Parallel Programming Patterns

03.05.23

Parallel Kernel

- Kernel is split up in blocks of threads

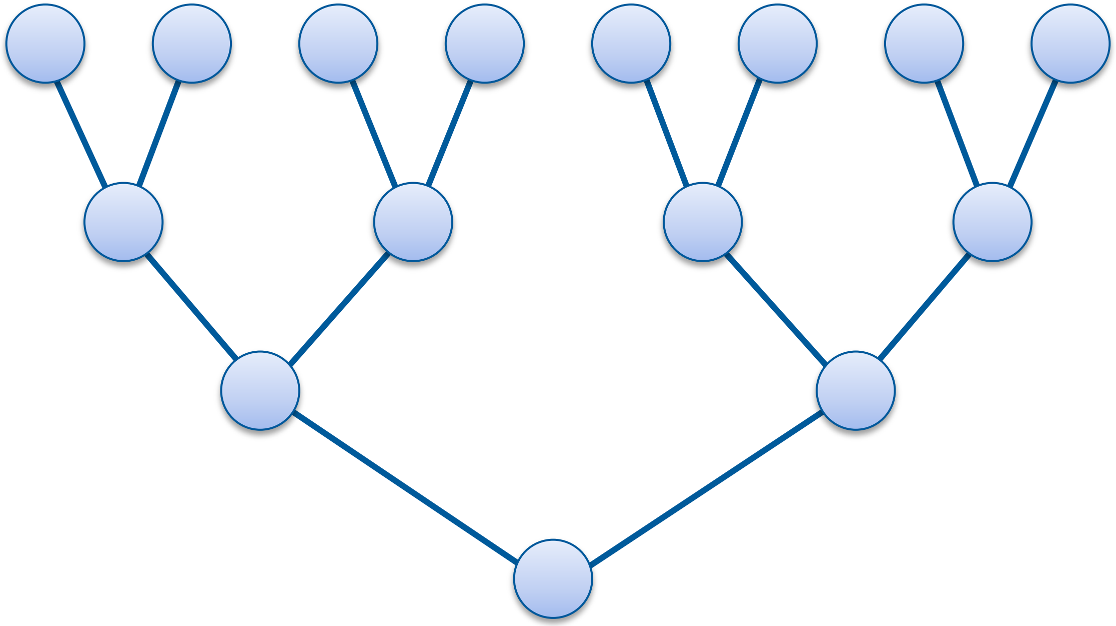
Kernel Launch

```
__global__ void myfunction(float* input, float* output) {
    uint id = threadIdx.x + blockIdx.x * blockDim.x;
    output[id] = input[id];
}


// ...
dim3 block_size(128, 1, 1);
dim3 grid_size(12, 1, 1)
myfunction<<<grid_size, block_size>>>(input, output);
```

(Parallel) Reduction

- Common and important data parallel primitive
  - Examples: minimum, maximum, sum
- Many data elements → single output (associative!)
- Easy to implement in CUDA
- Tree-based approach



Example Problem

- We generate data per thread, but don’t know beforehand how many
  - Where should each thread store the data?
  - Location depends on previous thread

    ➔ dependency chain that prevents parallelism

```
template <int T>
__global__ void produce(T* data) {
    int mynum = num_elements();
    for (int i = 0; i < mynum; ++i)
        data[???] = gen_data(i);
}
```

Parallel Prefix Sum

- Input: $x_0, x_1, x_2, x_3, \dots, x_n$
- Output: $y_0, y_1, y_2, y_3, \dots, y_n$

associative, binary operator

$$y_0 = x_0$$

$$y_1 = x_0 \oplus x_1$$

$$y_2 = x_0 \oplus x_1 \oplus x_2$$

$$y_n = x_0 \oplus x_1 \oplus x_2 \oplus \dots \oplus x_n$$

- Inclusive scan: `[2,3,3,1,0,1] → [2,5,8,9,9,10]`
- Exclusive scan: `[2,3,3,1,0,1] → [0,2,5,8,9,9]`

