Parallel Programming

CUDA C Extensions and Basic APIs

`

Overview

- CUDA C Extensions
  - Function type qualifiers
  - Variable qualifiers
  - Built-in types
  - Built-in variables
- CUDA Basic APIs
  - Memory management
  - Execution configuration & thread synchronization
  - Event management & error handling

Function Type Qualifiers

- Specify (1) whether a function executes on the host or on the device and (2) whether it is callable from the host or from the device.
- *__global__*
- *__device__*
- *__host__*

The __*global*__ qualifier

- Declares a function as being a kernel:
  - Executed on the device
  - Callable from the host
  - Callable from the device for devices of compute capability 3.x and up
- __*global*__ functions must have *void* return type.
- Any call to a __global__ function must specify its execution configuration <<<....>>>.
- A call to a __global__ function is asynchronous:
  - Returns before the device has completed its execution

The __*device*__ qualifier

- Executed on the device
- Callable from the device only
- No execution configuration
- No restriction on function types
- Can call another __*device*__ function
- Synchronous

The __*host*__ qualifier

- Executed on the host
- Callable from the host only
- Is optional:
  - Equivalent to without any of the three qualifiers
- Can be used together with __*device*__
  - compiled for both the host and the device
  - Inside the function the __*CUDA_ARCH*__ value tells whether it is for the host or the device

Example of __host__ __device__

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ >= 300
   // Device code path for compute capability 3.x
#elif __CUDA_ARCH__ >= 200
   // Device code path for compute capability 2.x
#elif __CUDA_ARCH__ >= 100
  // Device code path for compute capability 1.x
#elif !defined(__CUDA_ARCH__)
  // Host code path
#endif
}
```

Variable Type Qualifiers

- A variable type qualifier specifies the memory location of the variable **on the device**.
- Three type qualifiers for variables on the device:
  - *__device__*
  - *__constant__*
  - *__shared__*
- A variable declared in **device code** without a type qualifier typically resides in the register.

The __*device*__ qualifier

- Declares a variable that resides on the device
  - Resides in global memory space
  - Has the lifetime of an application
  - Is accessible from all the threads within the grid
  - Is accessible from the host through the runtime library
    - *cudaGetSymbolAddress(), cudaGetSymbolSize(),*
    - *cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()*

Example of __device__ variable

```
__device__ int d_value;
__global__ void test_Kernel()
{
	int threadID = threadIdx.x;
	d_value = 1;
	printf("threadID %-3d d_value%3d\n",threadID,d_value);
}
int main()
{
	int h_value = 0;
	test_Kernel<<<1,2>>>();
	cudaMemcpyFromSymbol(&h_value,d_value,
		sizeof(int),0,cudaMemcpyDeviceToHost);

	printf("Output from host: %d\n",h_value);
	return 0;
}
```

The __*constant*__ qualifier

- Declares a variable that
  - Resides in constant memory space (read-only)
  - Has the lifetime of an application
  - Is accessible from all the threads within the grid and from the host through the runtime library
  - Optionally used together with __device__

The __*shared*__ qualifier

- Declares a variable that
  - Resides in the shared memory space of a thread block
  - Has the lifetime of the block
  - Is only accessible from all the threads within the block
  - Optionally used together with __device__

Built-In Vector Types

- Structures of *<basic_type><i> (i=1,2,3,4)*
  - *char, uchar, short, ushort, int, uint*
  - *long, ulong, longlong, ulonglong*
  - *float, double* (*i=1,2* for *double* vectors)
- 1st, 2nd, 3rd, and 4th components (if any) are accessible through the fields *x*, *y*, *z*, and *w*, respectively
- Constructor in the form: *make_<type name>*
  - E.g., *int2 make_int2(int x, int y);*

Built-In Variables

- *gridDim*, *blockDim*
  - Both are of type *dim3* (based on *uint3*)
- *blockIdx*, *threadIdx*
  - Both are of type *uint3*
- *warpSize*
  - Type *int*; size of warp in number of threads

Frequently Used CUDA Types

- CUDA stream type
  - *typedef CUstream_st * cudaStream_t*
- CUDA event type
  - *typedef CUevent_st * cudaEvent_t*
- CUDA Error type
  - *typedef enumcudaError cudaError_t*

Memory Allocation and Deallocation

*cudaError_t cudaMalloc (void** devPtr,*
size_t size)

Allocate memory on the device.

cudaError_t cudaFree (void devPtr)*

Free memory on the device.

Get Memory Address and Size

cudaError_t cudaGetSymbolAddress
*(void** devPtr, const void* symbol)*

Find the address associated with a CUDA symbol.

cudaError_t cudaGetSymbolSize
(size_t size, const void* symbol)*

Find the size of the object associated with a CUDA symbol.

Memory Copy between Host Variables

cudaError_t cudaMemcpy
(void dst, const void* src, size_t count,*
cudaMemcpyKind kind)
Copy data between host and device.

enum cudaMemcpyKind
cudaMemcpyHostToHost (= 0: Host -> Host)
cudaMemcpyHostToDevice (= 1: Host -> Device)
cudaMemcpyDeviceToHost (= 2: Device -> Host
cudaMemcpyDeviceToDevice (= 3: Device -> Device)
cudaMemcpyDefault (= 4: Default unified virtual address space)

Memory Copy for Device Variable

cudaError_t cudaMemcpyToSymbol
(const void symbol, const void* src,*
size_t count, size_t offset = 0,
cudaMemcpyKind kind = cudaMemcpyHostToDevice)

Copy data to the given symbol on the device.

cudaError_t cudaMemcpyFromSymbol (void dst,*
const void symbol, size_t count, size_t offset = 0,*
cudaMemcpyKind kind = cudaMemcpyDeviceToHost)

Copy data from the given symbol on the device.

Execution Configuration

<<< Dg, Db, Ns, S >>>

- Dg is of type dim3 and specifies the dimension and size of the grid, such that Dg.x * Dg.y * Dg.z equals the number of blocks being launched;
- Db is of type dim3 and specifies the dimension and size of each block, such that Db.x * Db.y * Db.z equals the number of threads per block;
- Ns is of type size_t and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory. Ns is an optional argument which defaults to 0;
- S is of type cudaStream_t and specifies the associated stream; S is an optional argument which defaults to 0.

Thread Synchronization

[DEPRECATED]:
cudaError_t cudaThreadSynchronize (void)
Wait for compute device to finish.
Should use:
cudaError_t cudaDeviceSynchronize (void)
Wait for compute device to finish.

Within a block of threads:
void __syncthreads();

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to *__syncthreads()* are visible to all threads in the block.

Event Management

cudaError_t cudaEventCreate (cudaEvent_t* event)

Creates an event object.

cudaError_t cudaEventCreateWithFlags (cudaEvent_t* event, unsigned int flags)

Creates an event object with the specified flags.

cudaError_t cudaEventDestroy (cudaEvent_t event)

Destroys an event object.

cudaError_t cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end)

Computes the elapsed time between events.

cudaError_t cudaEventQuery (cudaEvent_t event)

Queries an event's status.

cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)

Records an event.

cudaError_t cudaEventSynchronize (cudaEvent_t event)

Waits for an event to complete.

Error Handling

const __cudart_builtin__ char cudaGetErrorName (cudaError_t error)*

Returns the string representation of an error code.

const __cudart_builtin__ char cudaGetErrorString (cudaError_t error)*

Returns the description string for an error code.

cudaError_t cudaGetLastError (void)

Returns the last error from a runtime call and resets it to *cudaSuccess*.

cudaError_t cudaPeekAtLastError (void)

Returns the last error from a runtime call.

Summary

- CUDA function type qualifiers specify where a function to be executed and to be called.
- CUDA variable type qualifiers specify where a device variable resides.
- CUDA has its own data types extended from C.
- CUDA has common memory management, event management, thread synchronization, and error handling functions.

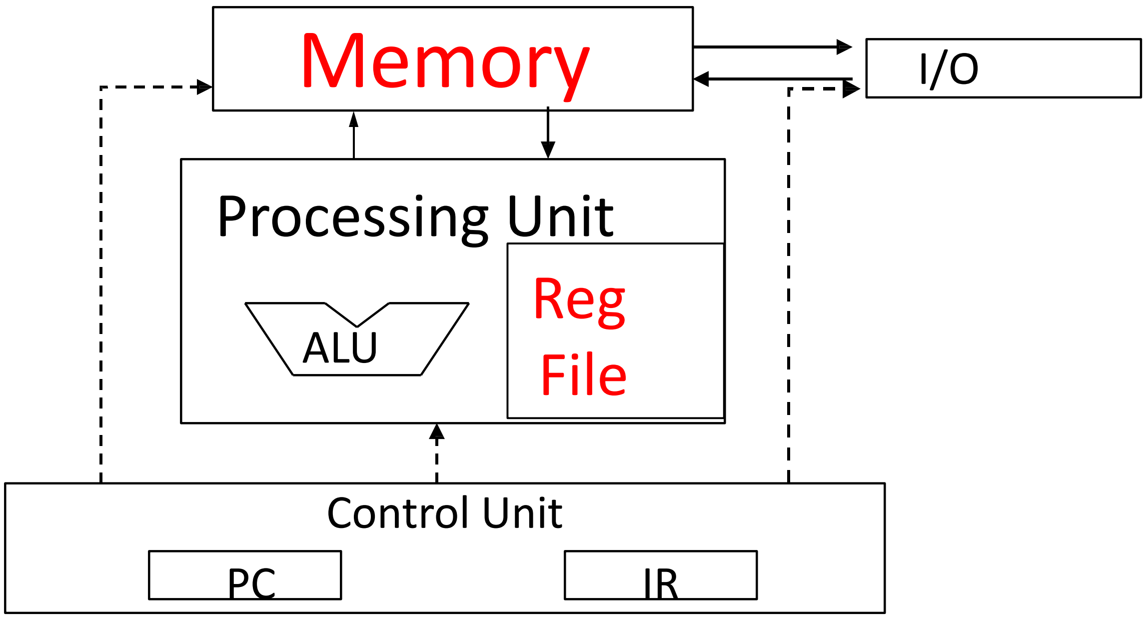
Parallel Programming

CUDA Memories and Optimizations

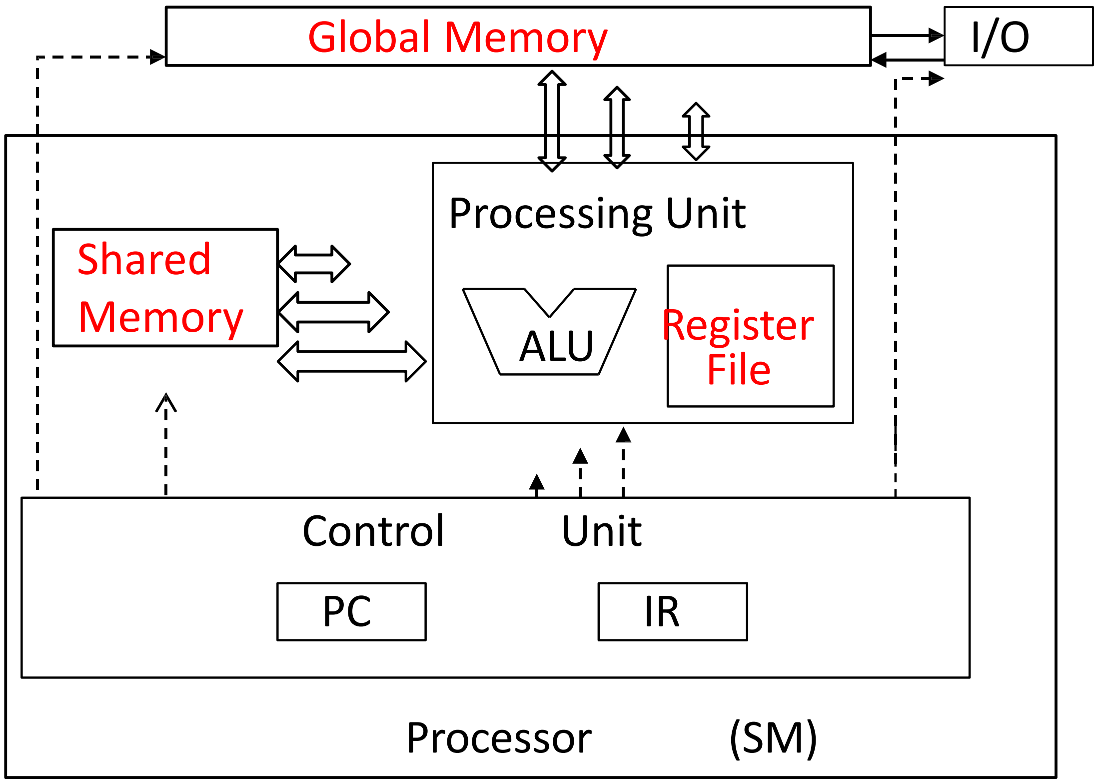
Overview

- CUDA Memories
  - Registers, shared memory, global memory
- Memory optimizations
  - General memory optimizations
  - Use of shared memory

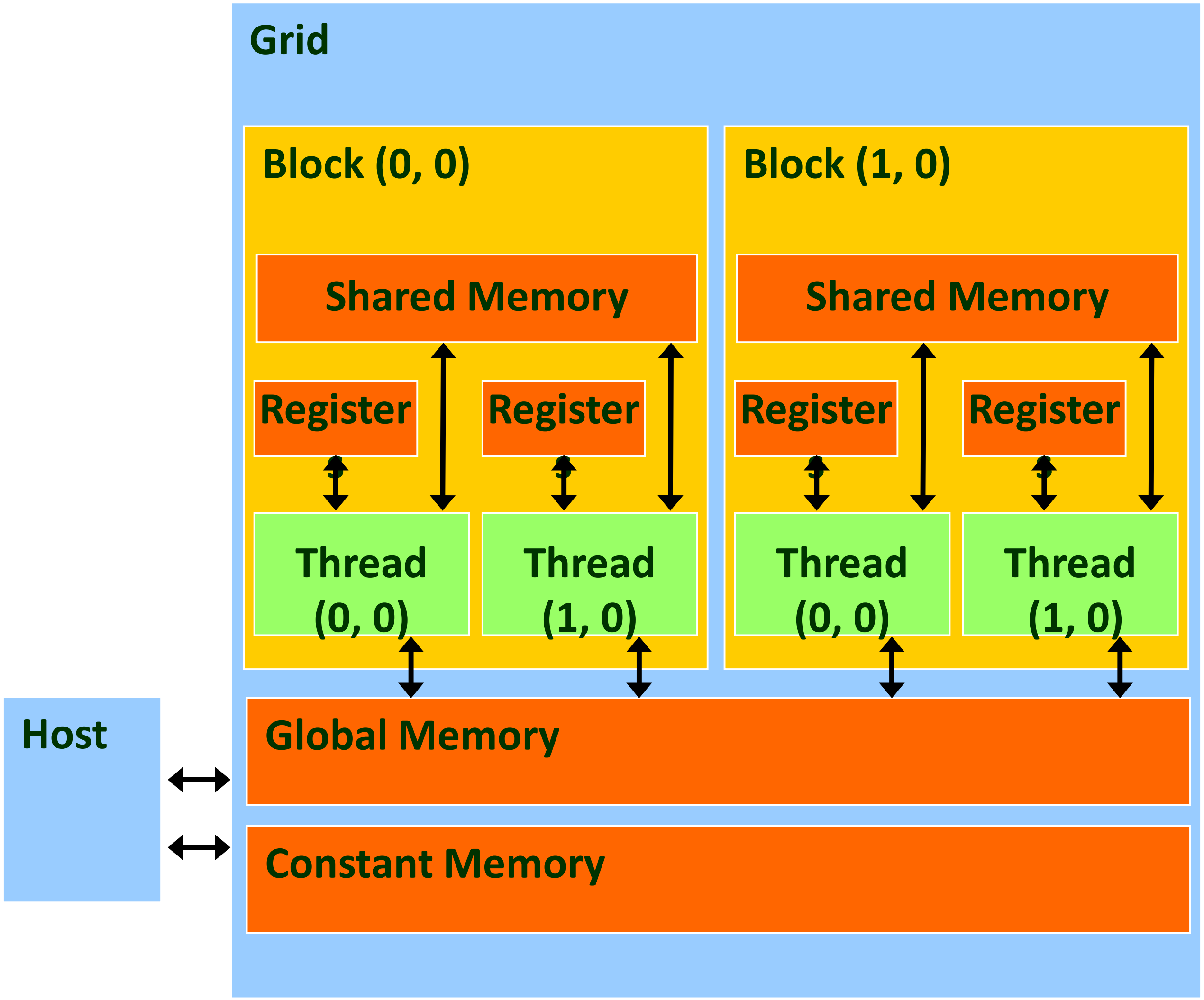
Memory and Registers in the Von-Neumann Model



CUDA Memories in a Similar Model



Programmer's View of CUDA Memories



Type Qualifiers of Device Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __device__ is optional when used with __shared__, or __constant__
- Automatic variables (variables declared without any of these qualifiers) reside in a register except per-thread arrays that reside in global memory

Global Memory

- Resides in device memory (high latency + high bandwidth)
- Accessed via 32-, 64-, or 128-byte memory transactions
  - Addresses in a transaction must be aligned to these sizes.
  - Memory accesses of the threads within a warp are coalesced into one or more of memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads.

Local Memory

- Resides in device memory
  - Same latency and bandwidth as global memory access
  - Same requirements for memory coalescing
  - Access cached same way as global memory access
- Organized such that consecutive 32-bit words are accessed by consecutive thread IDs
  - Accesses are therefore fully coalesced as long as all threads in a warp follow the access pattern

Constant Memory

- Resides in device memory
- Cached in the constant cache
- Accesses are split into separate memory requests depending on the addresses.
  - Each request is serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise.

Shared Memory

- On-chip
  - Much higher bandwidth and much lower latency than local or global memory
- Divided into equally-sized memory modules, called banks, which can be accessed simultaneously
  - If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized.

Texture and Surface Memory

- Reside in device memory
- Cached in texture cache
  - A texture fetch or surface read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache.
- The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance.

Details of CUDA Memories

Memory	Location	Cached	Access	Who	Latency
Register	On-chip	Resident	Read/write	One thread	O(1 cycle)
Shared	On-chip	Resident	Read/write	Threads in block	O(1 cycle) w/o conflict
Global	Off-chip	No/Yes	Read/write	All threads + host	O(1)- O(100) cycles, depending on if cached
Local	Off-chip	No/Yes	Read/write	One thread	O(1)- O(100) cycles, depending on if cached
Constant	Off-chip	Yes	Read only	All threads + host (host may write)	O(1)-O(100) cycles, depending on if cached
Texture	Off-chip	Yes	Read only	All threads + host (host may write)	O(1)- O(100) cycles, depending on if cached
Surface	Off-chip	Yes	Read/write	All threads+host	O(1)-O(100) cycles, depending on if cached

Targets of Memory Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in “fast memory”
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
    - Appropriate data placement and layout
    - Code reordering transformations

Data Placement: Conceptual

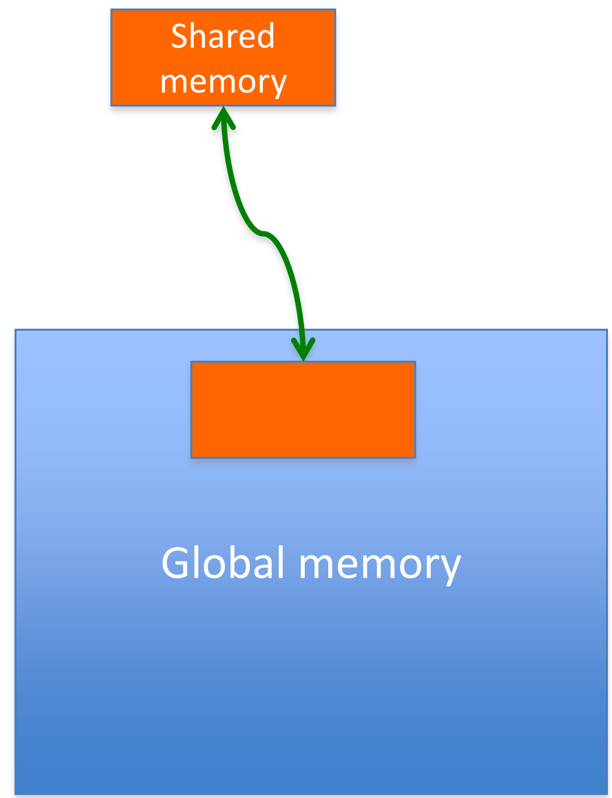
- Copies from host to device go to some part of global memory (possibly, constant or texture memory)
- How to use shared memory
  - Must construct or be copied from global memory by kernel program
- How to use constant or texture cache
  - Read-only "reused" data can be placed in constant & texture memory by host
- How to use registers
  - Most locally-allocated data is placed directly in registers
  - Even array variables can use registers if compiler understands access patterns
  - Can allocate vectors to registers, e.g., float4
  - Excessive use of registers will "spill" data to local memory

Data Placement: Syntax

- Through type qualifiers
  - __constant__, __shared__, __device__
- Through cudaMemcpy calls
  - Any directions between host and device memories
- Implicit default behavior
  - Device memory without other qualifier is global memory
  - Host by default copies to global memory
  - Thread-local variables go into registers unless capacity exceeded, then local memory

Common Programming Pattern of Using Shared Memory

- Load data into shared memory
- Synchronize (if necessary)
- Operate on data in shared memory
- Synchronize (if necessary)
- Write intermediate results to global memory
- Repeat until done



Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

