.

Let us assume there are already some items present in the table. On the left side, we can see that the home location of X is occupied by C . Thus, we try the next location. As the value is already present, we are done. On the right side, as in the previous case, the home location of X is occupied by C . The following location is occupied by B . Fortunately, the next location is empty, and thus we can insert X there.

Notice that all search operations are done by linear search. That is why it is called *linear* probing.

PARALLEL LINEAR PROBING - INSERTION

- Insertion may cause data race in parallel execution
- Atomic CAS can handle it efficiently

threads

hash(X) % N == 1
hash(Y) % N == 2
hash(Z) % N == 3

[0]	[1]	[2]	[3]
	C	B	

hash(X) % N == 1
hash(Y) % N == 2
hash(Z) % N == 3

[0]	[1]	[2]	[3]
	C	B	Y

➡ Insertion can be done one by one

Implementation

Elements:	Occupied bit (1 bit)	Value bits (31 bits)
-----------	-------------------------	-------------------------

```

_device__ InsertionResult insert(unsigned int k)
{
    int h = home(k);
    for (int i = 0; i < m_table.size(); i++)
    {
        int location = (h + i) % m_table.size();
        unsigned int r = atomicCAS(&m_table[location], 0 /* empty */, k | OCCUPIED_BIT);
        if (r == 0)
        {
            return INSERTED;
        }
        else if (r == (k | OCCUPIED_BIT))
        {
            return FOUND;
        }
    }
    return OUT_OF_MEMORY;
}

```

Only a thread can insert ☺

SIGGRAPH ASIA 2023
GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

As previously mentioned, insertion in linear probing can be implemented in parallel in a GPU-friendly way. Insertion may cause data races in parallel execution as one location can be accessed by multiple threads.

This problem can be solved by *atomic compare-and-swap* (*atomicCAS*). This operation takes three arguments: *address*, *old*, and *value*. If the value pointed by *address* is equal to the *old value*, then it assigns the *value* to the location pointed by the *address*, while returning the original value at the *address*.

Assuming that empty locations are marked by zeroes, we can simply use atomic CAS for the insertion. We use the *occupied* bit to be able to handle zero as an input value. If the returned value is zero, we know that the insertion has been successful. If the returned value is equal to the one we want to insert, then the value is already present. Otherwise, we know that the entry is occupied by another value, and we try the next entry.

Notice that we already use the same approach in the *parallel pool allocator*.

Slide 51

51

LINEAR PROBING – HIGH LOAD FACTOR ISSUE

- Linear probing is **simple** and **very fast**
- What if the table is close to **full**?

hash(W) % N == 1

Too bad...

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Linear probing with parallel implementation is simple and fast, making it a practical choice. However, what happens if the table is almost full? For example, the home location (on the left) is occupied by *C*, and there are a lot of existing elements in a line. In such a case, we have to go all the way here, which is expensive. This is a drawback of linear probing.

BIDIRECTIONAL LINEAR PROBING

- Use the order of the hash values

$$home(X) = N \frac{hash(X)}{HASH_MAX + 1}$$

$hash(X) = \{0, 1, 2, 3, 4, 5\} \quad \{6, 7, 8, 9, 10\} \quad \{11, 12, 13, 14, 15\}$

Home locations:

[0]	[1]	[2]

$hash(C) = 6 \quad hash(B) = 8 \quad hash(D) = 13 \quad hash(E) = 14$

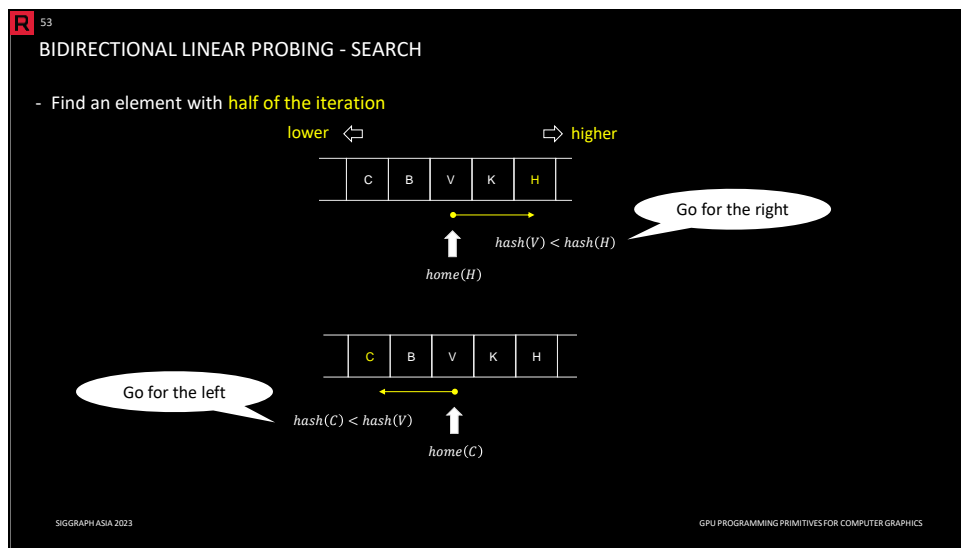
C	B	D	E
---	---	---	---

We can keep the hash value order for all elements

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

To address this issue, we introduce an extension of linear probing known as *bidirectional linear probing*. In the previous example, it would be better to go to the left to find the empty location more efficiently.

Bidirectional linear probing is based on ordering the hash values. *Home value* is defined by the equation above; the home locations are distributed based on the magnitude of the hash value: smaller hash values are distributed to the left while larger hash values are distributed to the right. In the example above, 0 to 5 are on the left, 6 to 10 are around the center, and the others are on the right. Moreover, we keep the order of the hash values in the storage even when there are some home conflicts.



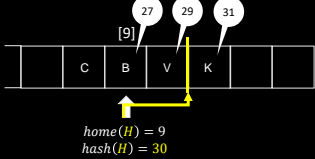
Thanks to the ordering, we can determine the direction of search and find an element with half of the iterations compared to standard (unidirectional) linear probing.

The example above depicts how the direction is determined. The lower values are on the left, and the higher values are on the right in the storage. We want to insert H (top), but the home location is already occupied by V . As the hash value of H is greater than the hash value of V , H should be located on the right, and thus we search on the right. Similarly, we want to insert C (bottom); as before, the home location is occupied by V . However, in this case, the hash value of C is lower than the hash value of V , and thus we search on the left side.

BIDIRECTIONAL LINEAR PROBING - INSERTION

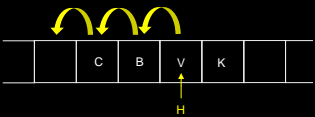
- Insertion

1. Search an insertion point to keep the order



$home(H) = 9$
 $hash(H) = 30$

2. Shift items and insert



H

💡 Move to the left or right?

The insertion point is on the right.

↓

The chunk is off-centered to the right.

↓

Let's move it to the left.