Parallel Prefix Sum: Work Efficiency

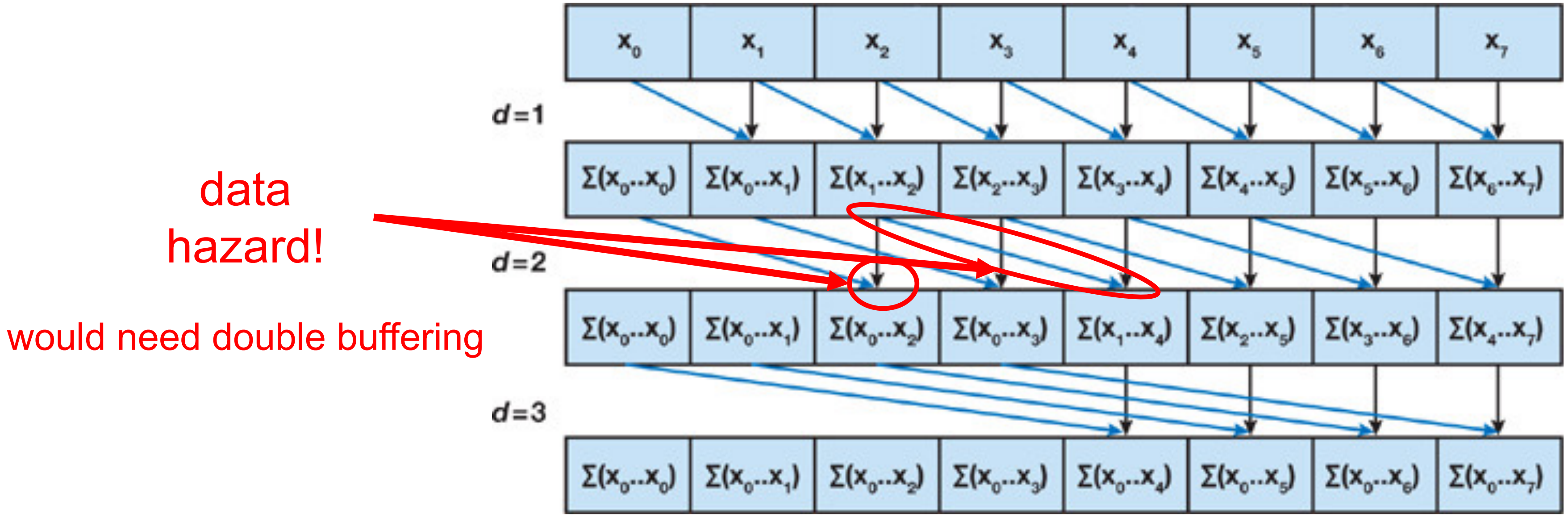
- Sequential algorithm
  - Complexity (order of growth) $O(n)$

```
template <int I>
void prefix_sum(I* in, I* out, int N) {
    out[0] = 0;
    for (int i = 1; i < N; ++i)
        out[i] = out[i-1] + in[i-1];
    return;
}
```

- Parallel algorithm
  - Complexity ???

- A parallel algorithm is **work-efficient**, if it performs the same amount of work as the (fastest) sequential algorithm.

Parallel Prefix Sum: Naïve Algorithm



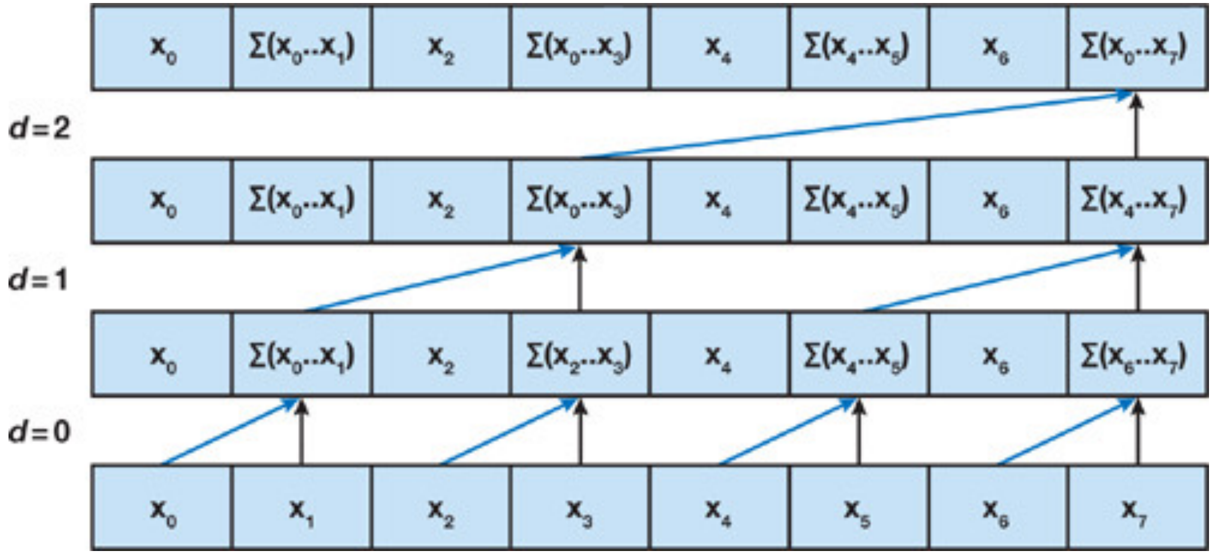
N = 8 → 8 + 6 + 4 operations
in general: $O(n \log n)$