```
extern __shared__ float  d_s_array[];

__host__ void outerCompute() {
  compute<<<gs,bs>>>();
}
__global__ void compute() {
   d_s_array[i] = …;
}
```

```
__global__ void compute2() {
__shared__ float d_s_array[M];

  // create or copy from global memory
  d_s_array[j] = …;
  //synchronize threads before use
  __syncthreads();
  … = d_s_array[x]; // now can use any element

 // more synchronization needed if updated
  __syncthreads();
  // may write result back to global memory
  d_g_array[j] =  d_s_array[j];
}
```

Tiling for Limited Capacity Storage

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in registers exceeds register capacity or data in shared memory for block still exceeds shared memory capacity

Summary

- Device variables reside in the global memory, the shared memory, or registers.
- CUDA memories have different latency and bandwidth characteristics.
- Memory optimizations can be done through data placement and reuse.
- Tiling for the shared memory is a common memory optimization in CUDA programming.

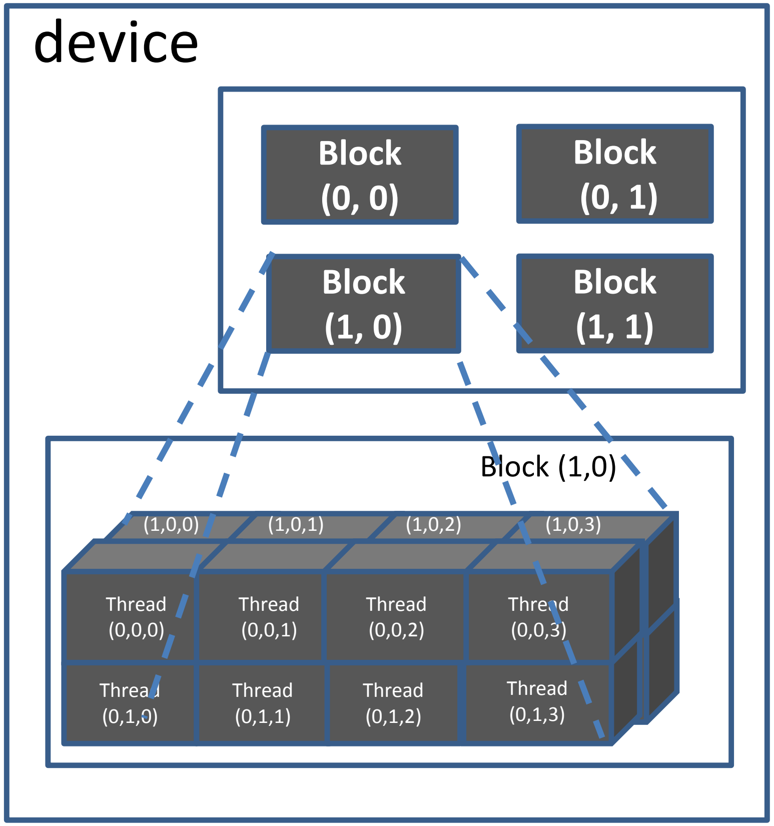
Parallel Programming

CUDA Threads

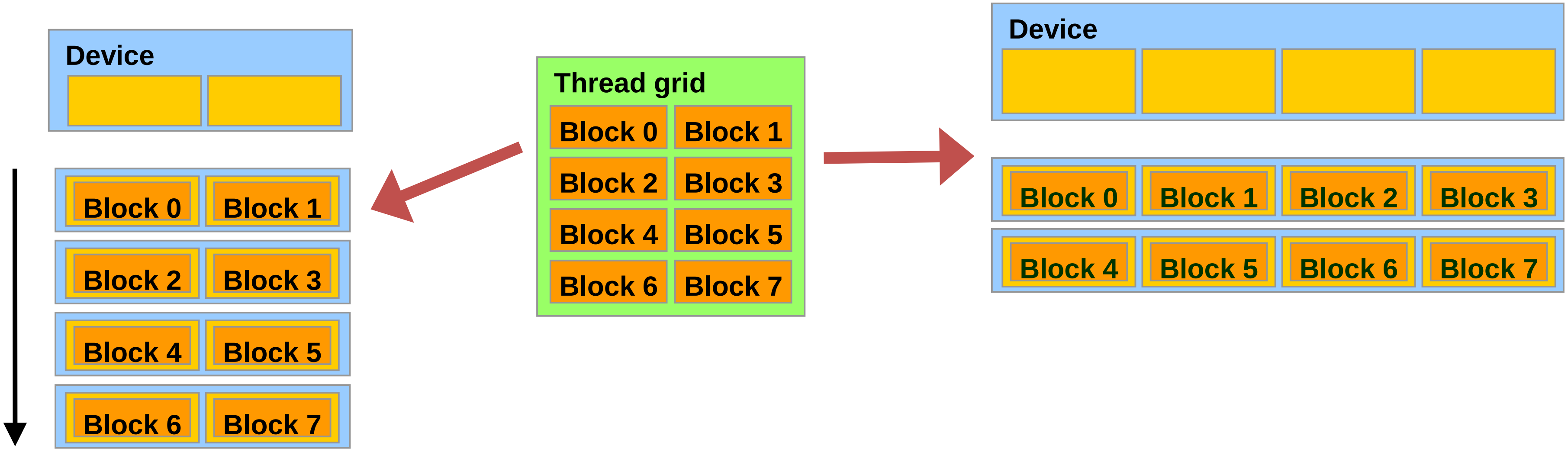
Overview

- Thread Mapping
- Warp Scheduling
- Control Divergence

A Multi-Dimensional Grid Example



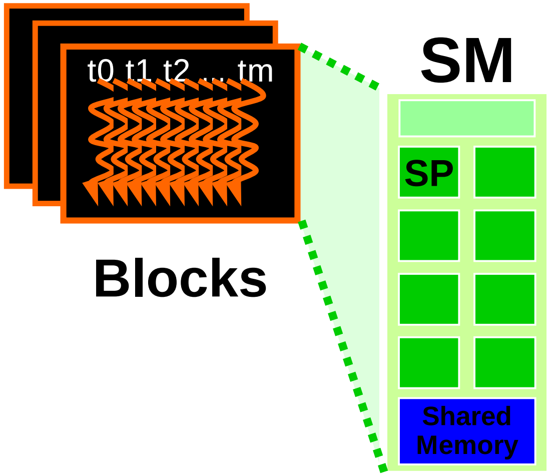
Transparent Scalability



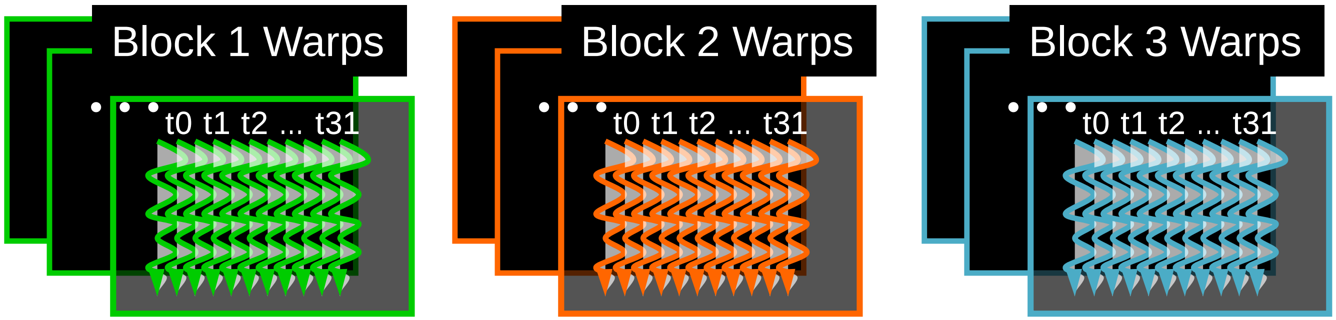
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to **8** blocks to each SM as resource allows
  - Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



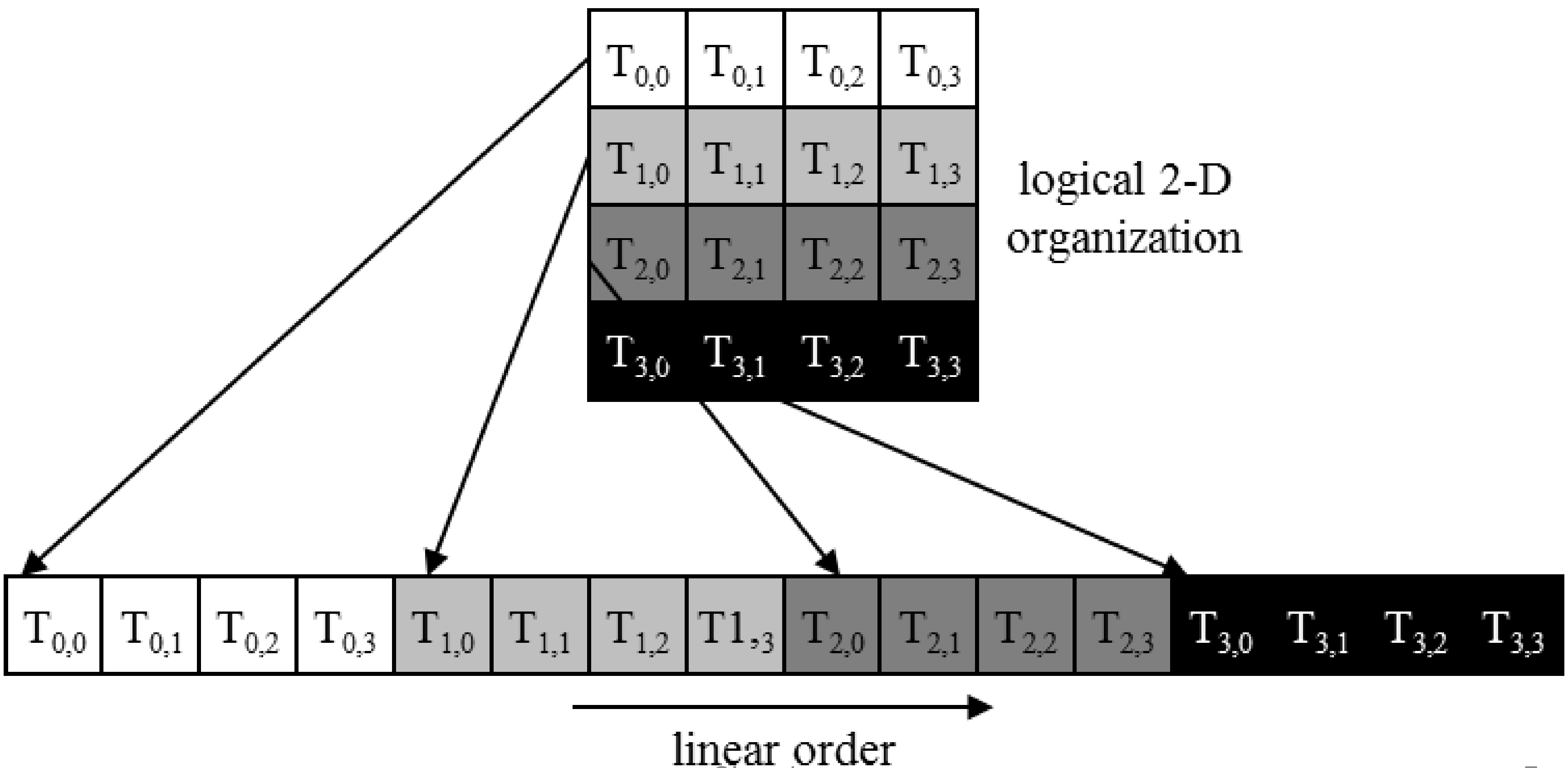
Warps as Scheduling Units



- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order: x followed by y followed by z

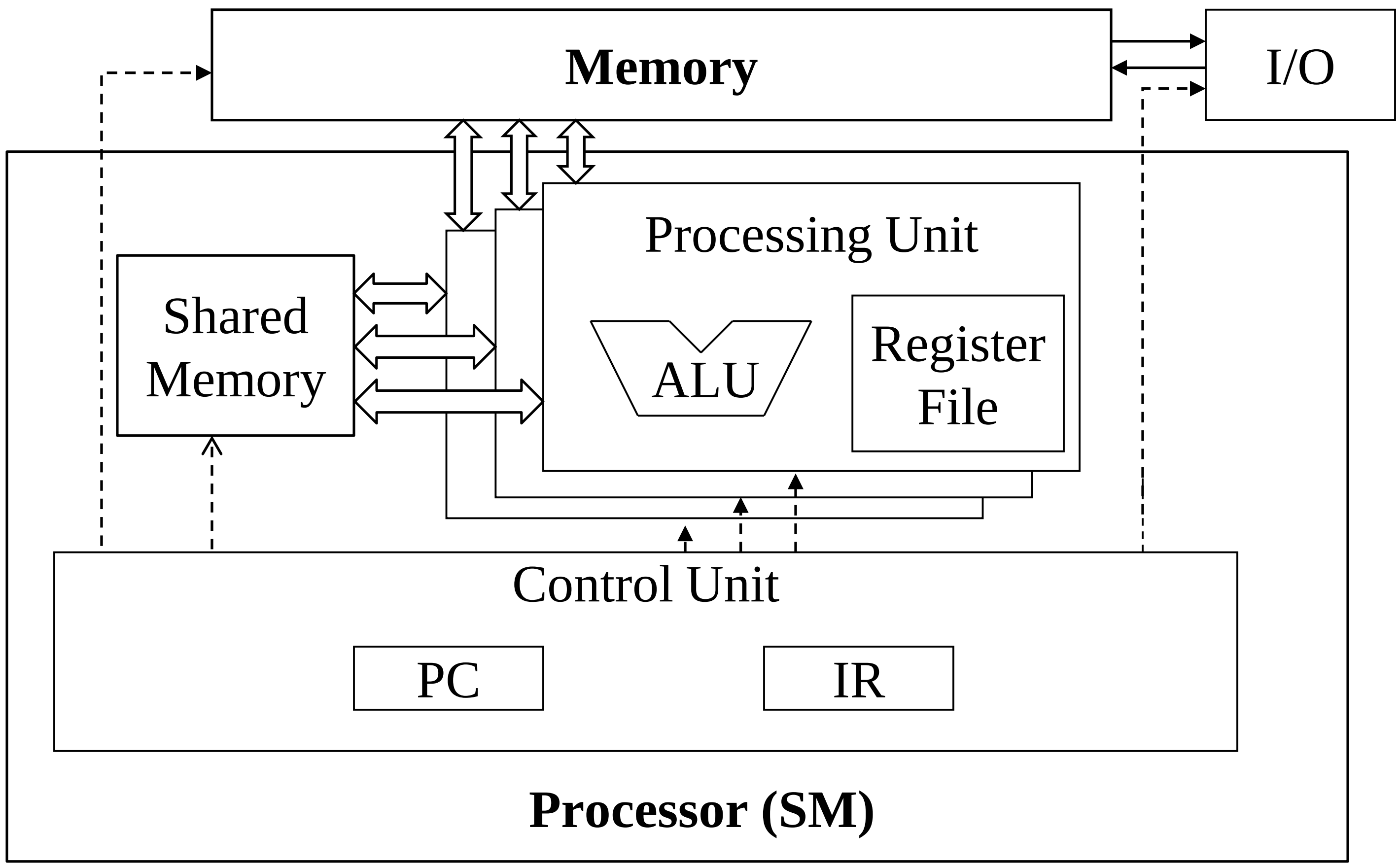


Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results.

SMs are SIMD Processors

Control unit for is shared among processing units



SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time

- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times

Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others

- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
  - if (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

Example: Vector Addition Kernel