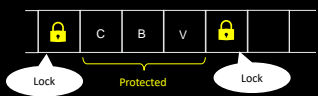
SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

This comes at a cost of keeping the order of the values. This means that when we insert a new value, we have to move the values to preserve the order.

That can be done in two steps. The first step is to find the insertion point. Similarly, as we did on the previous slide, we determine the home location and the search direction. The insertion point is a location such that the hash value of its element is less than the hash value of the inserted element and the hash value of the next location is greater than the hash value of the inserted element. In the example above, the home location is 9, and the search direction is right. The insertion point coincides with *V* as $hash(V) < hash(H)$ and $hash(H) < hash(K)$. The second step is to shift the elements to make an empty slot for the inserted element. We found an insertion point on the right. That means you took some iteration to find the item to the right. That means that the chunk is biased to the right, and thus we move it to the left. That is how bidirectional linear probing makes the search to be done in fewer iterations.

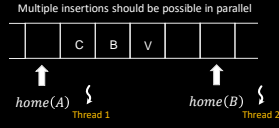
PARALLEL BIDIRECTIONAL LINEAR PROBING - INSERTION

- As a range of entries is affected, a single atomic CAS is not sufficient ☹
- However, it does not make sense to lock the whole hash table
 - It would result practically in a sequential execution
 - Only a fraction of the table is being modified
- Use a **region-based** lock instead
 - Put a lock flag at both edges of the occupied sequence in a spinlock style
 - Avoid the change of the sequence by other threads
 - Multiple locks can be held simultaneously



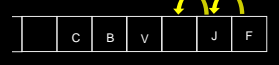
A hash table segment with six slots. The second, third, and fourth slots contain 'C', 'B', and 'V' respectively. The first and fifth slots are empty. A yellow bracket labeled 'Protected' spans from the first to the fourth slot. Yellow padlocks labeled 'Lock' are placed on the first and fifth slots.

Multiple insertions should be possible in parallel



Two threads, Thread 1 and Thread 2, are shown inserting elements. Thread 1 is at 'home(A)' and Thread 2 is at 'home(B)'. They are both pointing to the third slot, which contains 'B'. The diagram shows that multiple insertions can happen in parallel.

Avoid even concatenation of sequence

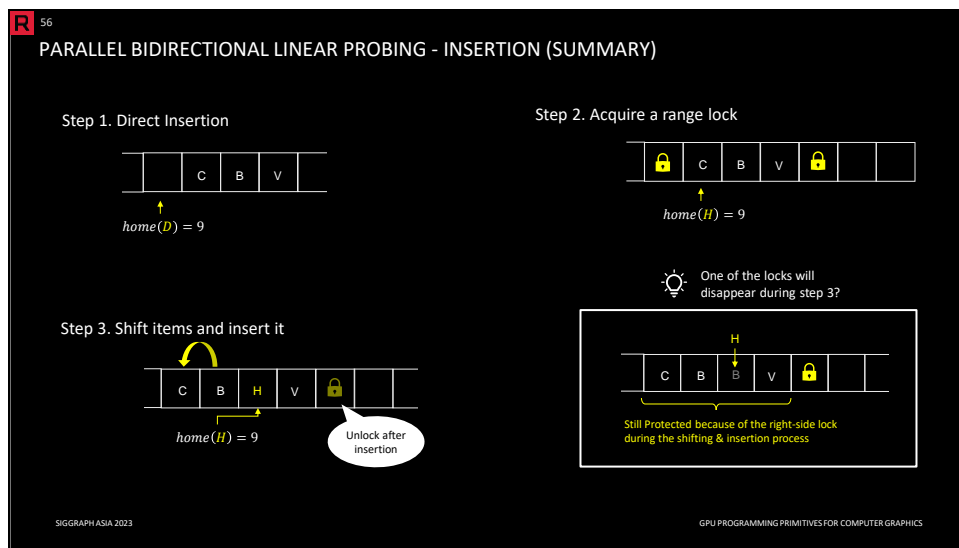


The diagram shows a hash table segment with six slots. The second, third, and fourth slots contain 'C', 'B', and 'V' respectively. The fifth and sixth slots are empty. A yellow bracket labeled 'Avoid even concatenation of sequence' spans from the first to the sixth slot. The diagram shows that the sequence of elements is maintained and no concatenation occurs.

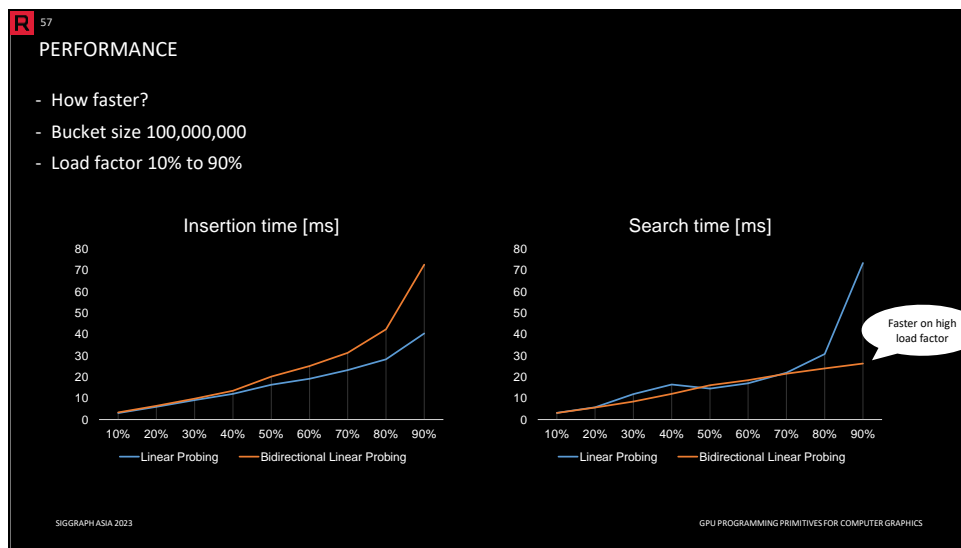
SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Unlike the standard (unidirectional) linear probing, a single *atomicCAS* is not sufficient as a range of entries might be affected. However, it does not make sense to lock the whole hash table globally as it would practically result in a sequential execution. It is also unnecessary as only a fraction of the table is being modified.

Therefore, we use region-based locks instead, marking empty locations at the beginning and the end of the modified occupied segment with lock flags (using the spinlock). In this way, the segment is uniquely identified by these two locks, and the race conditions do not happen. This would be more complicated if we allow to lock just a part of the occupied segment, requiring explicitly locking all entries. The elements can be either inserted directly if the target entry is empty (case 1); otherwise, we have to shift the elements to make a free slot for the inserted element (case 2).



The flow of the insertion is as follows. First, we try to insert the element directly via *atomicCAS*. If the insertion succeeded, we are done (step 1). Otherwise, the location is occupied, and we have to perform shifting before the actual insertion. We lock the occupied segment as we described on the previous slide (case 2). To prevent a deadlock, if the thread acquires the first lock but fails to acquire the second one, it releases the first one. If both locks have been successfully acquired, we shift the elements to make an empty space, and we insert the input element (case 3). Notice that we lose a lock on one side. It is still safe because there is one more lock on the right. That is also another reason why to have two locks.