not work-efficient: parallel algorithm does more work than sequential counterpart

also: assumes there are as many processors as elements

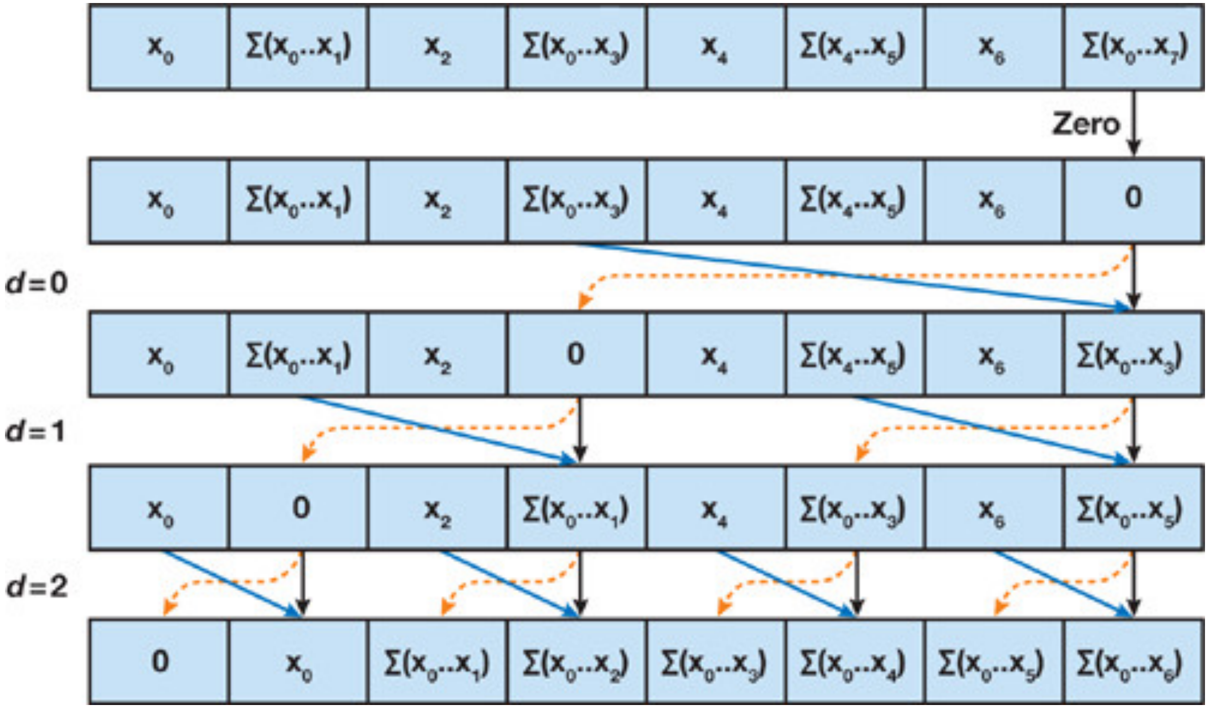
Parallel Prefix Sum: Up-Sweep

- Up-sweep (step1) performs a reduction
  - Root node (last node in the array) holds the sum of all nodes in the array