```
//  Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C,
  int n)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Analysis for vector size of 1,000 elements

- Assume that block size is 256 threads
  - 8 warps in each block
- All threads in Blocks 0, 1, and 2 are within valid range
  - i values from 0 to 767
  - There are 24 warps in these three blocks, none will have control divergence
- Most warps in Block 3 will not control divergence
  - Threads in the warps 0-6 are all within valid range, thus no control divergence
- One warp in Block 3 will have control divergence
  - Threads with i values 992-999 will all be within valid range
  - Threads with i values of 1000-1023 will be outside valid range
- Effect of serialization on control divergence will be small
  - 1 out of 32 warps has control divergence
  - The impact on performance will likely be less than 3%

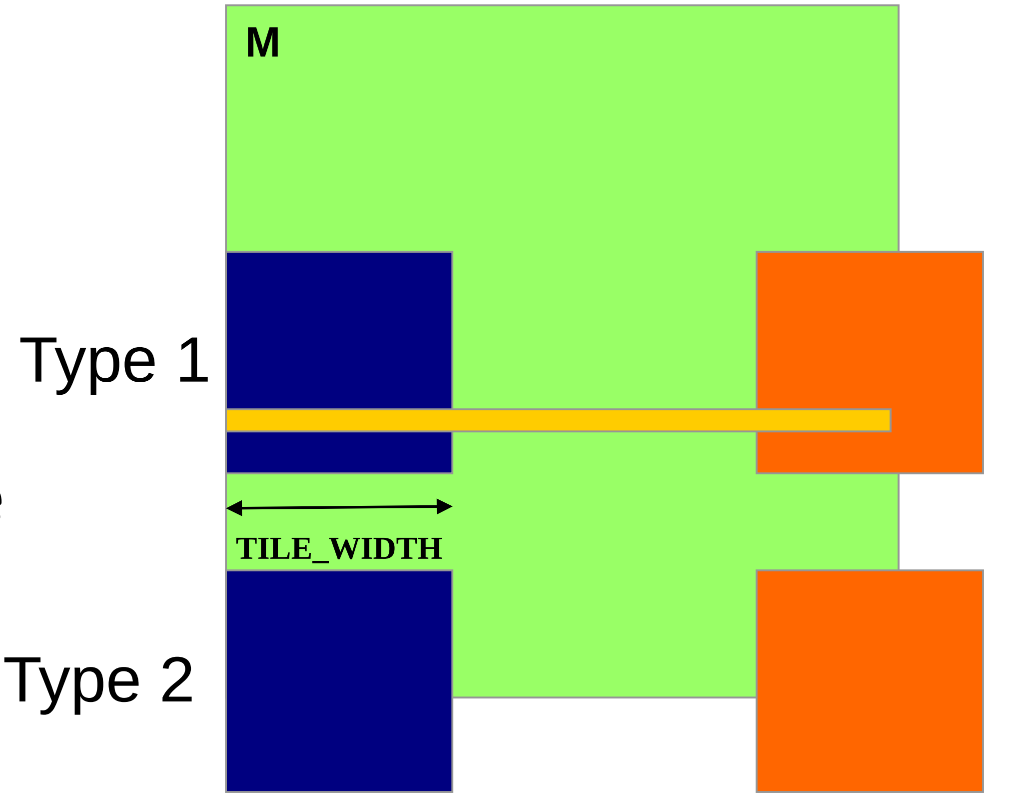
Performance Impact of Control Divergence

- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The tiled matrix multiplication kernel has many boundary condition checks
  - The concern is that these checks may cause significant performance degradation

```
if(Row < Width && t * TILE_WIDTH+tx < Width) {
  ds_M[ty][tx] = M[Row * Width + t * TILE_WIDTH + tx];
} else {
 ds_M[ty][tx] = 0.0;
}

if (t*TILE_WIDTH+ty < Width && Col < Width) {
  ds_N[ty][tx] = N[(t*TILE_WIDTH + ty) * Width + Col];
} else {
  ds_N[ty][tx] = 0.0;
}
```

Two types of blocks in loading M Tiles

- 1. Blocks whose tiles are all within valid range until the last phase.
- 2. Blocks whose tiles are partially outside the valid range all the way



Analysis of Control Divergence Impact

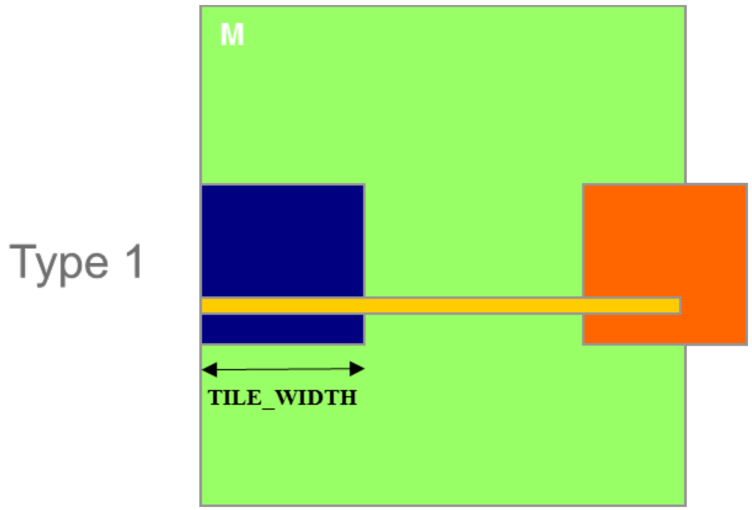
- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each thread will go through 7 phases (ceiling of 100/16)

- There are 49 thread blocks (7 in each dimension)

Control Divergence in Loading M Tiles

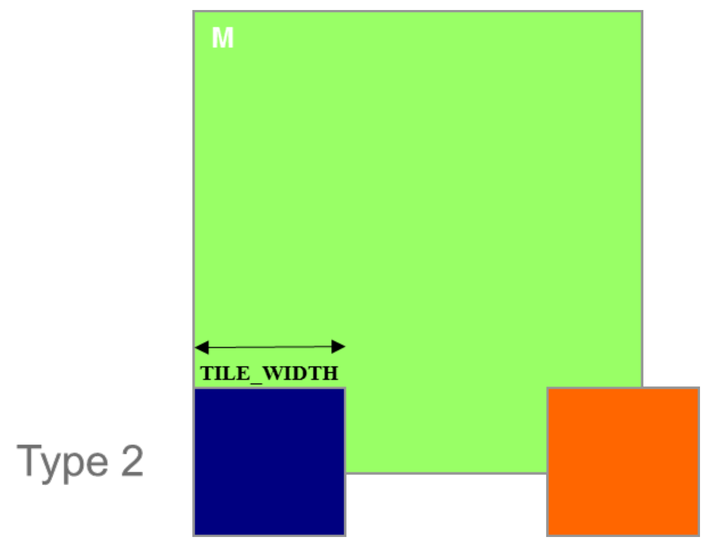
- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)

- There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps
- They all have 7 phases, so there are 2,352 (336*7) warp-phases
- The warps have control divergence only in their last phase
- 336 warp-phases have control divergence



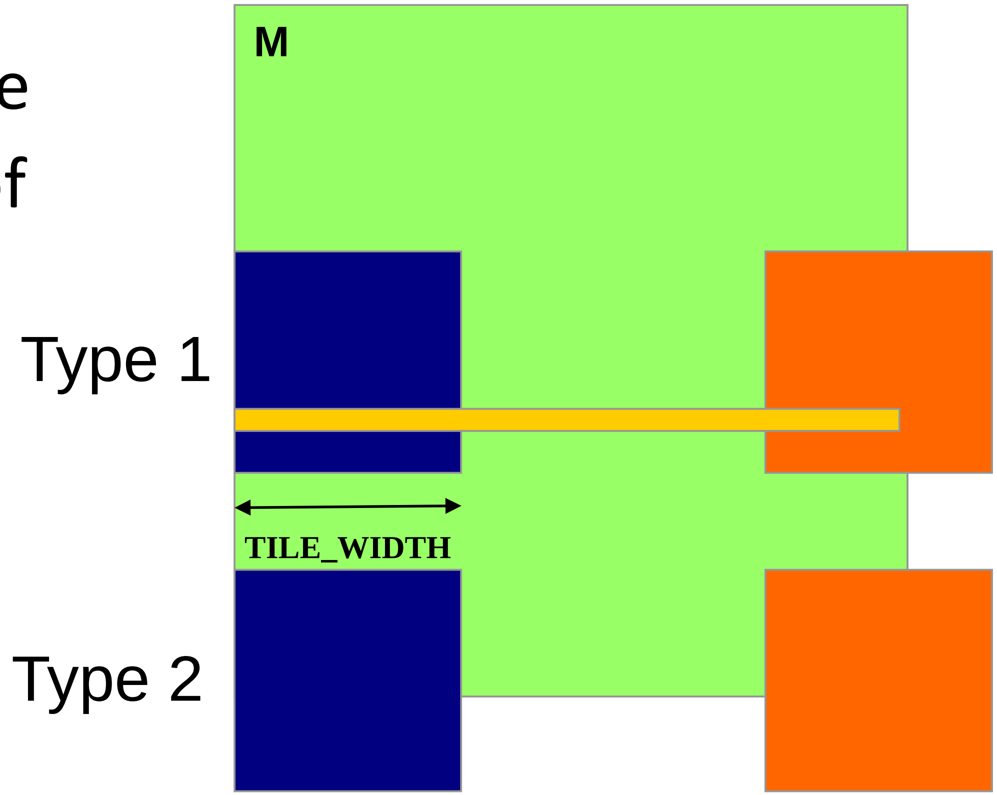
Control Divergence in Loading M Tiles (Type 2)

- Type 2: the 7 blocks assigned to load the bottom tiles, with a total of 56 (8*7) warps
- They all have 7 phases, so there are 392 (56*7) warp-phases
- The first 2 warps in each Type 2 block will stay within the valid range until the last phase
- The 6 remaining warps stay outside the valid range
- So, only 14 (2*7) warp-phases have control divergence



Overall Impact of Control Divergence

- Type 1 Blocks: 336 out of 2,352 warp-phases have control divergence
- Type 2 Blocks: 14 out of 392 warp-phases have control divergence
- The performance impact is expected to be less than 12% (350/2,944 or (336+14)/(2352+14))



Additional Comments

- The estimated performance impact is data dependent.
  - For larger matrices, the impact will be significantly smaller
- In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant
  - One should not hesitate to use boundary checks to ensure full functionality
- The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence

Summary

- Threads are transparently mapped to processors.
- Threads are scheduled in the unit of warps
- Branch code does not necessarily cause control divergence.
- Control divergence is data dependent.

Parallel Programming

Data-Parallel Primitives:

Gather and Scatter

`

Overview

- Data-Parallel Primitives
  - Map, Prefix Scan, Scatter, Gather, Split, Sort
  - Others: Reduce, Filter, Search…
- Optimizations on the GPU

Processing Large Data Sets

```
//sequential
   for (i = 0; i < N; i++)
            h_C[i] = h_A[i] + h_B[i];

//data-parallel
__global__ void VecAdd(int* A, int* B, int* C)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   C[i] = A[i] + B[i];
}
```

Map and Prefix Scan

Primitive: Map
Input: R_{in} [1, …, *n*], a map function *fcn*.
Output: $R_{out}[1,\ldots,n]$.
Function: $R_{out}[i]=fcn(R_{in}[i])$.

Primitive: Prefix Scan
Input: R_{in} [1, …, *n*], binary operator $\oplus$.
Output: $R_{out}[1,\ldots,n]$.
Function: $R_{out}[i]= \bigoplus_{j<i} R_{in}[j]$.

Scatter and Gather

Primitive: Scatter
Input: $R_{in}[1, \ldots, n]$, $L[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $R_{out}[L[i]]=R_{in}[i]$, $i=1, \ldots, n$.

Primitive: Gather
Input: $R_{in}[1, \ldots, n]$, $L[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $R_{out}[i]=R_{in}[L[i]]$, $i=1, \ldots, n$.

Split and Sort

Primitive: Split
Input: $R_{in}[1, \ldots, n]$, $func(R_{in}[i]) \in [1,\ldots,F]$, i=1, …, n.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $func(R_{out}[i]) \le func(R_{out}[j]), \forall i,j \in [1,..,n], i \le j$.

Primitive: Sort
Input: $R_{in}[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $R_{out}[i] \le R_{out}[j], \forall i,j \in [1,..,n]$ and $i \le j$.

Map Example

```
// for all samples – all threads execute this code
neighbors[x][y] =
0.25f * (value[x-1][y]+
value[x+1][y]+
value[x][y+1]+
value[x][y-1]);
diff = (value[x][y] - neighbors[x][y]);
diff *= diff; // squared difference
```

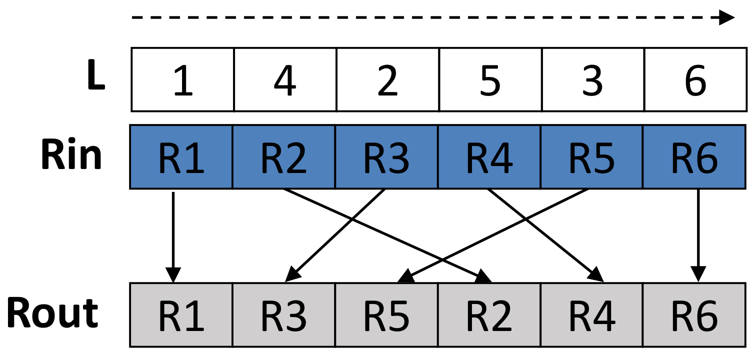
- Load from GPU memory, compute, store to GPU memory
- Make computation as dense as possible to amortize memory access cost
- Maximize number of concurrent threads

Scatter and Gather: Overview

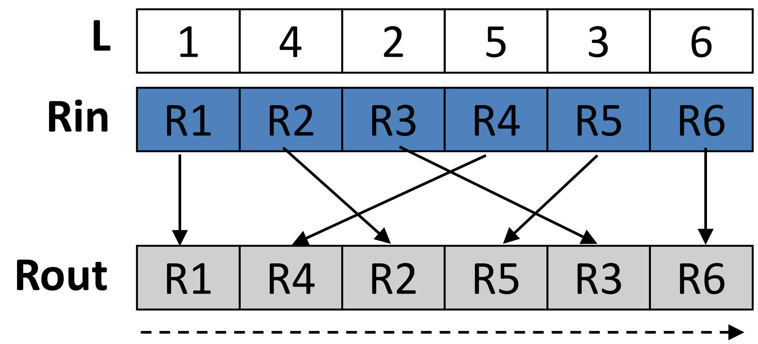
- Widely supported
  - Parallel programming languages, e.g., MPI, NESL, ZPL.
  - Supercomputers, e.g., Cray MTA, Stanford Merrimac
  - Commodity co-processors (IBM Cell, GPUs)
- Irregular access patterns
  - Sparse matrix computations, hashing, searching, etc.
- Performance is memory bandwidth limited
  - Require high bandwidth architectures
  - HPC benchmarks (HPC Challenge, NAS PB, etc.)