The question is what is the performance of bidirectional linear probing in comparison with the standard (unidirectional) one? On this slide, you can see the times needed for insertion and search, respectively, in a hash table with 100 million entries for different load factors (i.e., the ratio between the number of items and the hash table size). As you can see, the insertion of bidirectional linear probing is slower, which is because the algorithm is more complex. However, for the search phase, you can see better performance with bidirectional linear probing when the load factor is higher. This is because simple linear probing needs a lot of iterations to find an item, while bidirectional linear probing significantly reduces it.

SPINLOCK ON GPU

- Bidirectional linear probing requires **exclusive locks**
- Is a simple spinlock implementation sufficient?

```

global __void Increment(int* counter, int* mutex)
{
    while(atomicCAS(mutex, 0, 1) != 0) { }
    __threadfence();
    (*counter)++;
    __threadfence();
    atomicExch(mutex, 0);
}
  
```

A critical section

➔ No. It can produce a deadlock

- Independent thread scheduling is an option
 - Allows such thread divergence
 - Not supported on all GPUs

Lock failed? atomicCAS() != 0

Threads: 0, 1, 2, 3

Lock status: T, F, T, T

Continue if any of the threads fail to lock

Yes (IMP) / No

Threads never reach here

A critical section

All threads try to synchronize to avoid thread divergence on SIMT¹

¹Single-instruction-multiple-threads (SIMT)

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Bidirectional linear probing requires exclusive locks as we saw earlier. The question is whether a naive spinlock implementation is sufficient. Spinlock can be implemented via *atomicCAS*; a critical section is guarded by thread fences between atomic operations. The answer is no as it can end up in a deadlock.

In the example above, there are four threads. Only one thread acquires the mutex. However, if the threads in the warp are (implicitly) synchronized, the first thread is waiting for others to join it after the while loop, but this will never happen as they are waiting for the mutex to be released. Therefore, the while loop never ends.

This is not an issue for architectures with *independent thread scheduling* (with asynchronous warps), but many architectures still use synchronous warps, and we should keep this problem in mind.

Note that we already used this kind of exclusive lock in the dynamic allocation. However, as the spinning threads are from different warps, this issue does not occur.

SPINLOCK ON GPU

- Explicit warp control
 - Combine atomicCAS and the critical section

```

global __void Increment(int* counter, int* mutex)
{
    bool done = false;
    do
    {
        if (done == false && atomicCAS(mutex, 0, 1) == 0)
        {
            __threadfence();
            (*counter)++;
            __threadfence();
            atomicExch(mutex, 0);
            done = true;
        }
    } while(__all(done) == false);
}

```

→ This does not require Independent thread scheduling ©

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Threads

0 1 2 3

done == false

F F F F

Inc Inc Inc Inc

Unlock Unlock Unlock Unlock

JMP if __all(done) == false

Loop until all items are done.

JMP

All the critical sections have done eventually
The threads can reach here ☺

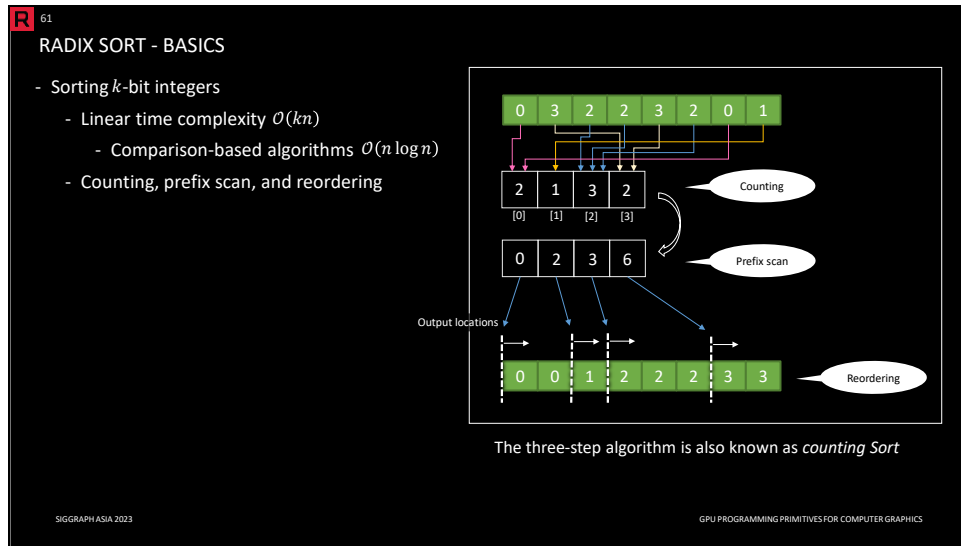
The good news is that we can fix it with relatively little effort. The trick is to let all threads in the warp participate in the while and postpone the exit until all threads are done. The thread that acquired the mutex can do its logic inside the if statement.

In the following example, the code can continue the logic even if some threads fail to acquire the lock. In each iteration, one thread acquires the mutex, does its work, releases the mutex, and sets the *done* flag as true. Eventually, all threads will be done, and the `__all` warp-level primitive will return true, allowing the threads in the warp to exit the loop. This logic works correctly on any platform.

Slide 60



In the following section, we explain *radix sort*, one of the most popular sorting algorithms. Sorting is a very general operation and the computer graphics area is not an exception.



The radix sort algorithm is a widely used sorting method that takes advantage of the binary representation of integers. Unlike comparison-based sorting algorithms, which have a time complexity of $\mathcal{O}(n \log n)$, radix sort has a time complexity of $\mathcal{O}(k \cdot n)$, where k represents the number of bits.

The algorithm consists of three operations: *counting*, *prefix sum*, and *reordering*. In the example above, we assume sorting keys with 2 bits (4 digits in $[0, 3]$):

- We count the occurrence of digits in the input values (a histogram with 4 bins). In particular, 0 occurs 2 times, 1 occurs 1 time, 2 occurs 3 times, and 3 is 2 times.
- We calculate the offset by performing the exclusive prefix scan on the histogram. The result is the sum of all values in preceding locations in the sequence (the offset for each digit).
- We reorder the sorting keys to the new locations indicated by these offsets. Once the sorting key is placed a new location, we increment the offset by one to provide the location for the next sorting key with the digit (in the case of duplicities).

This approach is also known as a *counting sort*.