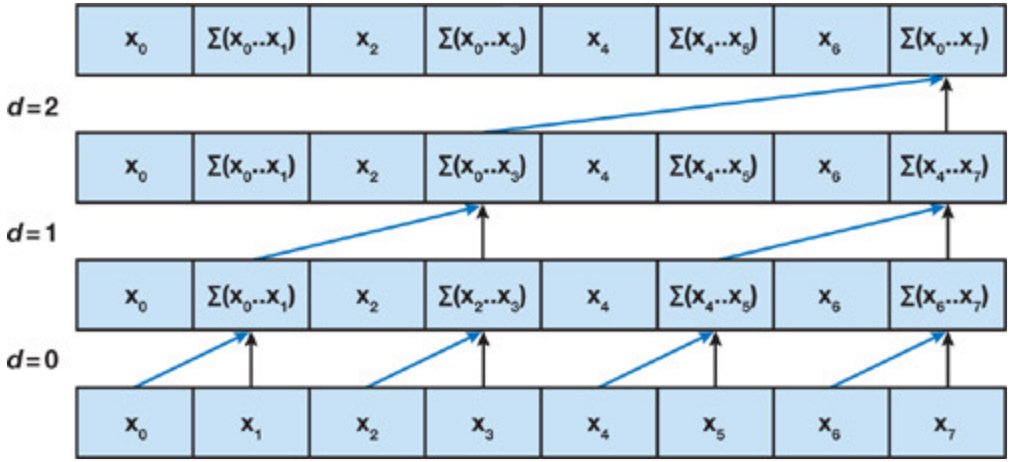


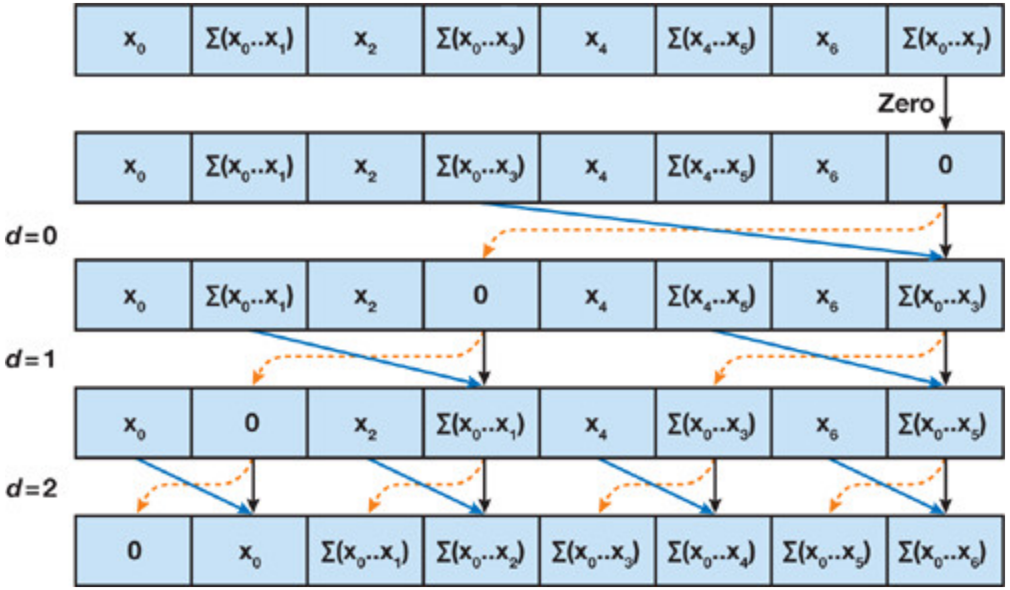
Parallel Prefix Sum: Down-Sweep

- Down-sweep (step 2) computes the scan based on the partial sums
  - Root node is initialized with 0