Access Patterns

- Scatter: sequential reads and random writes.
- Gather: random reads and sequential writes.



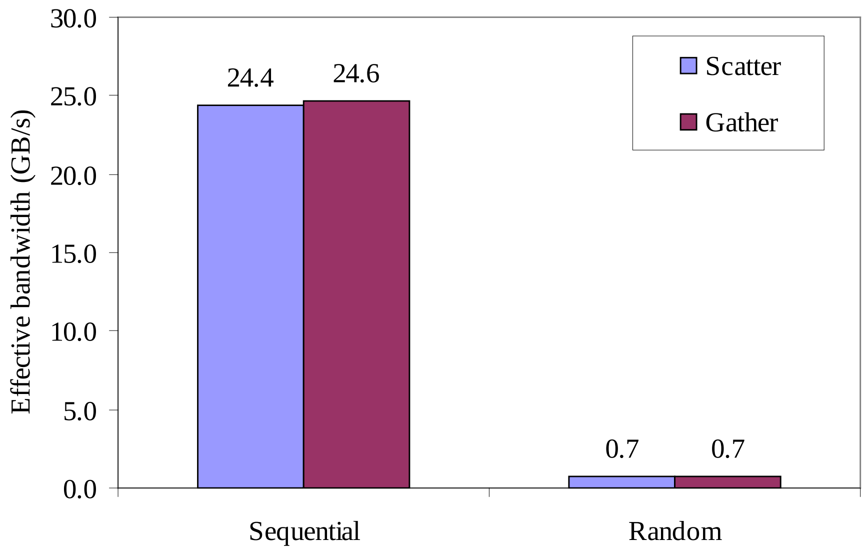
(a) Scatter



(b) Gather

Scatter and Gather on the GPU

- Access pattern makes a 30X difference in performance [Supercomputing 2007].



Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

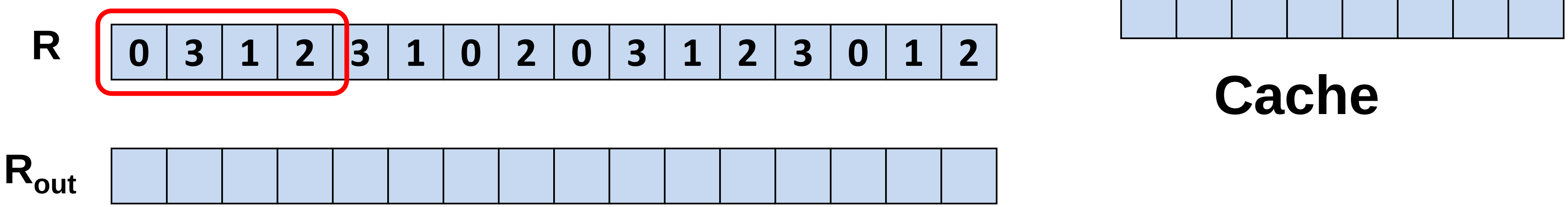
4 mem. blocks to write
4 concurrent threads
2 cache lines

Cache

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

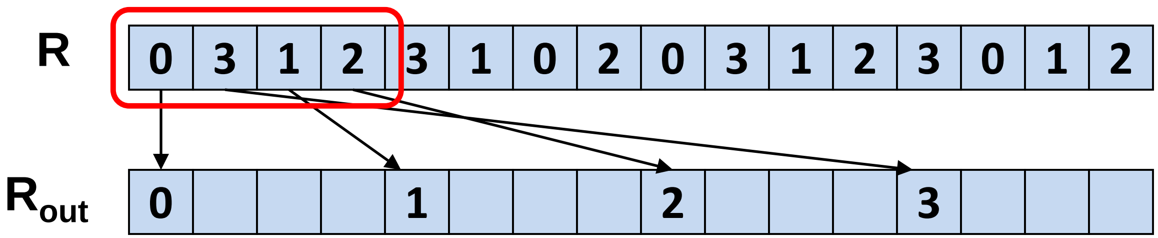
4 mem. blocks to write
4 concurrent threads
2 cache lines

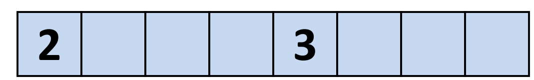


Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines





Cache

Cache Misses = 4
Cache Hits = 0

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0				1				2				3			

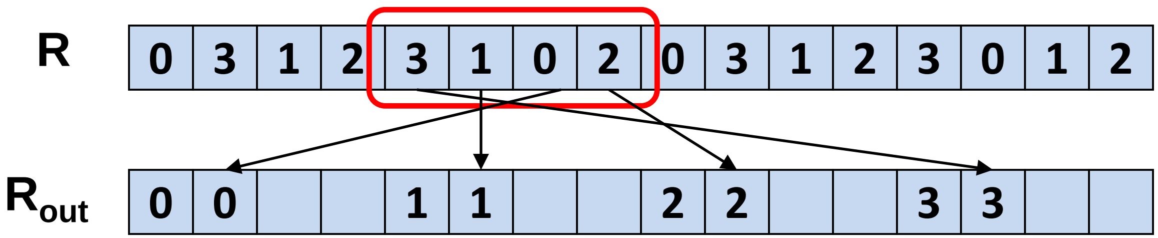
2				3			

Cache

Cache Misses = 4
Cache Hits = 0

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11



4 mem. blocks to write
4 concurrent threads
2 cache lines

0	0			1	1		

Cache

Cache Misses = 6
Cache Hits = 2

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

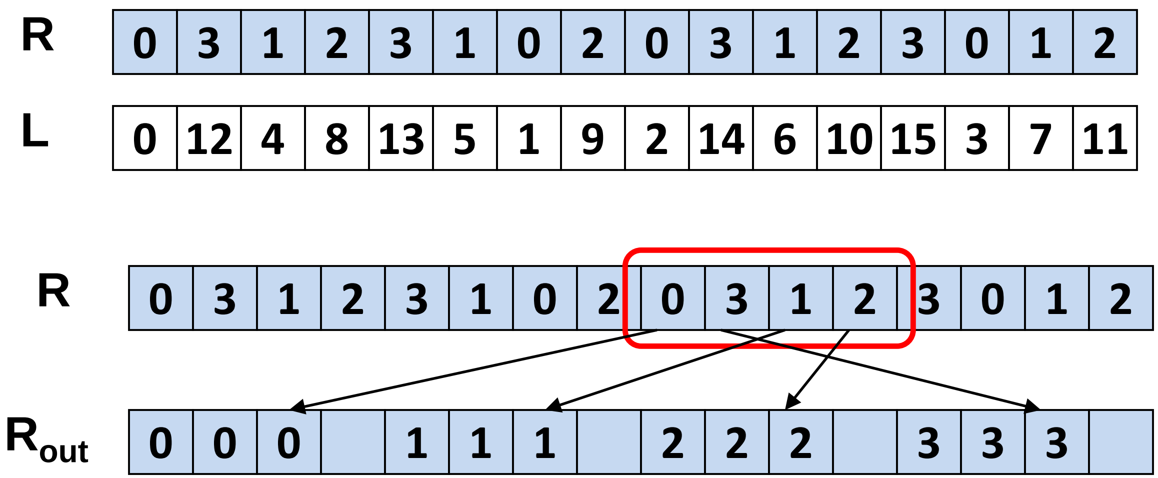
0	0			1	1		

Cache

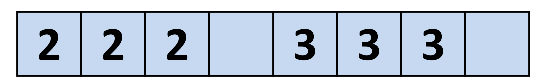
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0			1	1			2	2			3	3		

Cache Misses = 6
Cache Hits = 2

Example: Single-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 8
Cache Hits = 4

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

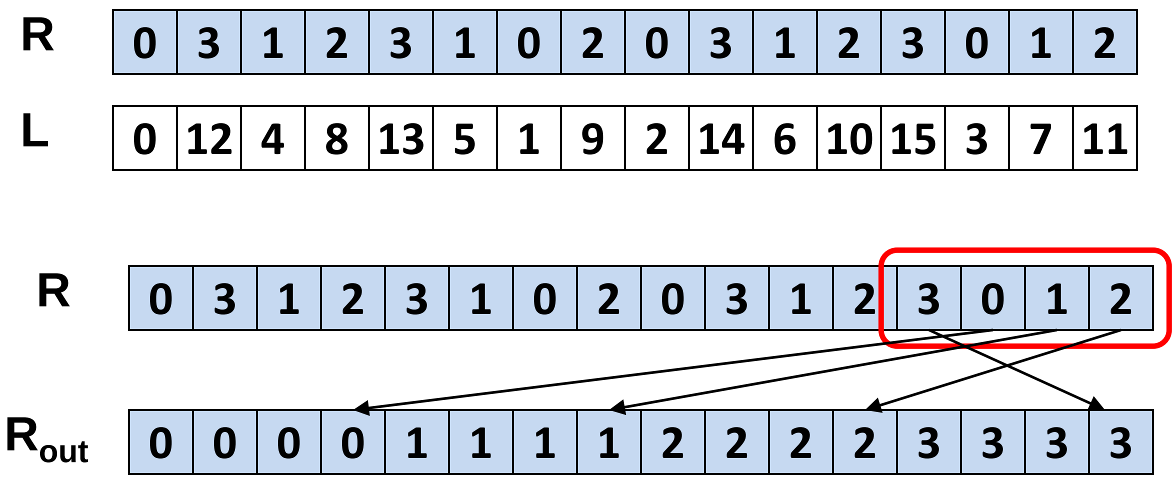
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0	0		1	1	1		2	2	2		3	3	3	

2	2	2		3	3	3	

Cache

Cache Misses = 8
Cache Hits = 4

Example: Single-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines

0	0	0	0	1	1	1	1

Cache

Cache Misses = 10
Cache Hits = 6

Cache miss rate = 62.5%
Effective write bandwidth = |R|/Transfer Time = 4/10* B_{seq}= 0.4 B_{seq}

Multi-pass Scheme

- The entire scatter is performed in multiple passes.
- Each pass writes to a small chunk

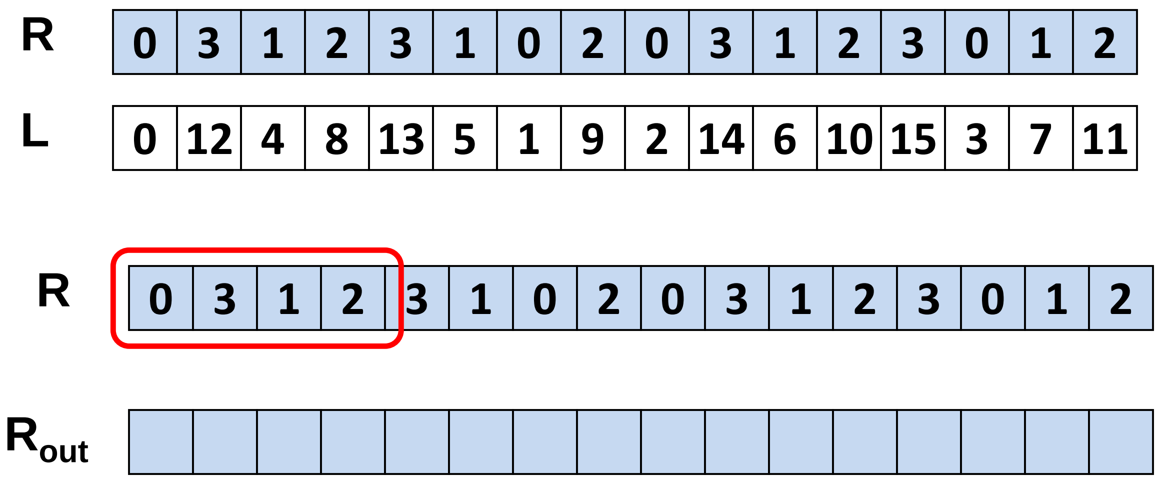
Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

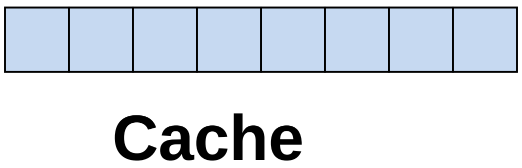
4 mem. blocks to write
4 concurrent threads
2 cache lines

Cache

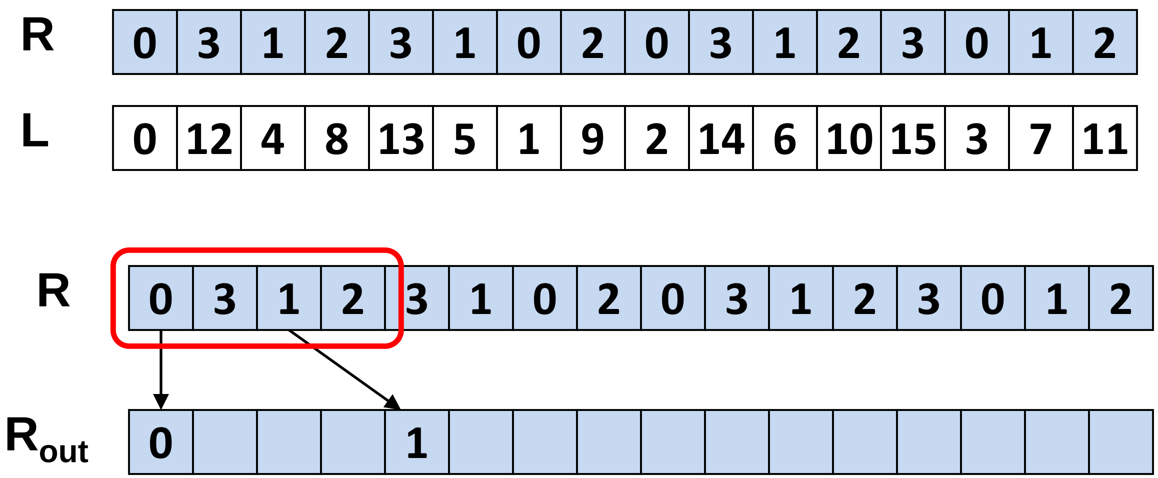
Two-pass Scatter



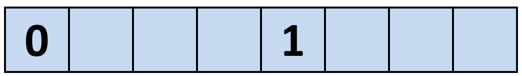
4 mem. blocks to write
4 concurrent threads
2 cache lines



Two-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 2
Cache Hits = 0

Two-pass Scatter

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

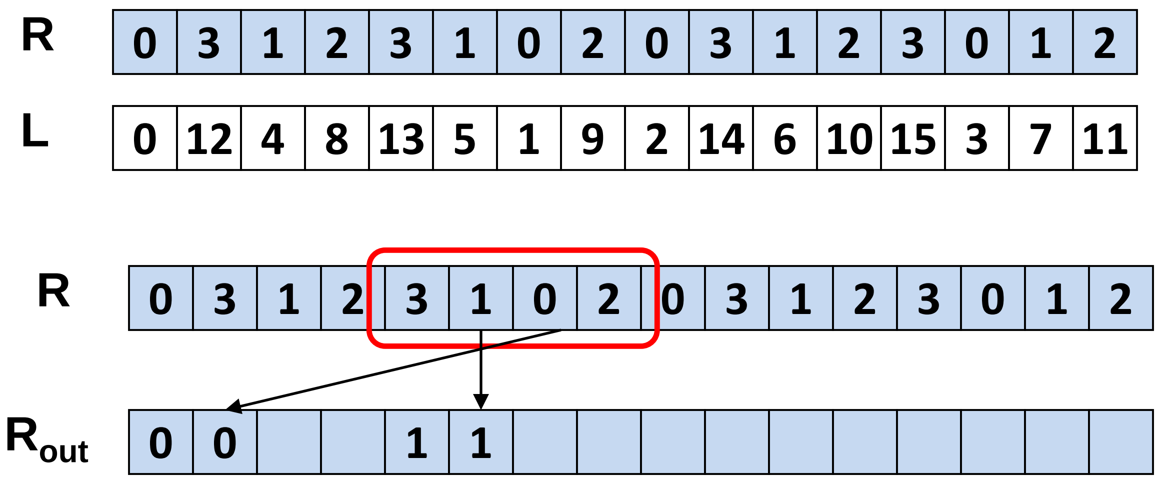
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0				1											

0				1			

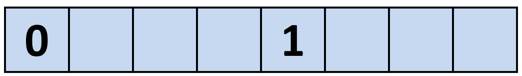
Cache

Cache Misses = 2
Cache Hits = 0

Two-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 2
Cache Hits = 2

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

R	0	3	1	2	3	1	0	2	$\boxed{0}$	$\boxed{3}$	$\boxed{1}$	$\boxed{2}$	3	0	1	2
R_{out}	0	0			1	1										