RADIX SORT - BASICS

- The size of the histogram exponentially grows with the number of bits ☹
- 32-bit sorting keys require 2^{32} bins
- Least-significant-digit (LSD) radix sort
 - The three-step counting sort can preserve the order from the previous passes (stable sort)
 - Use multiple passes proceeding from lower bits to higher bits
 - 32-bit sorting keys and 8 bits at a time \rightarrow 4 passes

A decimal example for multiple passes

Pass 1:

01, 22, 04, 13, 06, 15

01, 22, 13, 04, 15, 06

1st digit sort

Pass 2:

01, 22, 13, 04, 15, 06

01, 04, 06, 13, 15, 22

2nd digit sort

Preserving the order

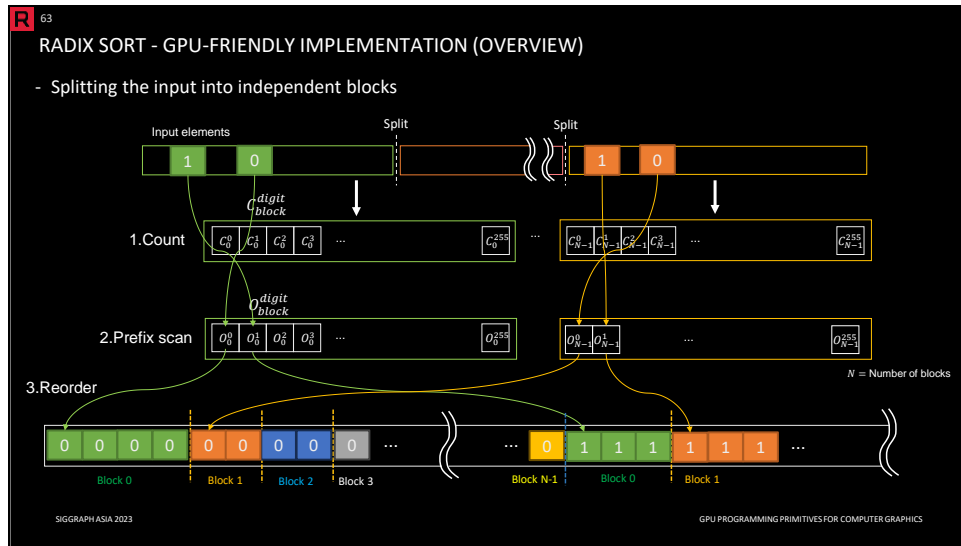
SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The previous example assumes only 2-bit sorting keys; however, in practice, we need to sort keys with significantly more bits (32 or 64 bits). The problem is that the number of bins in the histogram grows exponentially with the number of bits. For example, 32-bit sorting keys require a histogram with 2^{32} bins, which becomes practically inapplicable.

The idea of *radix sort* is to employ counting sort multiple times, processing only a fixed number of bits in each pass. For instance, assuming 32-bit sorting keys, if we process 8 bits at a time, it takes 4 passes in total.

The algorithm proceeds from the least significant digits (lower bits) to more significant ones (higher bits). This approach is also known as the *least-significant-digit* (LSD) radix sort in contrast to the *most-significant-digit* (MSD). Note that the MSD radix sort is not suitable for GPU processing, and we do not discuss it in this course. To preserve the order from the previous passes, the important is that the counting sort itself must be stable.

In the example above, we sort sorting keys with decimal digits in two passes. In the first pass (on the left side), we sort the keys according to the first digit (lower one). In the second pass (on the right side), we sort the keys according to the second digit (higher one), preserving the order from the first pass thanks to the stability of the counting sort.



In the following example, unless stated otherwise, we assume 32-bit sorting keys processing 8 bits in each pass. What is challenging is how to split the work into tasks that can be processed independently by each block (similarly to parallel prefix scan) and how to reconstruct the final result from these partial results.

As we already mentioned, the algorithm works iteratively, processing a fixed number of bits in each iteration, where each iteration consists of the following three steps:

- **Count:** We split the data into individual blocks and count the occurrences of digits in the input values in each block separately.
- **Prefix scan:** We compute offsets in the output buffer for all digits for each block via a single prefix scan.
- **Reorder:** We use the offsets from the previous step to determine output indices for individual sorting keys in each block.

RADIX SORT – COUNT

Counting the number of occurrences of digits within each block

Input buffer: 1 0 ...

Block 1 histogram: $c_0^0, c_0^1, c_0^2, c_0^3, \dots, c_0^{255}$

Block N-1 histogram: $c_{N-1}^0, c_{N-1}^1, c_{N-1}^2, c_{N-1}^3, \dots, c_{N-1}^{255}$

Number of bins in histogram: $2^8 = 256$ (8-bit)

N = Number of blocks

Block-wise counting with shared memory

```

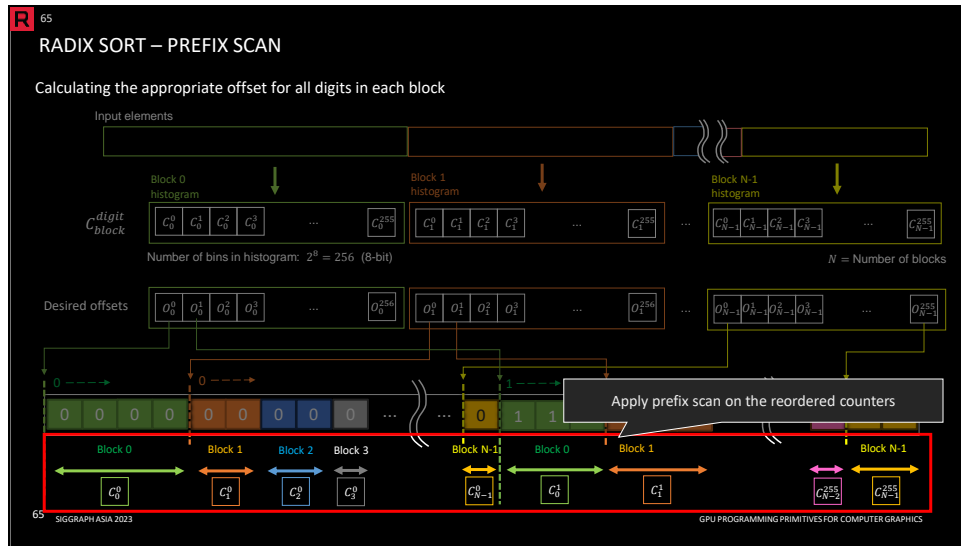
__shared__ histogram[BIN_SIZE]; // BIN_SIZE = 256
... Zero clear histogram here
for (int i = threadIdx.x; i < itemsPerBlock; i += blockDim.x)
{
    int indexOfItem = blockIdx.x * itemsPerBlock + i;
    if (indexOfItem < size)
    {
        int binIndex = (input[indexOfItem] >> START_BIT) & RADIX_MASK;
        atomicInc(&histogram[binIndex], 0xFFFFFFFF);
    }
}

```

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Let us check the details of each step.