Parallel Prefix Sum



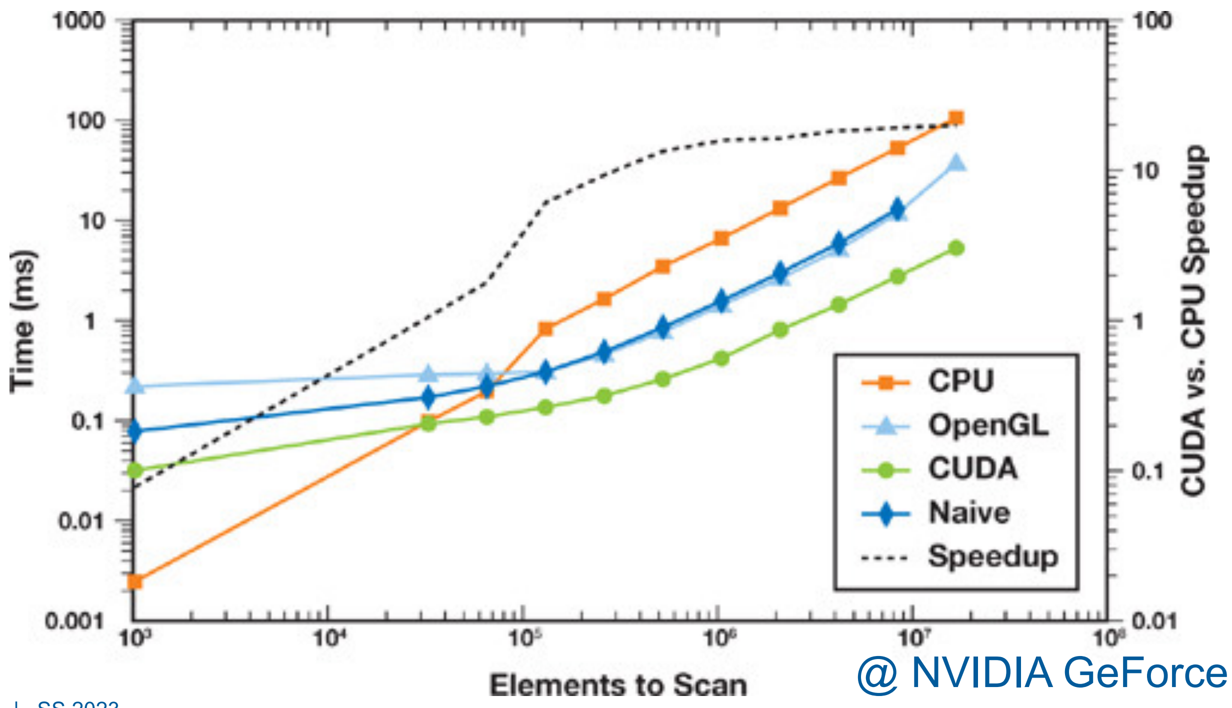


- *Guy Blelloch, 1990*

  Prefix Sums and Their Applications
- $O(n)$ operations → work-efficient
- No hazards within each step → can work in-place

Parallel Prefix Sum

- Can be efficiently implemented for parallel architectures
- GPU Gems 3 (2007), Chapter 39: Parallel Prefix Sum (Scan) with CUDA
  Mark Harris, Shubhabrata Sengupta, and John Owens



@ NVIDIA GeForce GTX 8800

Scan: Memory Offset

```
__global__ void analyze(int *numel) {
    // ...
    numel[tid] = num_elements();
}
template <int T>
__global__ void write(int *numel, T* data) {
    int mynum = num_elements();
    for (int i = 0; i < mynum; ++i)
      data[numel[tid] + i] = gen_data(i);
}

analyze<<<x, y>>>(numel);
int sum = scan_exlusive(numel);
data = gpu_malloc<T>(sum);
write<<<x, y>>>(numel, data);
```

numel: [2,3,3,1,0,1]

10 = [0,2,5,8,9,9]

data:

$[d_{0,0}, d_{0,1}, d_{1,0}, d_{1,1}, d_{1,2}, d_{2,0}, \ldots]$

Scan: Threads for Next Kernel

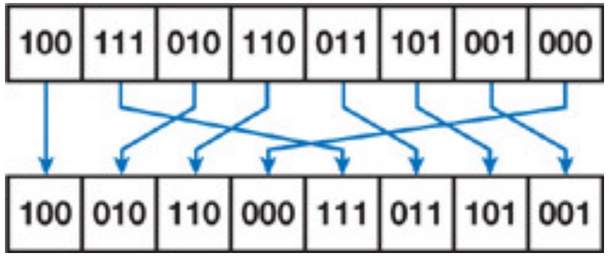
```
__global__ void this_step(int *numproc, ...) {
    // ...
    numproc[linid] = new_processors();
}
__global__ void map(int *numproc, int *mapping) {
    mapping[numproc[linid]] = 1;
}
__global__ void next_step(int *mapping, ...) {
    int myelement = mapping[linid];
    // ...
}

this_step<<<x, y>>>(numproc, ...);
int sum = scan_exlusive(numproc);
mapping = gpu_malloc_next_set<int>(sum, 0);
map<<<x, y>>>(numproc, mapping);
scan_inclusive(mapping);
next_step<<<sum/bs, bs>>>(mapping, ...);
```

```
numproc: [2,1,3,1]
      7 = [0,2,3,6]
mapping: [0,0,0,0,0,0,0]
mapping: [1,0,1,1,0,0,1]
mapping: [1,1,2,3,3,3,4]
```