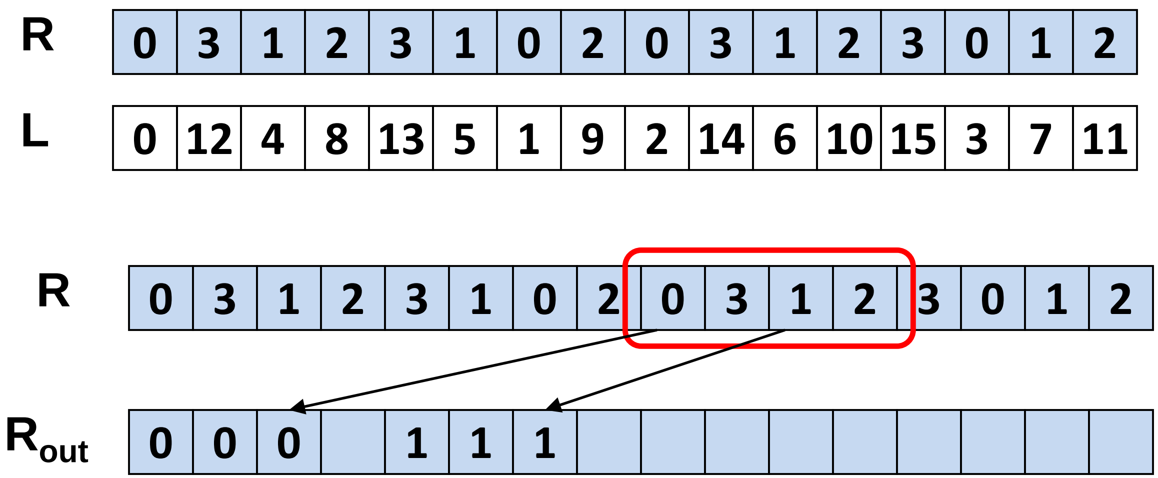
4 mem. blocks to write
4 concurrent threads
2 cache lines

0				1			

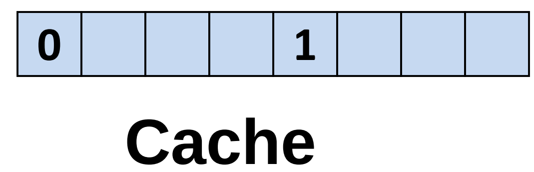
Cache

Cache Misses = 2
Cache Hits = 2

Two-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache Misses = 2
Cache Hits = 4

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

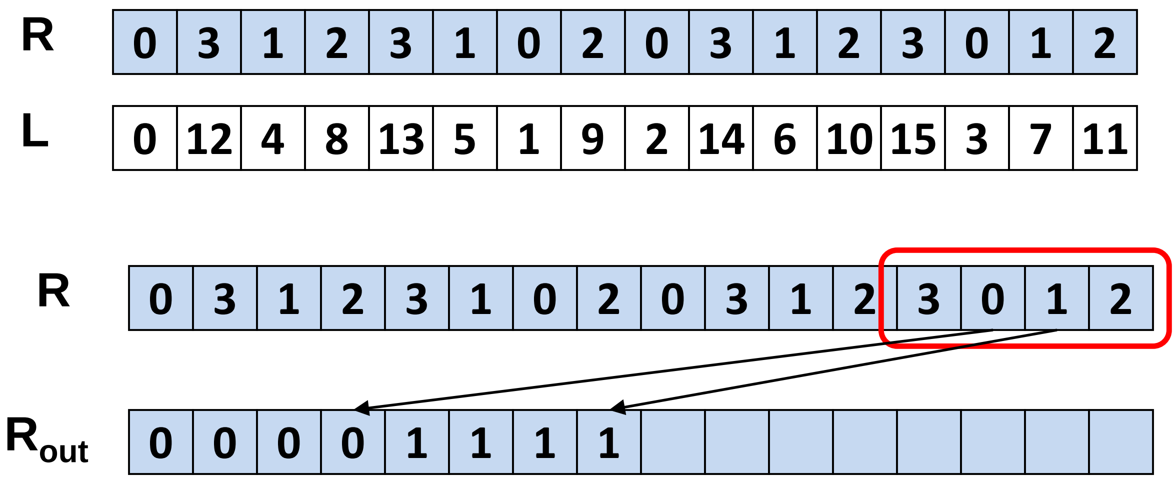
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0	0		1	1	1									

0				1			

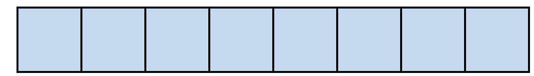
Cache

Cache Misses = 2
Cache Hits = 4

Two-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 2
Cache Hits = 6

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	$\boxed{3}$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$
R_{out}	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

Cache

Cache Misses = 4
Cache Hits = 12

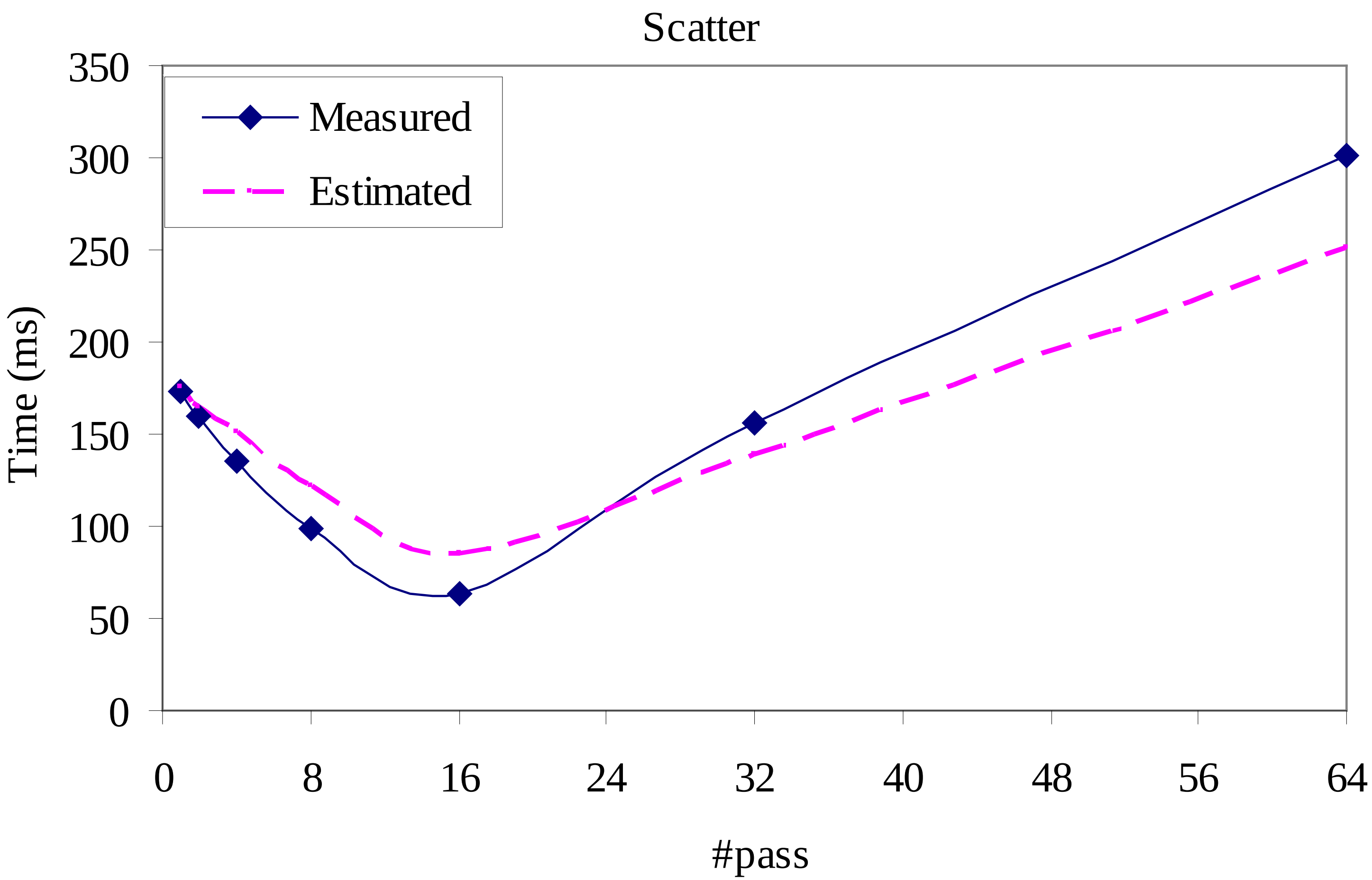
Cache miss rate = 25%
Effective write bandwidth = B_{seq}

Cost Model

- Estimate the performance of different access patterns
  - Sequential bandwidth
  - Random bandwidths of different degrees
- Estimate the total cost of sequential access and random access in the multi-pass scheme.

  $T_{scatter}$= (|R| + |L|) * npasses/B_{seq}+ |R|/B_{rand}
- Determine the optimal number of passes.

Performance Results
-- Multi-pass Scatter

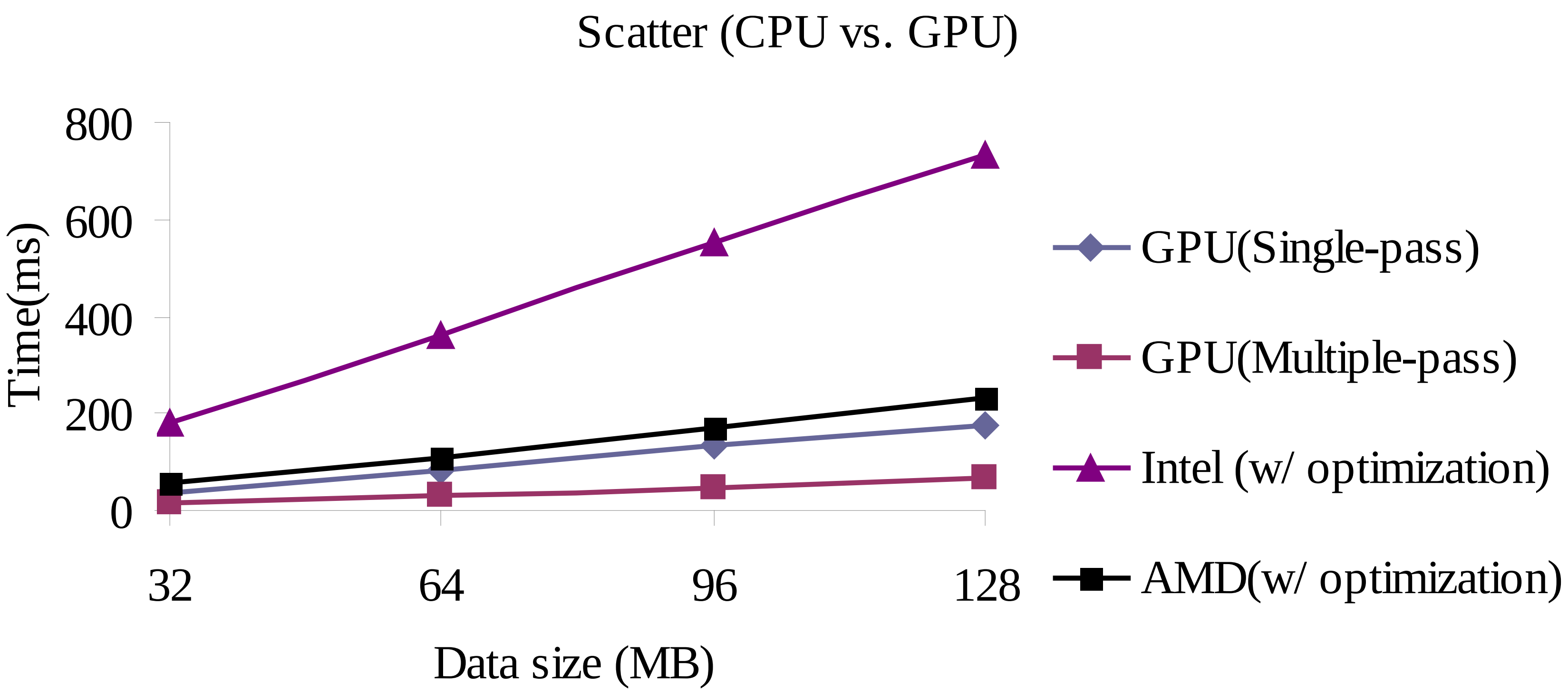


The optimal number of passes is 16.

Applications and Analysis

- Applications
  - Radix sort, hash search, and sparse-matrix vector multiplication
- Platforms
  - CPUs: Intel Quad, or two AMD dual-core processors.
  - GPU: Nvidia 8800 GTX.
- Overall results
  - The cost model has an accuracy of over 85%.
  - The multipass scheme improves the application 10%~50%.
  - The GPU-based algorithm outperforms the CPU-based algorithm by 2-7X.

Performance Impact of Multi-Pass Scatter



(1) The speedup is 7-13X and 2-4X on Intel and AMD, respectively.
(2) The multi-pass scheme improves the GPU-based scatter by 2-4X.

Summary

- Data-parallel primitives are an effective way of utilizing GPU's parallelism.
- Scatter and gather are memory-bound and can be optimized through multi-pass schemes.

References:

Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. ACM/IEEE SuperComputing (SC), Nov 2007.

http://www.cse.ust.hk/gpuqp

Parallel Programming

Data-Parallel Primitives:
Reduction

`

Overview

- The Reduction Operation
- Sequential Implementation
- Baseline Reduction Kernel
- Improved Reduction Kernel

Reduce (Reduction)

- A commonly used strategy for processing large input data sets
- There is no required order of processing elements in a data set (associative and commutative)
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- Google and Hadoop MapReduce frameworks support this strategy

Reduction in Other Parallel Operations

- Reduction is also needed to clean up after some commonly used transformations
- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location

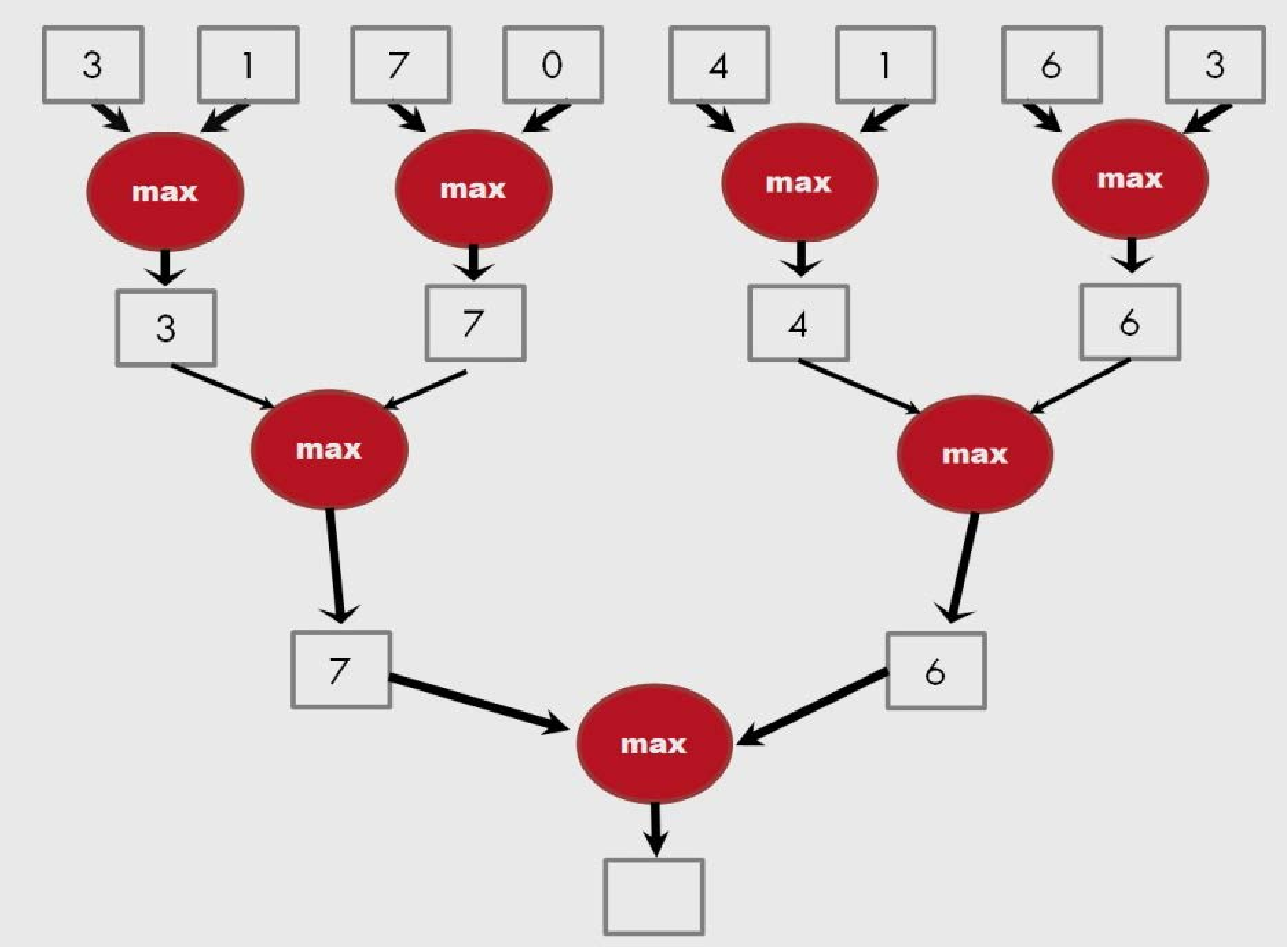
Computation used in Reduction

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
  - User defined reduction operation function as long as the operation
    - Is associative and commutative
    - Has a well-defined identity value (e.g., 0 for sum)