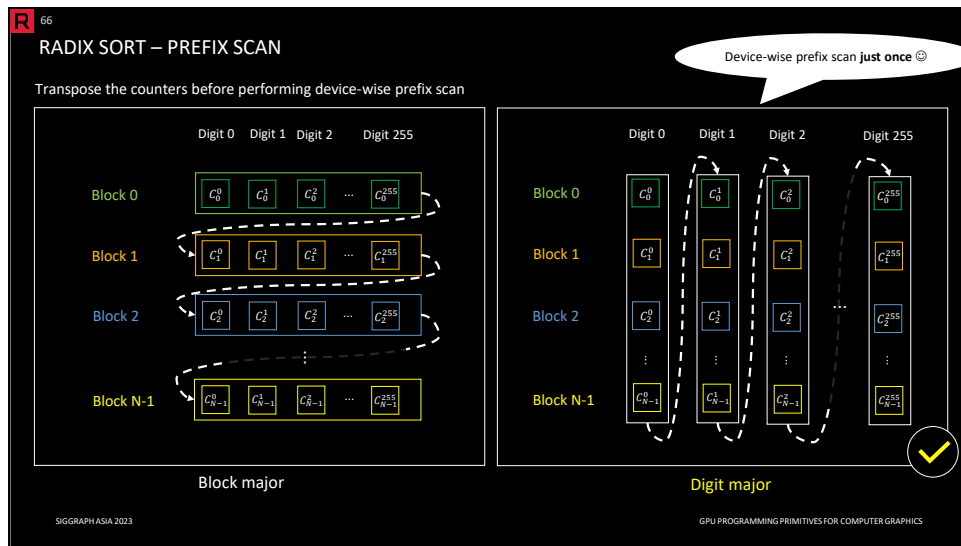
In the count step, we count occurrences of digits of the input values, which are derived from the sorting keys by masking the relevant bits. In the figure above, each block computes its own histogram (with $2^8=256$ bins) individually. Shared memory and atomics fit well for this histogram calculation as shown in the code example. The atomic addition prevents race conditions in the case of duplicities (i.e., two threads want to update the same counter).



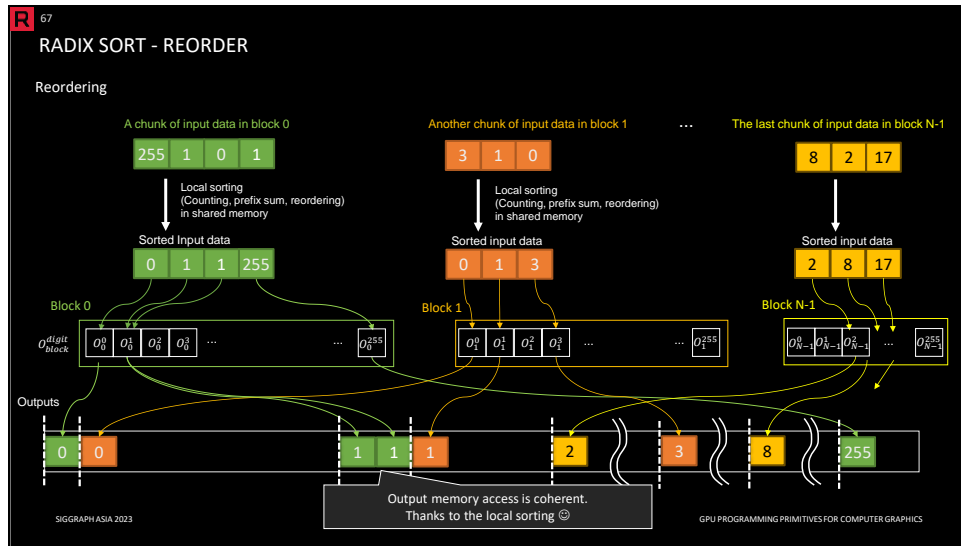
After each block processes its input and counts the occurrence of the digits, we utilize this result and calculate the offsets for all counts in the block.

Imagine how the layout of the final result should be: all 0s must be placed before all 1s, and all 1s before all 2s, and so on. We need to calculate how many 0s there are in total to determine the offset for 1s. Similarly, we need to calculate how many 0s and 1s there are in total to determine the offset for 2s. Furthermore, we want to determine the offsets of individual 0s (and similarly other digits): 0s of block 0 must be placed before 0s of block 1, and so on.

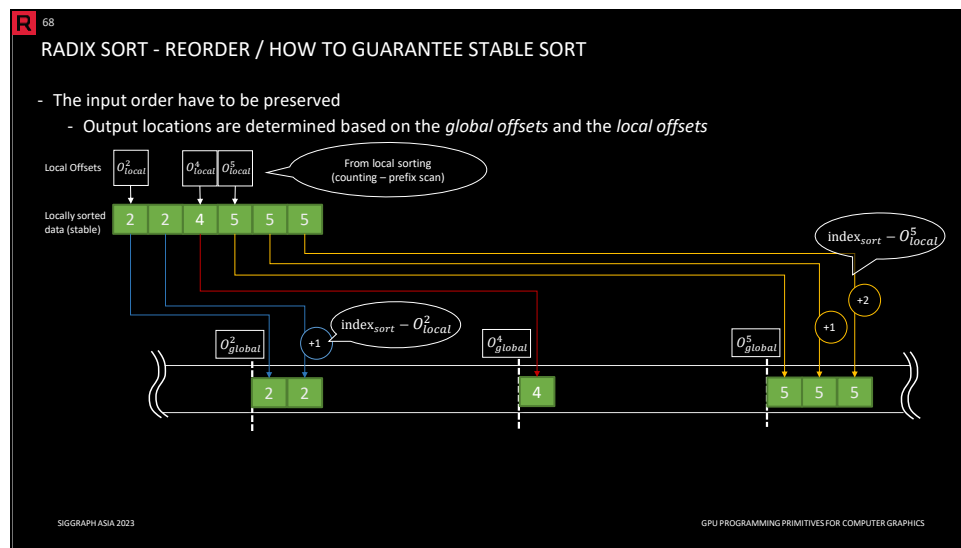
The question is how we can compute the offsets efficiently and in parallel.



To compute these offsets, we rearrange the count results such that we group the counts of 0s for all blocks followed by counts of 1s for all blocks, and so on. Note that this can be considered as a matrix transposition (i.e., switching the superscript and subscript). After rearranging, we apply the prefix scan to obtain the desired offsets that we use in the last step. The device-wise prefix scan here is calculated only once.