Radix Sort using Scans

- Look at bit i of each key
  - Starting at LSB
- Reorder such that keys with 0 appear before keys with 1
  - Can be done using parallel scan
- Repeat with next bit until done



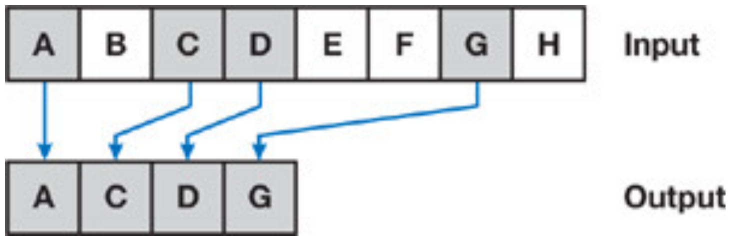
Sorting on the GPU

- Often required, often used
- Need more than $10k$ keys to be faster than CPU
- For a high number of keys, radix sort seems to perform best
- For a low number of keys ($< 10k$), bitonic merge sort might be a better choice
- For small amounts of data, insertion sort might even work well

  → experiment

Parallel Prefix Sum: Applications

- Stream compaction



- Sorting
- Load balancing
- Compare strings, add multi precision numbers, evaluate polynomials, solve recurrences, implement radix sort, implement quicksort, solve tridiagonal linear systems, delete elements from an array, dynamically allocate array elements, perform lexical analysis, search for regular expressions, implement tree operations, label components in two dimensional images, …

Atomic Operations

Synchronization – Data Races

- Task: List all possible outcomes!

```
int a, x, y;
void thread2() {
    a = 1;
    y = a;
}

void thread1() {
    thread t = createThread(thread2);
    a = 0;
    x = a;
    t.join();
    // what is x, y?
}
```

thread1	thread2
a = 0;	
x = a;	
	a = 1;
	y = a;
x = 0 && y = 1	

thread1	thread2
	a = 1;
	y = a;
a = 0;	
x = a;	
x = 0 && y = 1	

thread1	thread2
	a = 1;
a = 0;	
x = a;	
	y = a;
x = 0 && y = 0	

thread1	thread2
a = 0;	
	a = 1;
x = a;	
	y = a;
x = 1 && y = 1	

Synchronization – Data Races

- **Question:** What is the simplest C++ operation that can cause conflicts?
- **Answer:** `x++;`
- Explanation: Could translate into three assembly instructions:

```
mov rax, x    // load x into register
inc rax       // increase rax by one
mov x, rax    // store x
```

- **Task:** What are the possible outcomes when running two threads with `x++;` ?

Synchronization – Data Races

Definition: *Data Race*[1]:

- Non-deterministic behavior in a parallel programming
- Programming location without proper synchronization
- Occurs, if:
  - Concurrent tasks perform unsynchronized operations on the same address
  - One of the operations is a write
- **Problem:** Code may *randomly* operate correct or incorrect!

[1]The definition is from McCool et al., but they claim that they define a **race condition**.
But **race condition** and **data races** are something different.

Synchronization – Data Races

- How to properly handle data races?
- Programmer: Identify and mark *critical sections*
- **Critical Section:** Region of code that may not be executed in concurrently (or in parallel)
  - Not parallel to itself or any other piece of code
  - Protects a resource
    - CPU, memory location, device
  - Critical section gives exclusive access to the resource

Synchronization – Data Races

- Task: List all possible outcomes!

```
int a, x, y;
void thread2() {
    critical {
        a = 1;
        y = a;
    }
}

void thread1() {
    thread t = createThread(thread2);
    critical {
        a = 0;
        x = a;
    }
    t.join();
    // what is x, y?
}
```

Here the critical section locks the entire system

thread1	thread2
a = 0;	
x = a;	
	a = 1;
	y = a;
x = 0 && y = 1	

thread1	thread2
	a = 1;
	y = a;
a = 0;	
x = a;	
x = 0 && y = 1	

thread1	thread2
	a = 1;
a = 0;	
x = a;	
	y = a;
x = 0 && y = 0	

thread1	thread2
a = 0;	
	a = 1;
x = a;	
	y = a;
x = 1 && y = 1	

Mutual Exclusion

- CPU mutex strategies can hardly be used
- Locks not feasible with thousands of threads
  - Caution: deadlock as a result of hardware scheduling
  - Supported starting with Volta $CC \geq 7.0$
- Avoid operations on same memory
- Use data convergence points as sparsely as possible