Sequential Reduction

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- Iterate through the input and perform the reduction operation between the result value and the current input value
  - N reduction operations performed for N input values

A Reduction Tree



Analysis of Reduction Tree

- For N input values, the reduction tree performs

(1/2)N + (1/4)N + (1/8)N + … 1 = (1- (1/N))N = N-1 operations

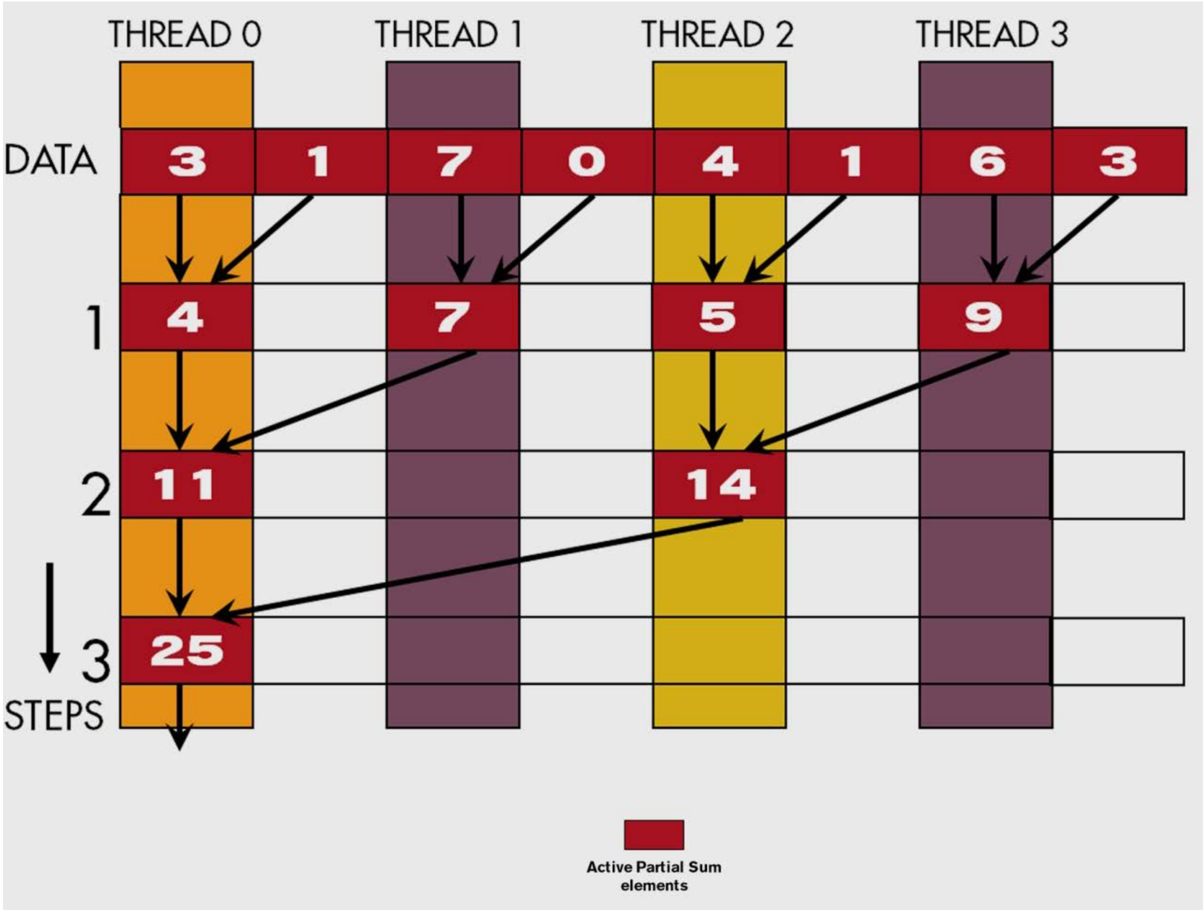
- In Log (N) steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
- Average Parallelism (N-1)/Log(N))
  - For N = 1,000,000, average parallelism is 50,000
  - However, peak resource requirement is 500,000!
  - This is not resource efficient.
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to sequential
  - Many parallel algorithms are not work efficient

Parallel Implementation

- Parallel execution of reduction tree
  - Add two values per thread in each step
  - Halve # of threads for next step
  - Takes log(n) steps for n elements
  - Requires n/2 threads at most in a step
- In-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0
  - Reduces global memory traffic due to partial sum values

n<=2048 for current GPU due to limit of number of threads per SM

Example of Parallel Reduction



Baseline Thread-to-Data Mapping

- Each thread is responsible for an even-index location of the partial sum vector
  - In each step, one of the input is always from the location of responsibility
  - The other input comes from an increasing distance away
- After each step, half of the threads are no longer needed

Simple Thread Block Design

- Each thread block takes 2* BlockDim.x input elements
- Each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];
unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim.x+t] = input[start + blockDim.x+t];
```

Reduction

```
for (unsigned int stride = 1; stride <= blockDim.x;
stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t]+=partialSum[2*t+stride];
}
```

Synchronization Barrier

- __syncthreads() is needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

Finishing Up Reduction

- At the end of the kernel, Thread 0 in each thread block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x
- There can be a large number of such sums if the original input array for reduction is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together.

Problems in the Simple Reduction Kernel

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources

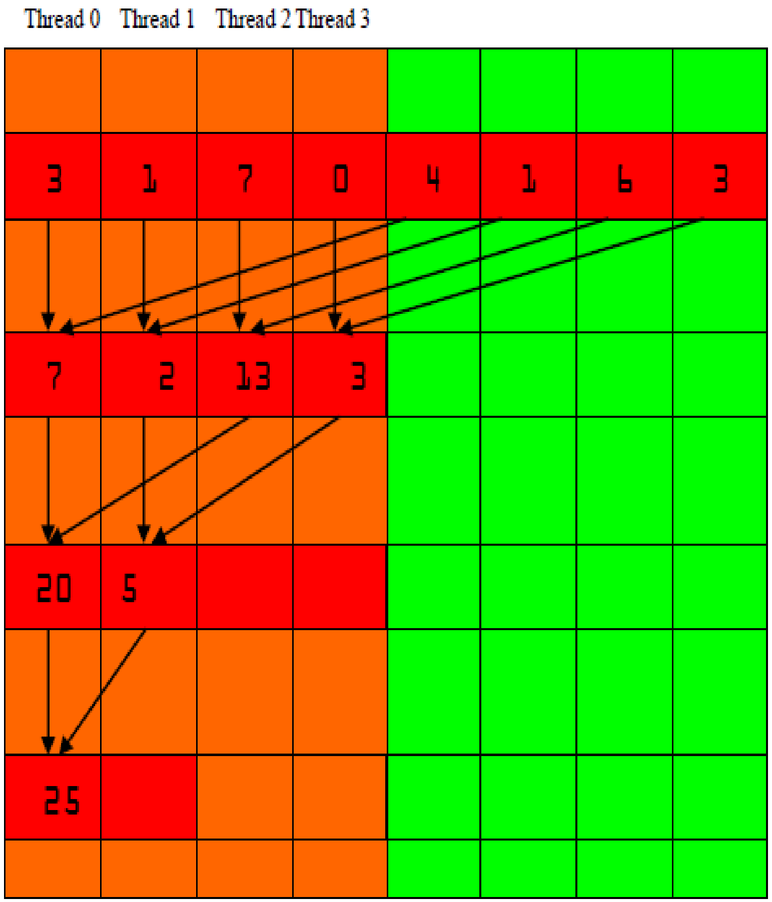
Problems in the Simple Reduction Kernel

- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence.
  - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- Always compact the partial sums into the front locations in the partialSum[] array
- Keep the active threads consecutive

An Example of Four Threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
{
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
}
```

Analysis on the Better Kernel

- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

Summary

- Reduction or reduce is also a data-parallel primitive
- Sequential implementation is of O(n) time complexity
- Parallel reduction tree algorithm is work efficient
- Thread index mapping improves reduction kernel performance

Parallel Programming

Data-Parallel Primitives:

Prefix Scan (Prefix Sum)

Overview

- Prefix Scan the Primitive
- Sequential implementation
- Work-Inefficient parallel implementation
- Work-efficient parallel implementation

Prefix Scan

- Frequently used for parallel work assignment and resource allocation, e.g., allocating memory to parallel threads or for communication channels
- A key primitive in many parallel algorithms to convert serial computation into parallel computation
- A foundational parallel computation pattern

Definition of Prefix Scan

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22 25].

Example of Prefix Scan

- Assume that we have a 100-inch sausage to feed 10 people
- We know how much each person wants in inches:

[3 5 2 7 28 4 3 0 8 1]

- How do we cut the sausage quickly?
- How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.

- Method 2: calculate prefix sum and cut in parallel:

[3, 8, 10, 17, 45, 49, 52, 52, 60, 61]

(39 inches left)

Building Block for Parallel Algorithms

- Scan is a simple and useful parallel building block
- Sequential

```
for(j=1;j<n;j++) out[j] = out[j-1] + f(j);
```

- Parallel:

```
forall(j) { temp[j] = f(j) }; scan(out, temp);
```

- Useful for many parallel algorithms:
  - Radix sort, Quicksort, String comparison, Lexical analysis
  - Stream compaction, Polynomial evaluation
  - Solving recurrences, Tree operations
  - Histograms, ….

Inclusive Sequential Addition

- Given a sequence [x0, x1, x2, ...],
- Calculate output [y0, y1, y2,...]
- Such that
  - y0 = x0
  - y1 = x0 + x1
  - y2 = x0 + x1+ x2
  - …
- Recursive definition: $y_i = y_{i-1} + x_i$

A Work-Efficient C Implementation

```
y[0] = x[0];
for (i = 1; i < Max_i; i++) y[i] = y [i-1] + x[i];
```

- Computationally efficient:
- N additions needed for N elements - O(N)

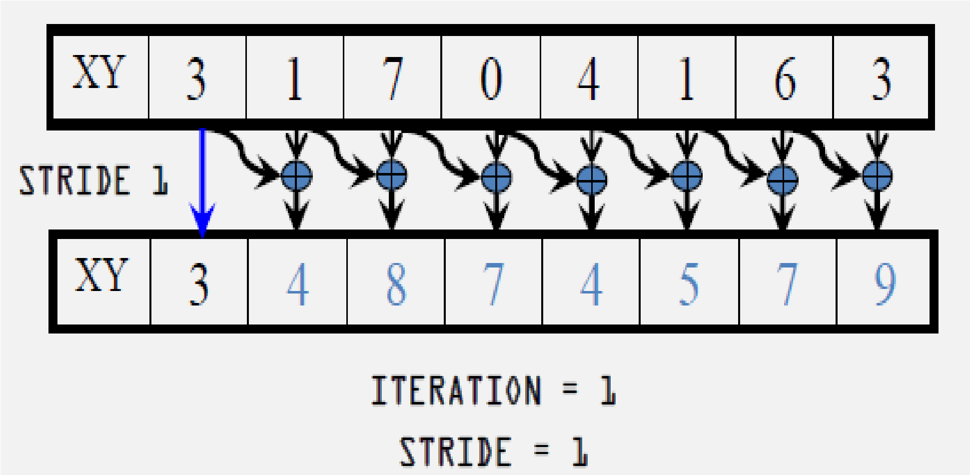
A Simple Parallel Algorithm

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element
  - y0 = x0
  - y1 = x0 + x1
  - y2 = x0 + x1+ x2
  - …

A Better Parallel Algorithm

1. Read input from device global memory to shared memory

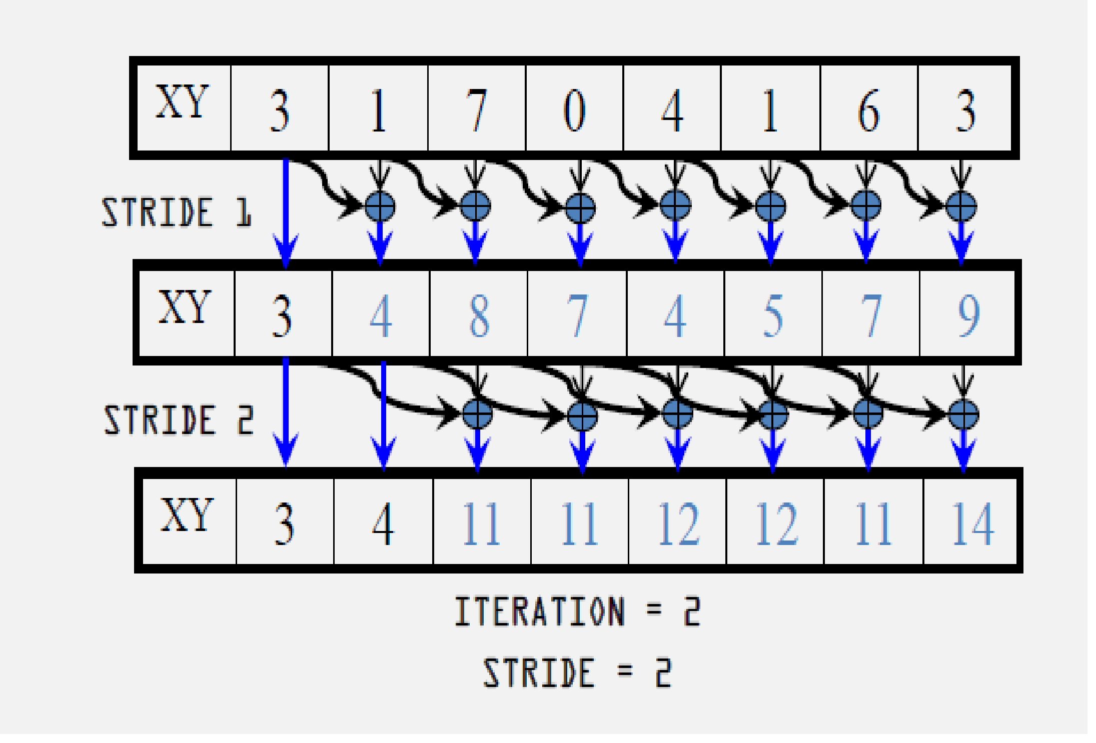
2. Iterate log(n) times; *stride* from 1 to n-1: double *stride* each iteration



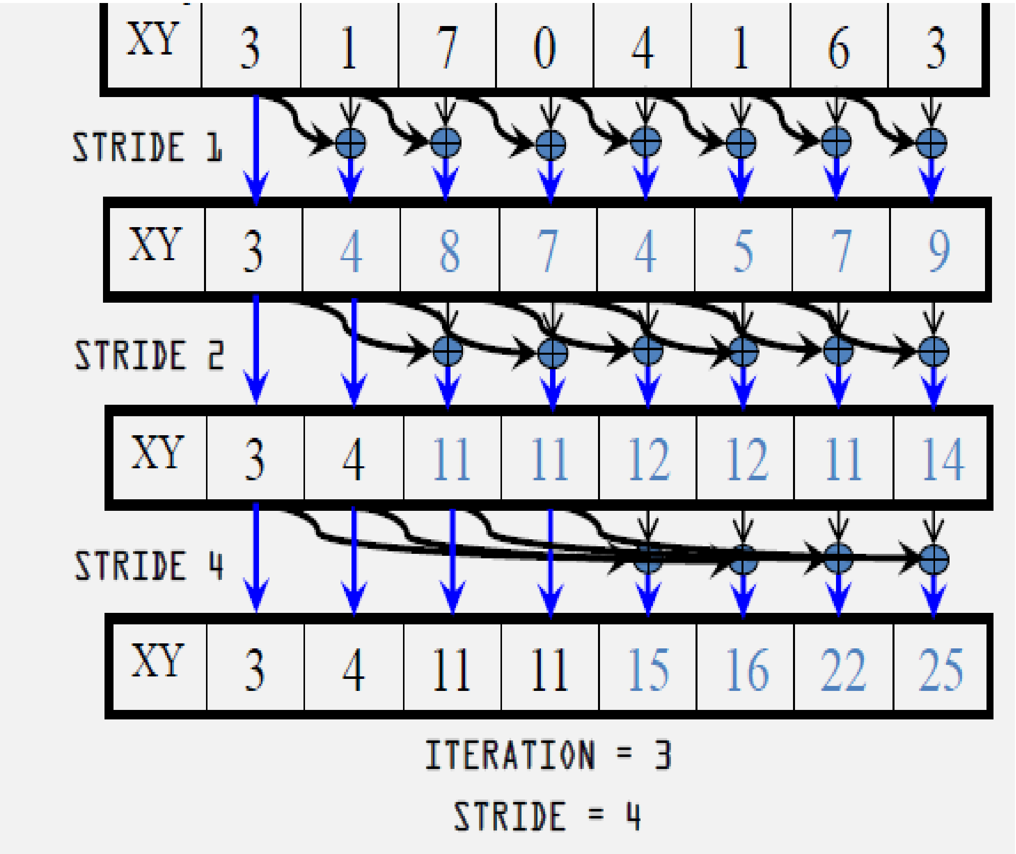
A Better Parallel Algorithm (cont.)