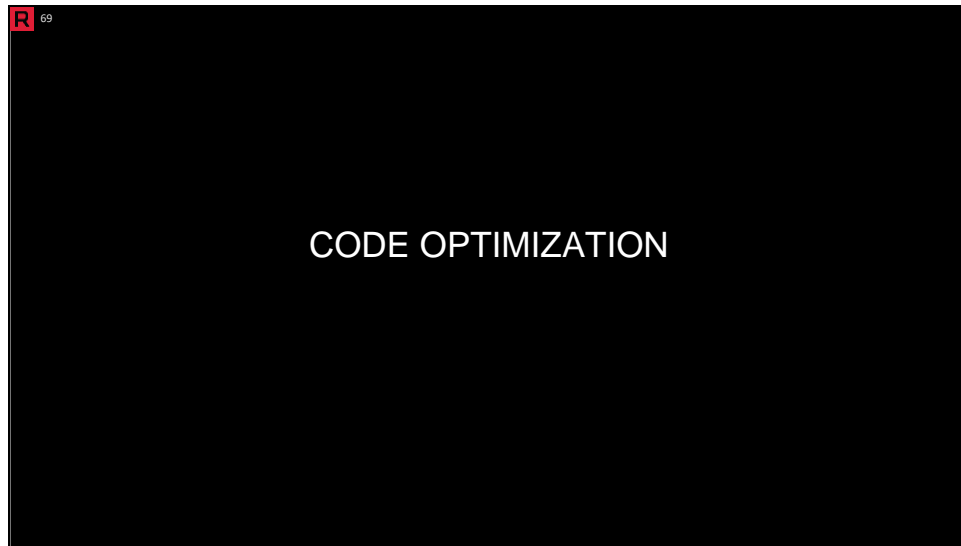
After we get the results from the previous prefix scan kernel, we reorder the input elements accordingly. The reordering process is conceptually straightforward, processing the input data one by one and putting them in new positions based on the corresponding offset while incrementing the offset. While this is true, we have to process values with the same digit sequentially to preserve the relative order, which limits parallelism. Furthermore, the output destination might be scattered, and thus memory accesses might be very incoherent, causing high memory latency.



To address this issue, we compute the *local index* for each input element by sorting the input elements locally (using a stable sorting algorithm) in the shared memory before applying the global offset from the previous kernel. After local sorting, we have a *local offset* (i.e., a prefix scan of the histogram of the block) for each digit along with sorted data. We also have the *global offset* from the previous slides.

We have all information needed to determine the output location. The *global offset* tells us where to start outputting elements with the same digit within a given block. We need to preserve the relative order of the values with the same digit. The values are already locally sorted in a stable way, and thus preserving the order. Therefore, we can output values with the same digit as they are sorted; we have to drop all preceding digits. In other words, the final output location is *global offset* (for each digit and each block) plus *sort index* (the block-wise stable sorting) minus *local offset* (dropping the previous digits).

Slide 69



In this section, we provide a couple of basic recommendations for the code optimization.

70
COALESCED MEMORY ACCESS TO GLOBAL MEMORY

- Sequential and dense memory accesses in a warp can be combined into a single transaction
 - Lower latency and higher throughput can be expected
 - Worth considering when the memory access is the bottleneck
 - E.g. Local sorting on reorder for radix sort

```

_device__ void copySomething(float* input, float* output)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    output[index] = input[index];
}

```

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Access to the global memory is very expensive in general. Memory coalescing is an access pattern allowing threads within a half-warp to combine their memory accesses into a single transaction. To achieve that, the consecutive threads within a half-warp must access the elements consecutively. The size of data elements must be 4B, 8B, or 16B with proper alignment (128B). Otherwise, the access is split into individual transactions (one per thread within the half-warp). Memory coalescing can be achieved by reordering the data beforehand, such as refactoring the data into structure-of-arrays (SoA).

Slide 71

71

COALESCED MEMORY ACCESS TO GLOBAL MEMORY

- Coalesced memory access is often fragile
 - Strided memory access ☹️

8 bytes stride (4 bytes gap)

```
struct Vector2
{
    float x;
    float y;
};

__device__ void copySomething(Vector2* output, Vector2* input)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    output[index].x = input[index].x;
}
```

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

72

COALESCED MEMORY ACCESS TO GLOBAL MEMORY

- Coalesced memory access is often fragile
 - Strided memory access ☹
- **Structure-of-Arrays** data layout helps to achieve memory coalescing
 - Guarantee proper sequentiality and data type (4, 8, or 16 bytes)

```
struct Matrix
{
    float a;
    float b;
    float c;
    float d;
    ...
};

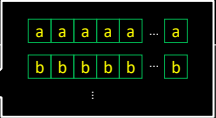
int index = blockIdx.x * blockDim.x + threadIdx.x;
data[index].a = ...;
```

Array-of-Structures (AoS)

```
struct Matrices
{
    float a[N];
    float b[N];
    float c[N];
    float d[N];
    ...
};

int index = blockIdx.x * blockDim.x + threadIdx.x;
data.a[index] = ...;
```

Structure-of-Arrays (SoA)



SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

BANK CONFLICTS IN SHARED MEMORY

- The shared memory is fast, but bank conflicts might hinder its performance ☹
- Multiple memory addresses are assigned at a bank

Bank	0	1	2	3	4	5	6	7	8	9	10	...	28	29	30	31
Address	0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31	32-35	36-39	40-43	...	112-115	116-119	120-123	124-127
	128-131	132-135	136-139	140-143	144-147	148-151	152-155	156-159	160-163	164-167	168-171	...	240-243	244-247	248-251	252-255

Conflicted memory accesses are serialized
Even the addresses are different ☹

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The shared memory has a significantly shorter latency compared to the global memory, but bank conflicts might hinder its performance. The shared memory banks are organized such that successive 4-byte words are assigned to successive banks (and the bandwidth is 4-byte per bank per clock cycle). The bank conflict occurs if two or more threads within a half-warp access the same bank. The exception is if all threads access the same bank (so-called broadcast). If bank conflicts occur, the memory accesses are serialized. In the example above, you can see memory is split into 16 banks. The bank conflicts can be avoided by using a different access pattern, e.g., structure-of-arrays (SoA).

BANK CONFLICTS IN SHARED MEMORY

- The shared memory is fast, but bank conflicts might hinder its performance.
 - Multiple memory addresses are assigned at a bank

Bank	0	1	2	3	4	5	6	7	8	9	10	...	28	29	30	31
Address	0-3 128-131	4-7 132-135	8-11 136-139	12-15 140-143	16-19 144-147	20-23 148-151	24-27 152-155	28-31 156-159	32-35 160-163	36-39 164-167	40-43 168-171	...	112-115 240-243	116-119 244-247	120-123 248-251	124-127 252-255

```

struct Vector
{
    float x;
    float y;
};
__shared__ Vector vectors[N];
objects[threadIdx.x].x = 42.0f;
  
```

Array-of-Structures (AoS)

Only half of the banks are utilized ☹
Conflicted memory accesses are serialized ☹

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The shared memory has a significantly shorter latency compared to the global memory, but bank conflicts might hinder its performance. The shared memory banks are organized such that successive 4-byte words are assigned to successive banks (and the bandwidth is 4-byte per bank per clock cycle). The bank conflict occurs if two or more threads within a half-warp access the same bank. The exception is if all threads access the same bank (so-called broadcast). If bank conflicts occur, the memory accesses are serialized. In the example above, you can see memory is split into 16 banks. The bank conflicts can be avoided by using a different access pattern, e.g., structure-of-arrays (SoA).