  → rely on atomic operations

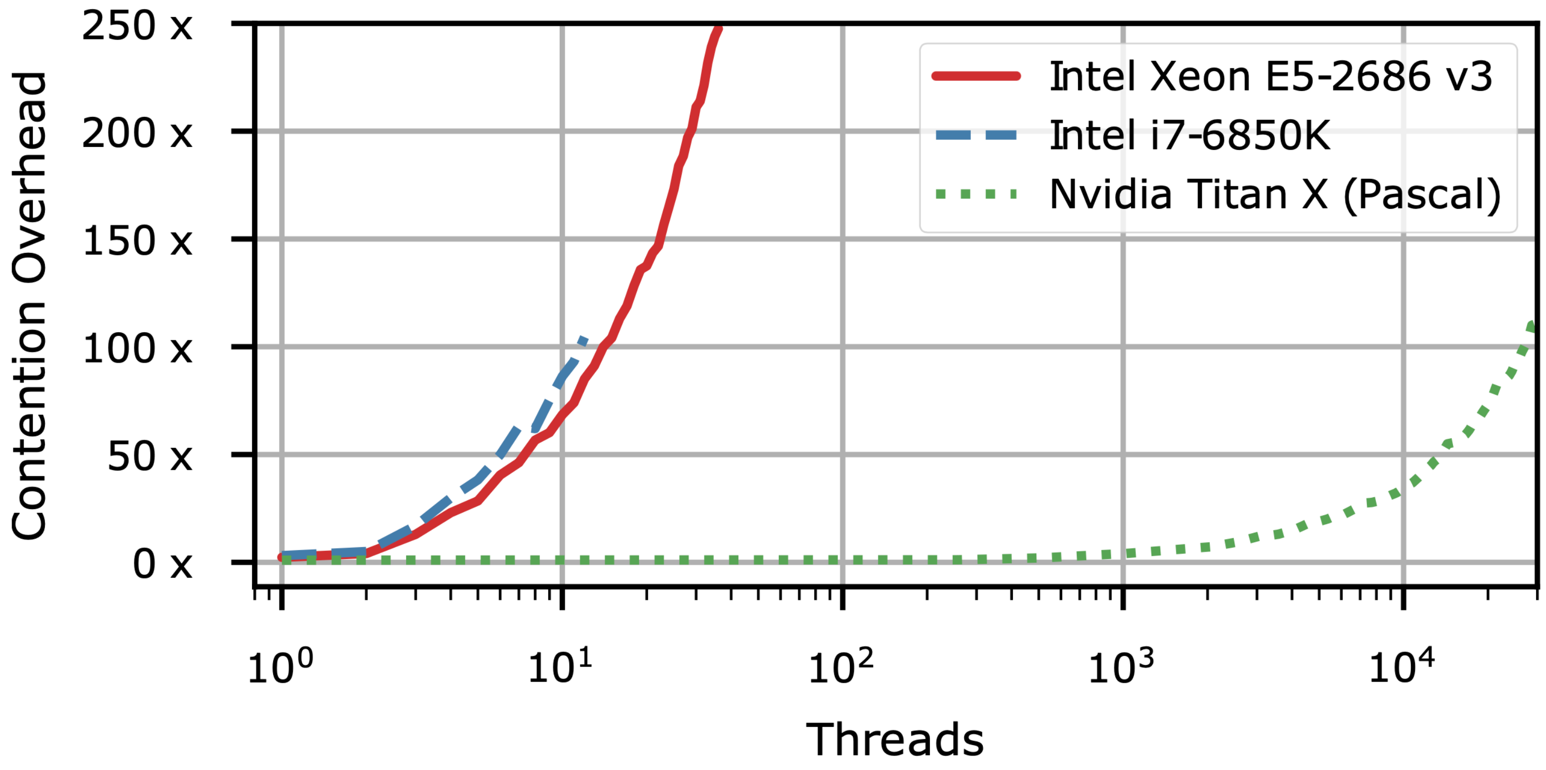
  (alternative: reduction)

Atomic Operators

- Atomic ops are supported for shared and global memory
- `atomicAdd`, `atomicSub`, `atomicExch`, `atomicMin`, `atomicMax`, `atomicInc`, `atomicDec`, `atomicCAS`, `atomicAnd`, `atomicOr`, `atomicXor`
- Some only available `int` data types (signed / unsigned 32-bit)
- One of the most important tools for parallel programming