- In each iteration,
  - For each thread *j* from *stride* to *n-1* (*n-stride* threads)
    - Thread *j* adds elements *j* and *j-stride* from shared memory and writes result into element *j* in shared memory
    - Requires barrier synchronization, once before read and once before write

A Better Parallel Algorithm (cont.)



A Better Parallel Algorithm (cont.)



Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
- Barrier synchronization to ensure all input have been properly generated
- All threads secure input operand that can be overwritten by another thread
- Barrier synchronization to ensure that all threads have secured their input
- All threads perform addition and write output

The Better Scan Kernel

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y,
int InputSize) {
  __shared__ float XY[SECTION_SIZE];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {XY[threadIdx.x] = X[i];}
 // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {
    __syncthreads();
    float in1 = XY[threadIdx.x - stride];
    __syncthreads();
    XY[threadIdx.x] += in1;
  }
```

Work Efficiency Considerations

- This scan executes log(n) parallel iterations
  - An iteration performs (n-1), (n-2), (n-4),..,n/2 adds
  - Total adds: n * log(n) - (n-1) → O(n*log(n)) work
- This scan algorithm is not work efficient
  - Sequential scan algorithm does n adds
  - A factor of log(n) can hurt: 10x for 1024 elements!
- A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency

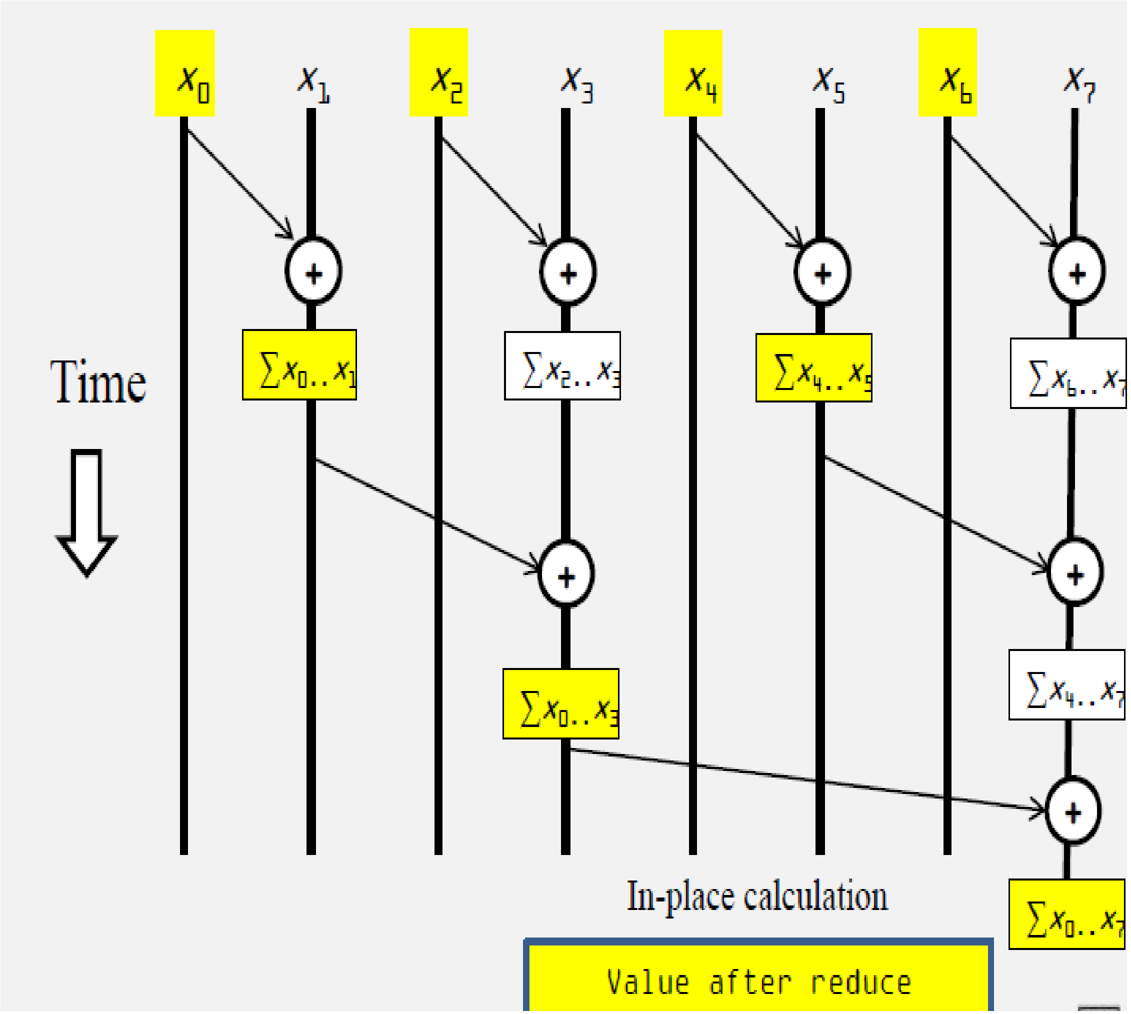
How To Improve Efficiency

- Two-phase balanced tree traversal
- Aggressive reuse of computation results
- Reducing control divergence with more complex thread index to data index mapping

Balanced Tree for Scan

- Form a balanced binary tree on the input data and sweep it to and from the root
- Here the tree is not an actual data structure, but a concept to determine what each thread does at each step
- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
  - Root holds sum of all leaves
  - Traverse back up the tree building the output from the partial sums

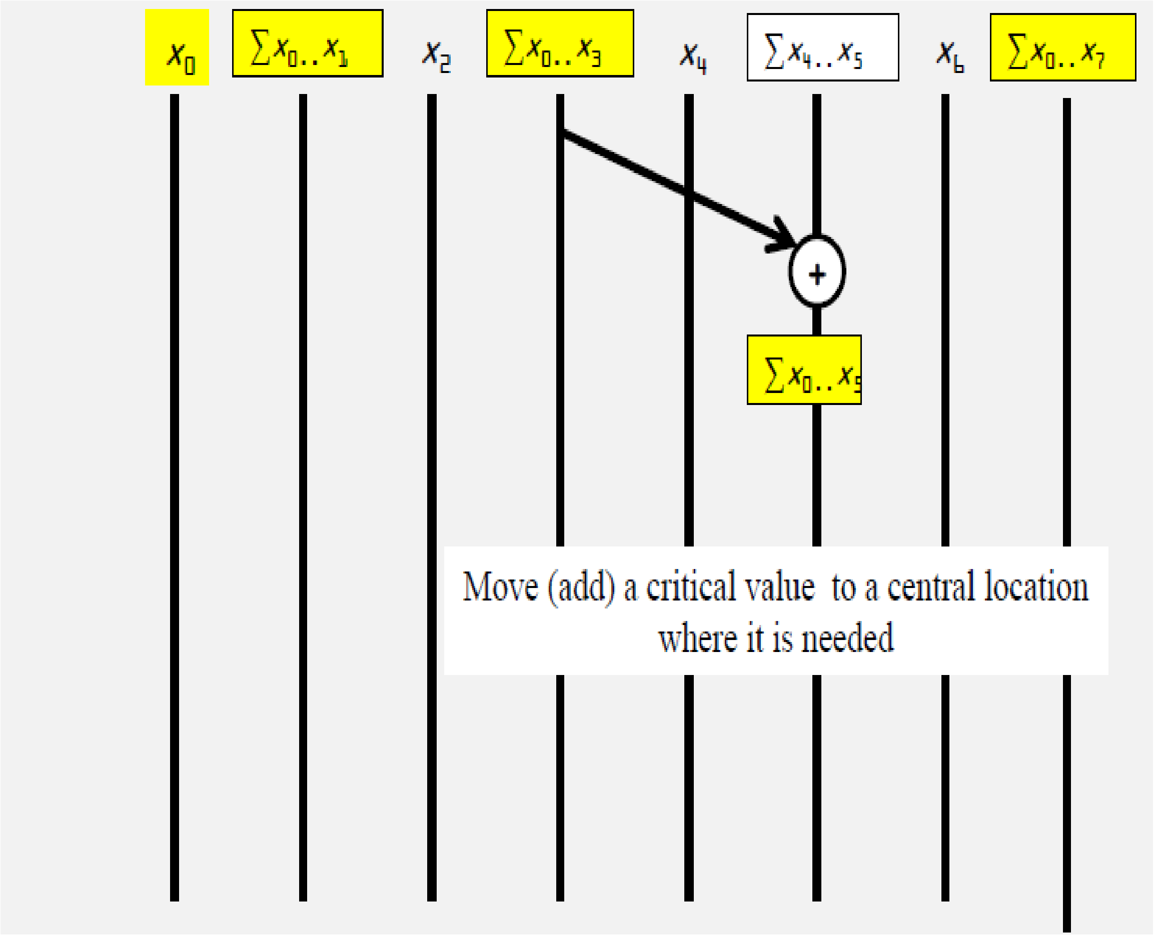
Parallel Scan – Reduction Phase



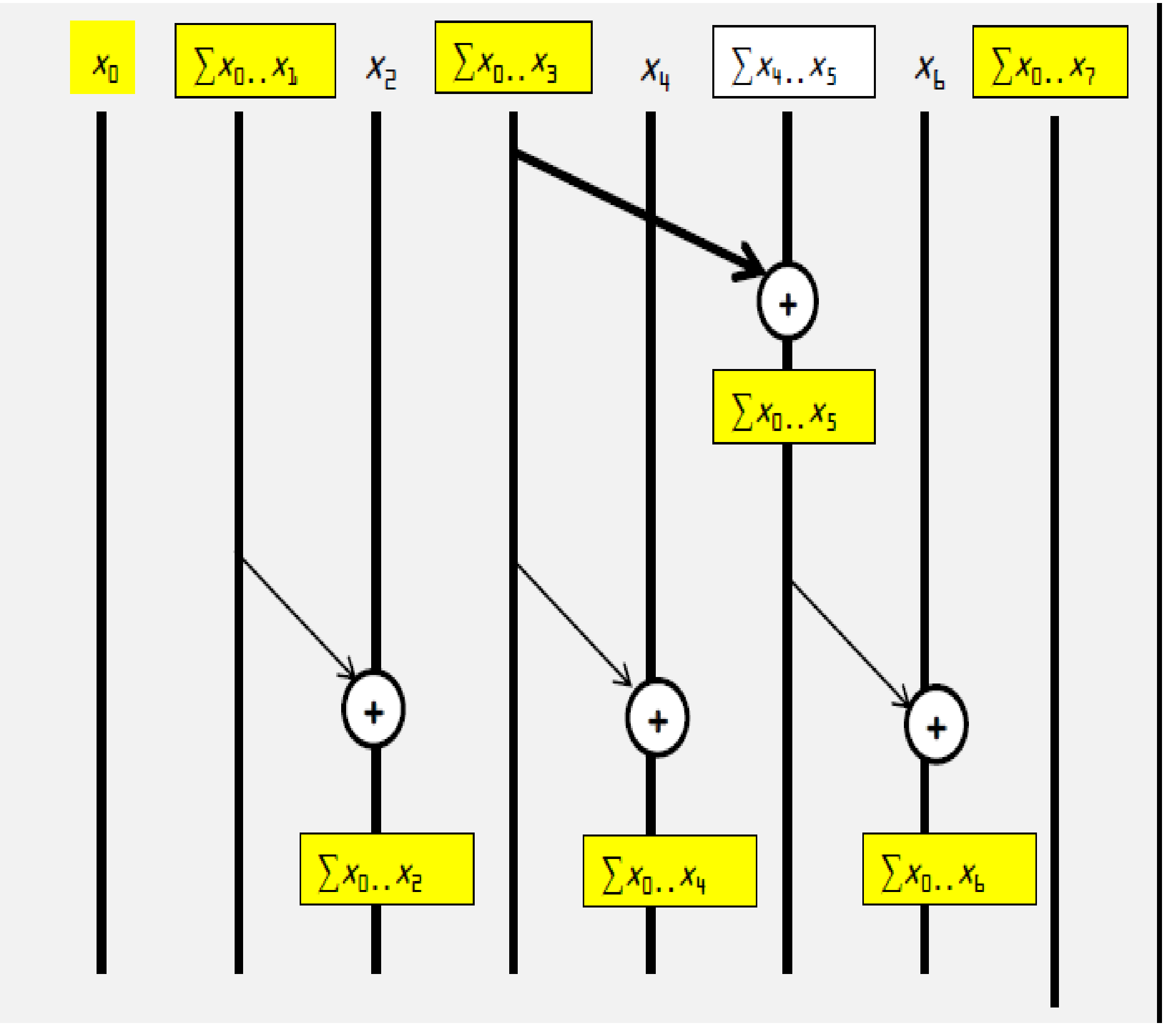
Reduction Phase Kernel Code

```
// XY[2*BLOCK_SIZE] is in shared memory

for (int stride = 1;stride <= BLOCK_SIZE; stride *= 2) {
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
     __syncthreads()
}
```

Parallel Scan –
Post Reduction Reverse Phase



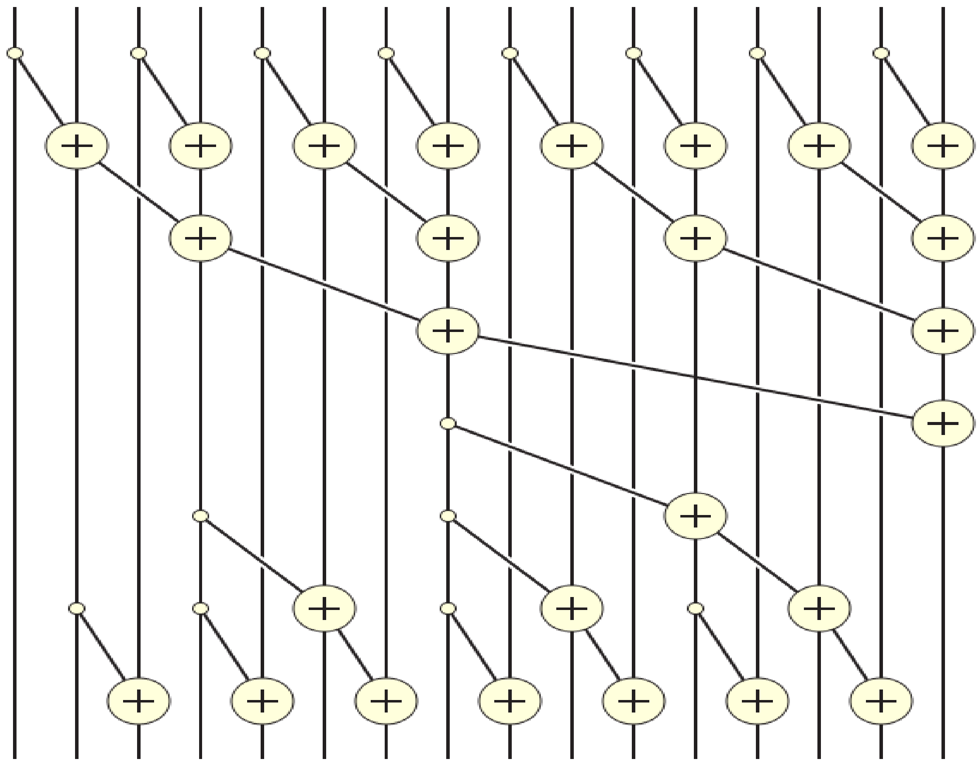
Parallel Scan – Post Reduction Reverse Phase



Post Reduction Reverse Phase Kernel Code