BANK CONFLICTS IN SHARED MEMORY

- The shared memory is fast, but bank conflicts might hinder its performance.
 - Multiple memory addresses are assigned at a bank
 - **Structure-of-Arrays** data layout may help

Bank	0	1	2	3	4	5	6	7	8	9	10	...	28	29	30	31
Address	0-3 128-131	4-7 132-135	8-11 136-139	12-15 140-143	16-19 144-147	20-23 148-151	24-27 152-155	28-31 156-159	32-35 160-163	36-39 164-167	40-43 168-171	...	112-115 240-243	116-119 244-247	120-123 248-251	124-127 252-255

```

struct Vectors
{
    float x[N];
    float y[N];
};
__shared__ Vectors vectors;
objects.x[threadIdx.x] = 42.0f;
  
```

Structure-of-Arrays (SoA)

All banks are utilized 😊

SIGGRAPH ASIA 2023 GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

The shared memory has a significantly shorter latency compared to the global memory, but bank conflicts might hinder its performance. The shared memory banks are organized such that successive 4-byte words are assigned to successive banks (and the bandwidth is 4-byte per bank per clock cycle). The bank conflict occurs if two or more threads within a half-warp access the same bank. The exception is if all threads access the same bank (so-called broadcast). If bank conflicts occur, the memory accesses are serialized. In the example above, you can see memory is split into 16 banks. The bank conflicts can be avoided by using a different access pattern, e.g., structure-of-arrays (SoA).

OCCUPANCY

- Streaming Multiprocessor (SM) can schedule multiple warps
 - Hiding latency to maximize throughput
- **Occupancy** is a ratio of the number of active warps per SM to the maximum number of possible active warps
 - Depending on **the register pressure, shared memory size, block size**, and device capability
 - Not a silver bullet, using vendor-provided profilers is recommended to analyze the latency or other bottlenecks

$$\text{Occupancy} = \frac{\text{The number of active warps}}{\text{Maximum number of warps on the HW}}$$

SIGGRAPH ASIA 2023

GPU PROGRAMMING PRIMITIVES FOR COMPUTER GRAPHICS

Latency hiding is a technique to substantially increase throughput by queuing a massive number of requests or tasks while waiting on expensive resources.

The *streaming multiprocessor* (SM) can schedule multiple warps to hide latency to maximize throughput. Specifically, the SM can schedule the warp that is ready to run and stalls the one that requires data. In general, we cannot fully utilize the hardware if there are not enough warps to be scheduled. However, it cannot hide latency if there is not enough concurrent warps.

For example, a single warp on the left needs to stall to obtain data from the memory. On the other hand, if you have multiple warps like the figure on the right, other warps can be scheduled while the warp requests data is waiting.

Occupancy is a ratio of the number of active warps with respect to the maximum number of possible active warps. This number is affected by the register pressure, shared memory size, and device capability; it is statically or dynamically measured. Since hardware resources (registers and shared memory) are allocated separately for each warp, we may need to reduce the usage to increase occupancy and improve the overall throughput.

Keep in mind that it is not a silver bullet. Relying on vendor-provided profilers is always recommended.

R

THANK YOU FOR YOUR ATTENTION!

Questions?



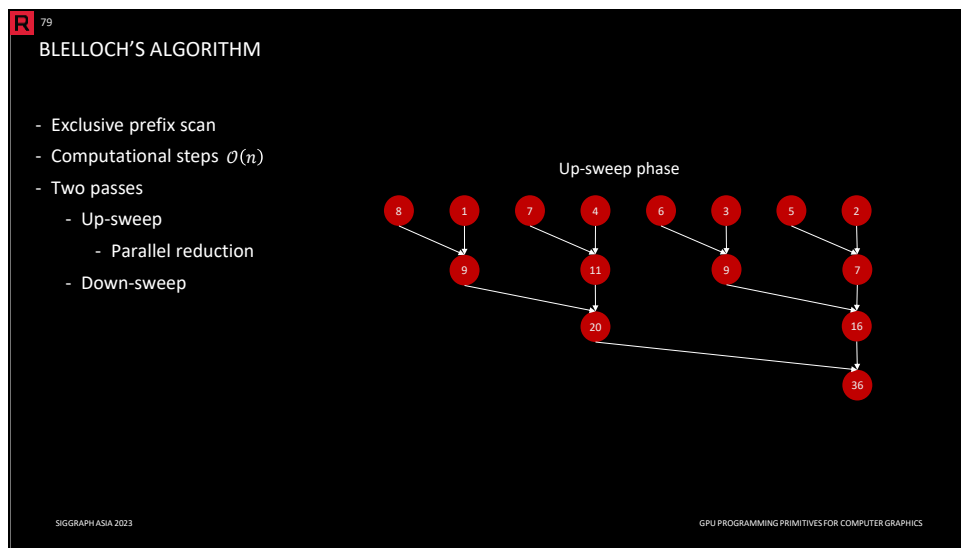
AMD 

ARR
Advanced Rendering Research Group

Slide 78

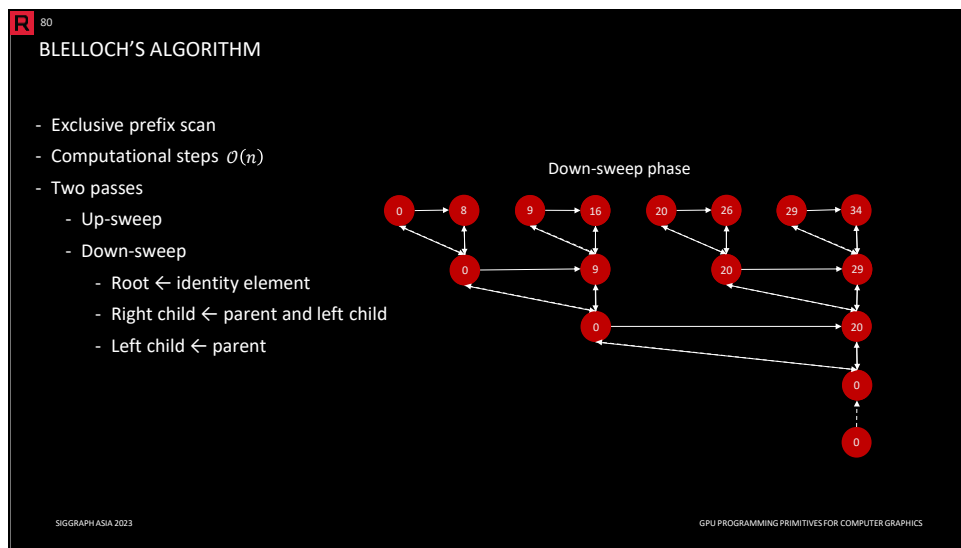


In this section, we provide a couple of additional slides that we left out due the time constrains of the course.