- Contention when multiple threads access the same word in parallel

  → Performance loss relative to simple memory access



Atomic Operations Example

```
__device__ uint front, back;
__device__ T ring_buffer[RingSize];

__device__ void push(T data) {
    uint spot = atomicInc(&back, RingSize-1);
    ring_buffer[spot] = data;
}

__device__ T pop() {
    uint spot = atomicInc(&front, RingSize-1);
    return ring_buffer[spot];
}
```

simplified circular buffer

Atomic Operations Example

- Mutex using atomic operations ?

```
__device__ unsigned int mutex = 0;

__device__ void acquire() {
    while (atomicCAS(&mutex, 0, 1) == 1);
        /* mutex acquired */
}

__device__ void release() {
    atomicCAS(&mutex, 1, 0);
}
```

hardware scheduler tries to collect warp here!

Barriers and VOTEs

Synchronization

- Synchronize all threads in a block (device code)

```
void __syncthreads();
```

- Synchronize threads in a warp

```
void __syncwarp(unsigned int mask);
```

  - Wait for all warp lanes in mask $CC \geq 7.0$
  - All threads in mask must execute same `__syncwarp()` $CC < 7.0$
- Synchronize on kernel launch (host code)

```
void cudaDeviceSynchronize();
```

Synchronization with Voting

$CC \geq 2.0$

- Get additional information when synchronizing
  - Number of threads with predicate != 0

```
int __syncthreads_count(int predicate);
```

  - Logical and on predicate of all threads

```
int __syncthreads_and(int predicate);
```

  - Logical or on predicate of all threads

```
int __syncthreads_or(int predicate);
```

Atomic Operations and Barriers

```
__device__ uint front, back;
__device__ T ring_buffer[RingSize];

__device__ void push(T data) {
    uint spot = atomicInc(&back, RingSize);
    ring_buffer[spot] = data;
}

__device__ T pop() {
    uint spot = atomicInc(&front, RingSize);
    return ring_buffer[spot];
}
```

simplified circular buffer

Barriers and Fences

- Make sure that data is available before continuing
  - `__threadfence()` halts the current thread until all previous writes to shared and global memory are visible by other threads

```
__device__ uint front, back;
__device__ T ring_buffer[RingSize];
__device__ uint ring_buffer_flags[RingSize];

__device__ void push(T data) {
    uint spot = atomicInc(&back, RingSize);
    ring_buffer[spot] = data;
    __threadfence();
    ring_buffer_flags[spot] = Ready;
}
```

make sure data is visible

flag as being ready

simplified circular buffer #2

Warp Votes

- Fast votes within warps
  - non-zero if all threads' predicates are non-zero

```
int __all_sync(unsigned int mask, int predicate);
```

  - non-zero if any thread's predicate is non-zero

```
int __any_sync(unsigned int mask, int predicate);
```

  - n^{th} bit is set if n^{th} thread's predicate is non-zero

```
int __ballot_sync(unsigned int mask, int predicate);
```

  - `mask`: bitmask identifying which threads of the warp participate
- Active mask of threads

```
unsigned int __activemask();
```

Votes

```
do {
    // ...
    bool run = should_run();
} while (__any_sync(0xFFFFFFFF, run));
```

Example 1: state-based loop

```
__device__ void myfunction(...) {
    uint bres = __activemask();
    uint active_threads = __popc(bres);
    // ...
}
```

Example 2: active threads

Warp Shuffle

$CC \geq 3.0$

- Exchange a variable between threads within a warp without shared memory
- Can only read values from participating threads

```
T __shfl_sync(unsigned int mask, T var,
              int src_lane, int width = WarpSize);
```

- Special shuffle functions with deltas and XOR
- Enables faster reduction methods
- Reduces shared memory pressure

Warp Match

$CC \geq 7.0$

- Broadcast-and-compare operations within warp
  - Returns **mask** of thread that have same value for **value** in **mask**

```
T __match_any_sync(unsigned int mask, T value);
```

  - Returns **mask** if all threads in **mask** have the same value for **value**, otherwise 0 is returned. **predicate** is set to true if all values in **mask** have same value, otherwise false

```
T __match_all_sync(unsigned int mask, T value, int* predicate);
```