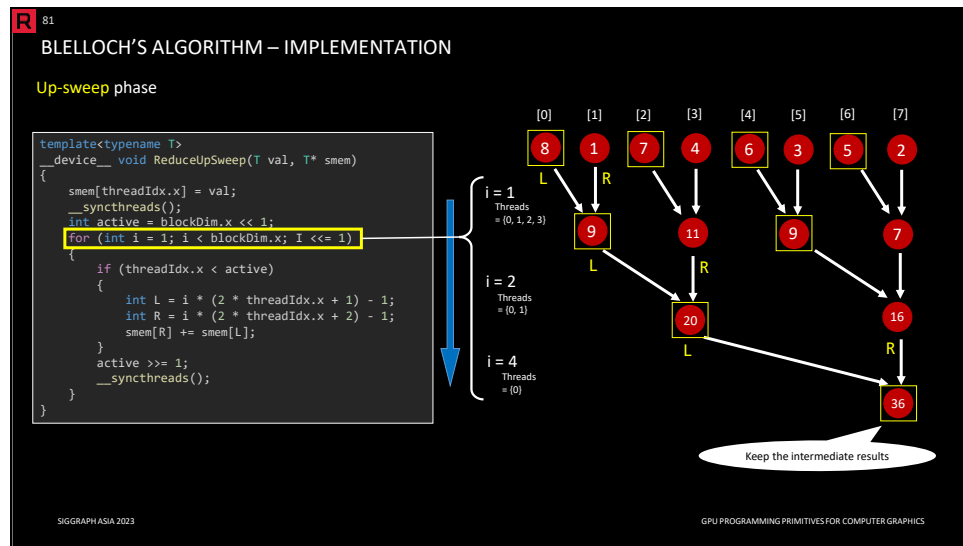


Blelloch's algorithm implicitly computes exclusive prefix scan in two passes (up-sweep and down-sweep) with $O(n)$ computational steps.

The up-sweep phase is practically a parallel reduction. A caveat is that Blelloch's algorithm uses not just the resulting sum but also intermediate results. The parallel reduction can be computed thanks to the associativity of the operator in any order, resulting in different partial sums. Therefore, we have to make sure that the up-sweep computation scheme corresponds to the down-sweep one.

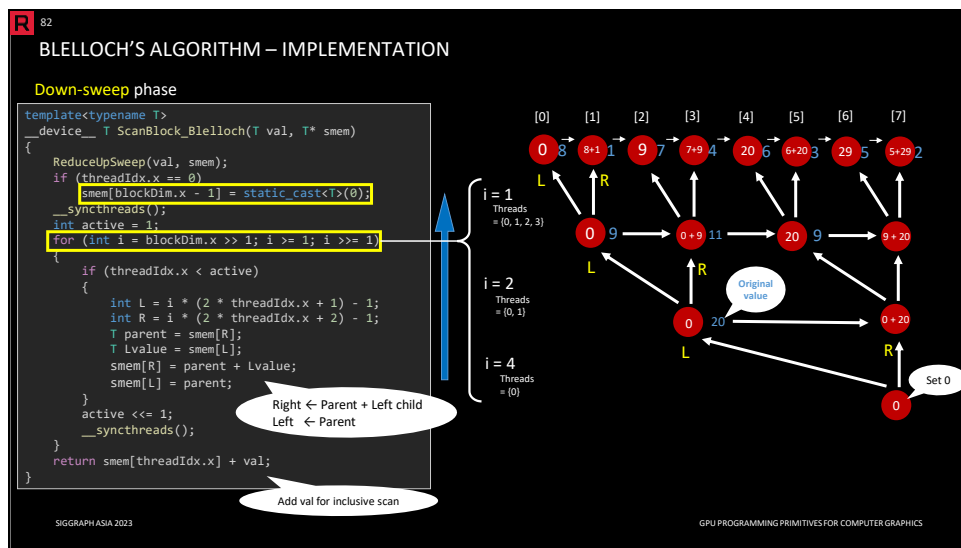


The down-sweep phase proceeds from the root of the reduction computational tree, using the partial sums from the previous phase to reconstruct the prefix scan. The goal is to modify the tree such that each interior node contains the sum of all leaves preceding the node in the preorder traversal. The root value is set to the identity element because there are no leaves preceding the root. Each left child node has the same number of preceding leaves as its parent node; hence each left child node has the same value as its parent node. The value of each right child node is the sum of the parent value and the left sibling value.

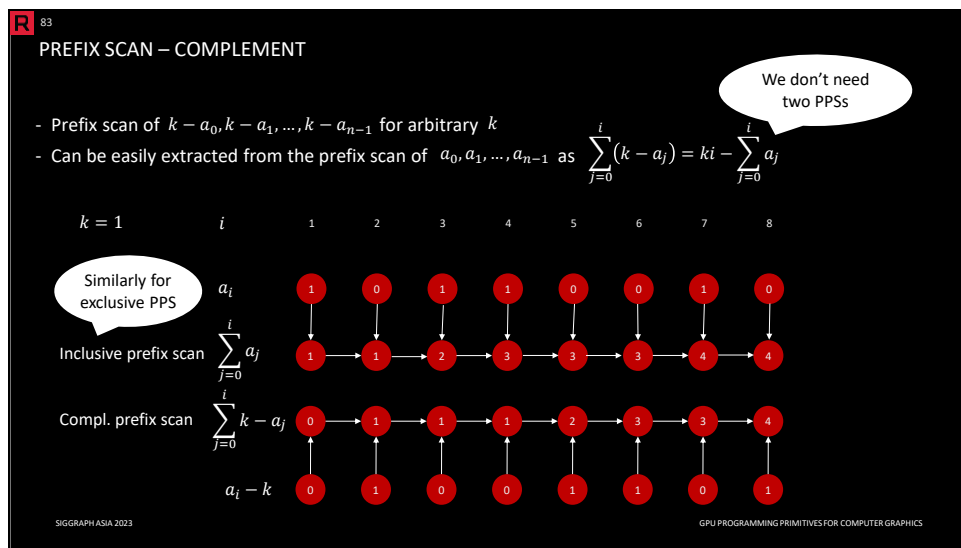


For standard parallel reduction, only what is important is the final result (e.g., the sum). In the implementation that we presented, eventually, all entries will contain the final sum as we let all threads participate in each iteration. In the up-sweep phase, we have to be careful not to overwrite the intermediate results that are important for the down-sweep phase.

Therefore, we let participate only threads they are contributing to a single computation tree (depicted above). We use sequential addressing with variable *active*, indicating how many threads are active in a particular iteration. The active threads are mapped to appropriate entries in the computational tree. The values are fetched from shared memory, subsequently added, and assigned back to shared memory.



In the down-sweep phase, we proceed from the root back to the leaves, using the values from the up-sweep phase (stored in shared memory). First, we replace the root value with the identity element (zero in the case of addition). In the main loop, we proceed in exactly opposite order than in the up-sweep phase. In each iteration, we assign the sum of the parent and the left child to the right child and the original value of the parent to the left child.



We define a prefix scan complement of a given sequence for arbitrary k as a prefix scan of a sequence where each element is the difference between k and the corresponding element of the original sequence. We can simply extract the prefix scan complement from the prefix of the original sequence without the necessity to compute it from scratch. Here you can see an example with binary values and $k=1$. This property might be handy for some practical applications.