```
for (int stride = BLOCK_SIZE/2; stride > 0; stride /= 2) {
  __syncthreads();
  int index = (threadIdx.x+1)*stride*2 - 1;
  if(index+stride < 2*BLOCK_SIZE) {
    XY[index + stride] += XY[index];
  }
}
__syncthreads();
if (i < InputSize) Y[i] = XY[threadIdx.x];
```

Putting it all together



Efficiency Analysis

- The work efficient kernel executes log(n) parallel iterations in the reduction step
  - The iterations do n/2, n/4,..1 adds
  - Total adds: (n-1) → O(n) work
- It executes log(n)-1 parallel iterations in the post reduction reverse step
  - The iterations do 2-1, 4-1, …. n/2-1 adds
  - Total adds: (n-2) – (log(n)-1) → O(n) work
- Both phases perform up to no more than 2*(n-1) adds
- The total number of adds is no more than twice of that done in the efficient sequential algorithm

Tradeoffs

- The work efficient scan kernel is normally more desirable
  – Better Energy efficiency
  – Less execution resource requirement
- However, the work inefficient kernel could be better for absolute performance due to its single-step nature if there is sufficient execution resource

Summary

- Prefix scan is a common data-parallel primitive.
- A naïve parallel implementation is work inefficient.
- A work-efficient implementation takes a two phase balanced tree approach.
- There are tradeoffs between the two implementations.

Parallel Programming

Data-Parallel Primitives:

Split and Sort

Split and Sort

Primitive: Split
Input: $R_{in}[1, \ldots, n]$, $func(R_{in}[i]) \in [1,\ldots,F]$, i=1, …, n.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $func(R_{out}[i]) \leq func(R_{out}[j]), \forall i,j \in [1,..,n], i \leq j$.

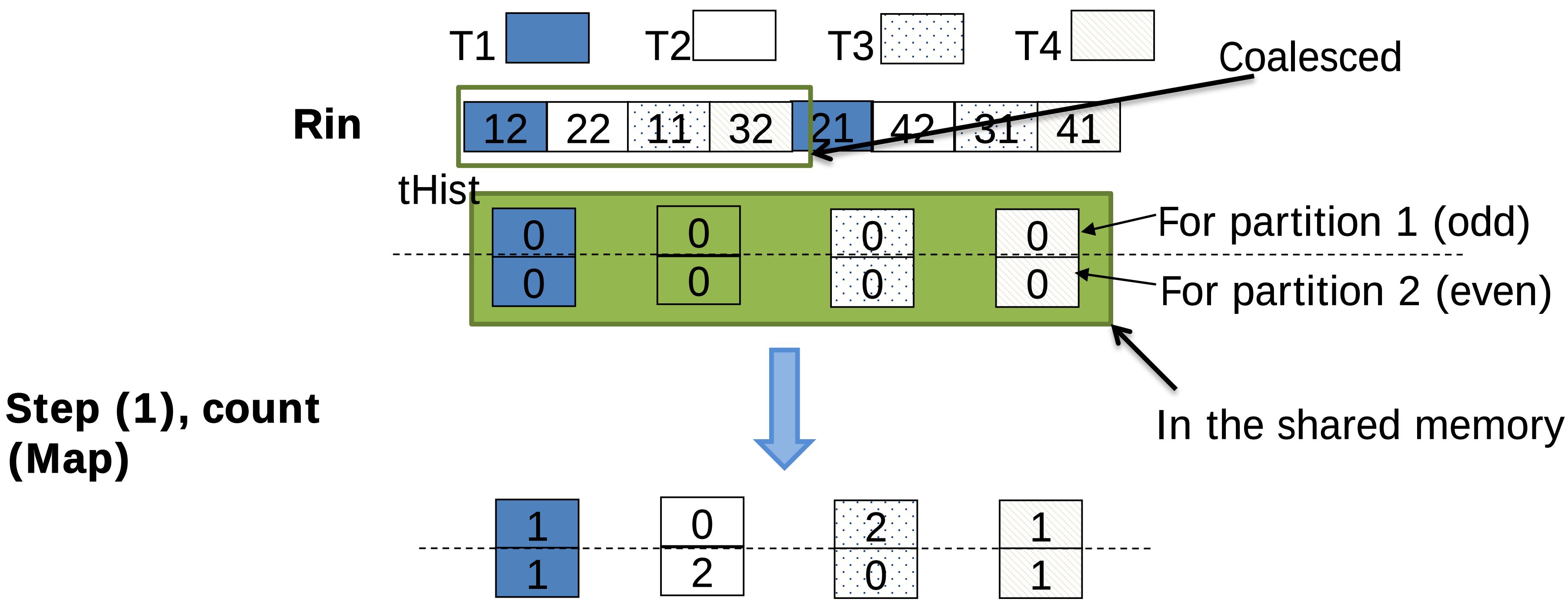
Primitive: Sort
Input: $R_{in}[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $R_{out}[i] \leq R_{out}[j], \forall i,j \in [1,..,n]$ and $i \leq j$.

Algorithm for Split

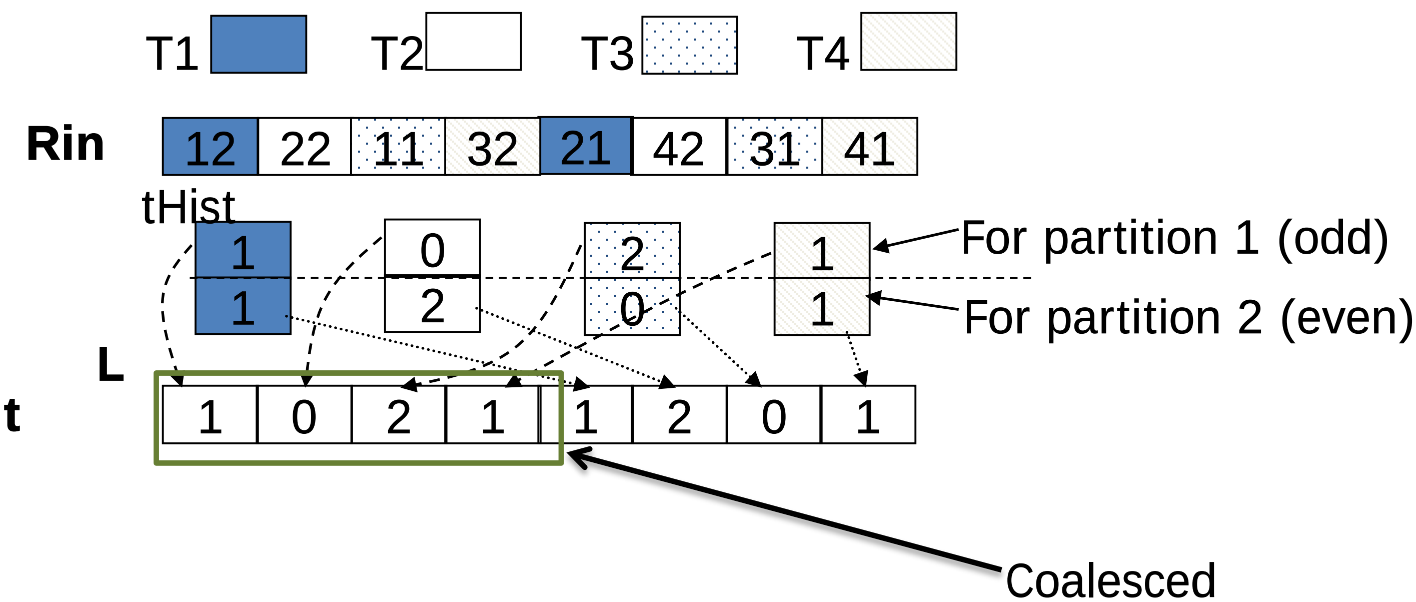
- A lock-free algorithm
  - Each thread is responsible for a portion of the input relation.
  - Each thread computes its local histogram (number of tuples in each output partition).
  - Given the local histograms, each thread computes its write locations.
  - Each thread writes the tuples to the output relation in parallel.

An Example of Split

Split (Rin[1,..., 8], fcn, Rout[1,...,8]), fcn(x)=x mod 2



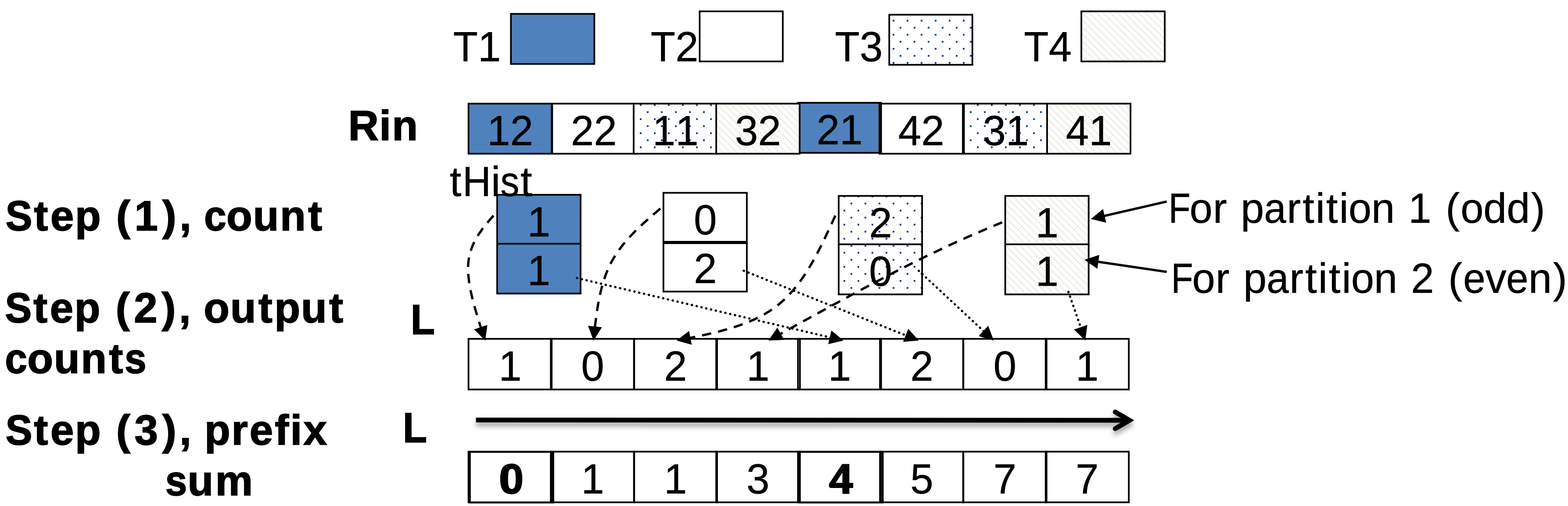
An Example of Split



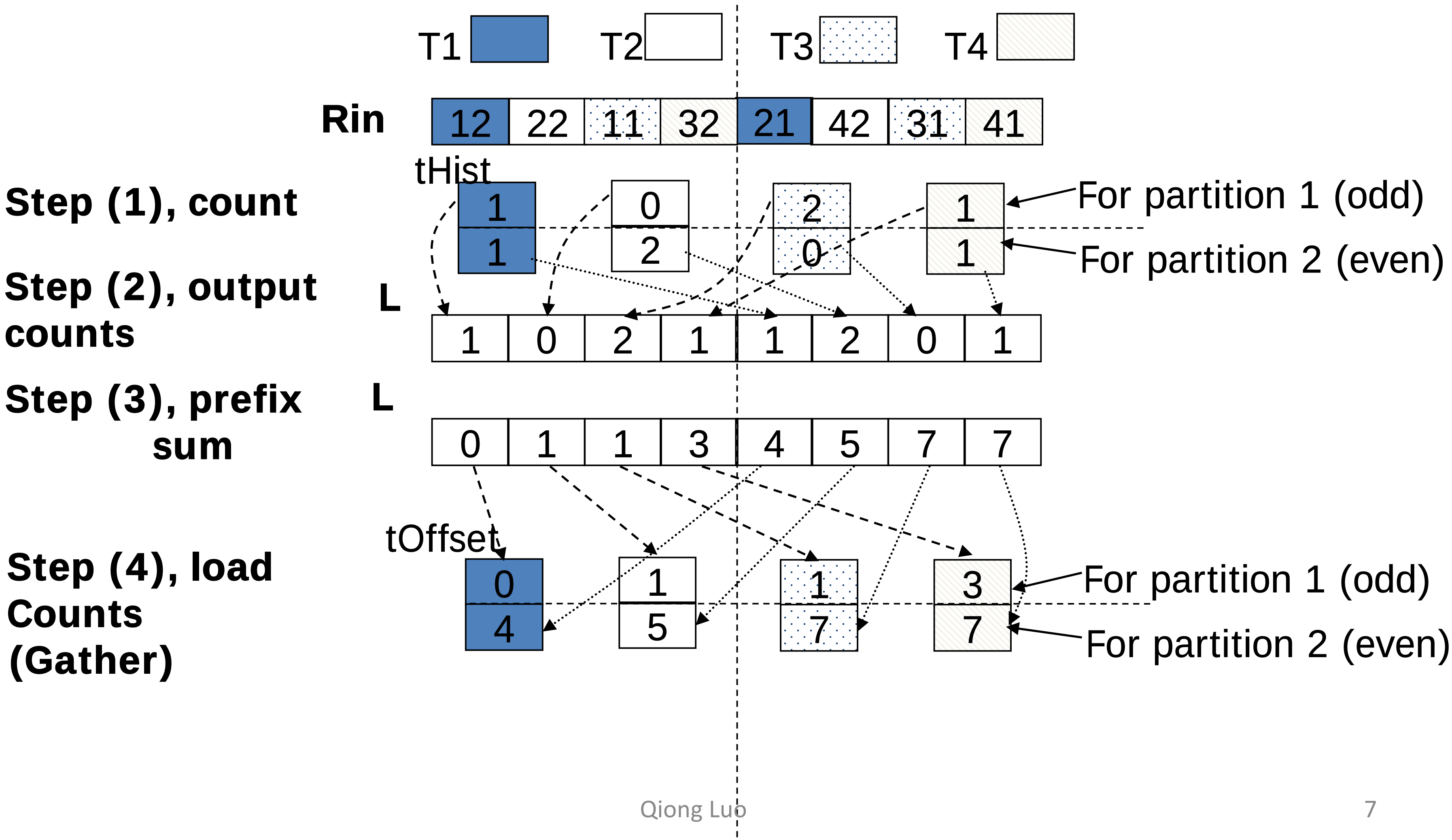
Step (1), count (Map)

Step (2), output Counts (Scatter)

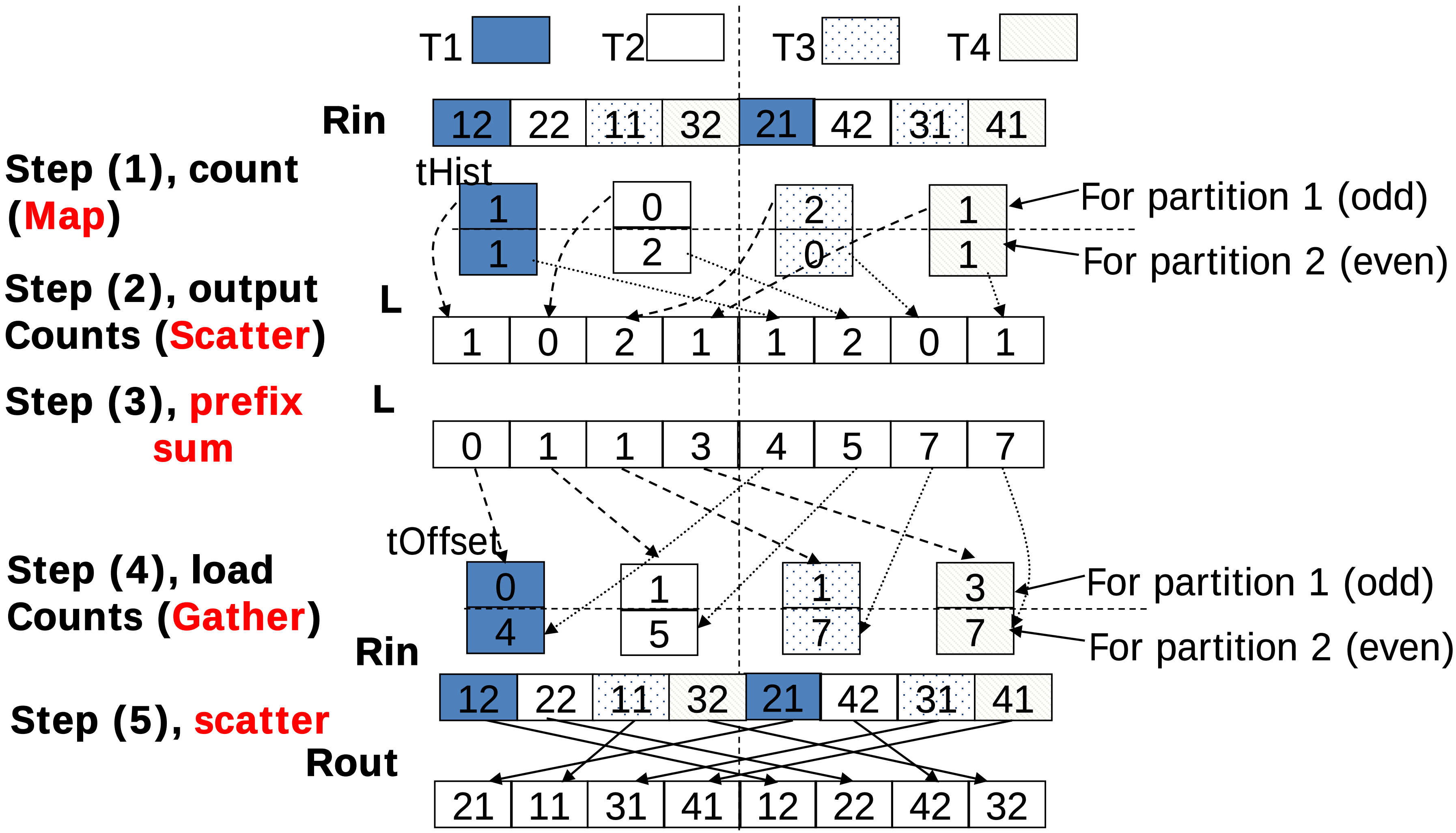
An Example of Split



An Example of Split



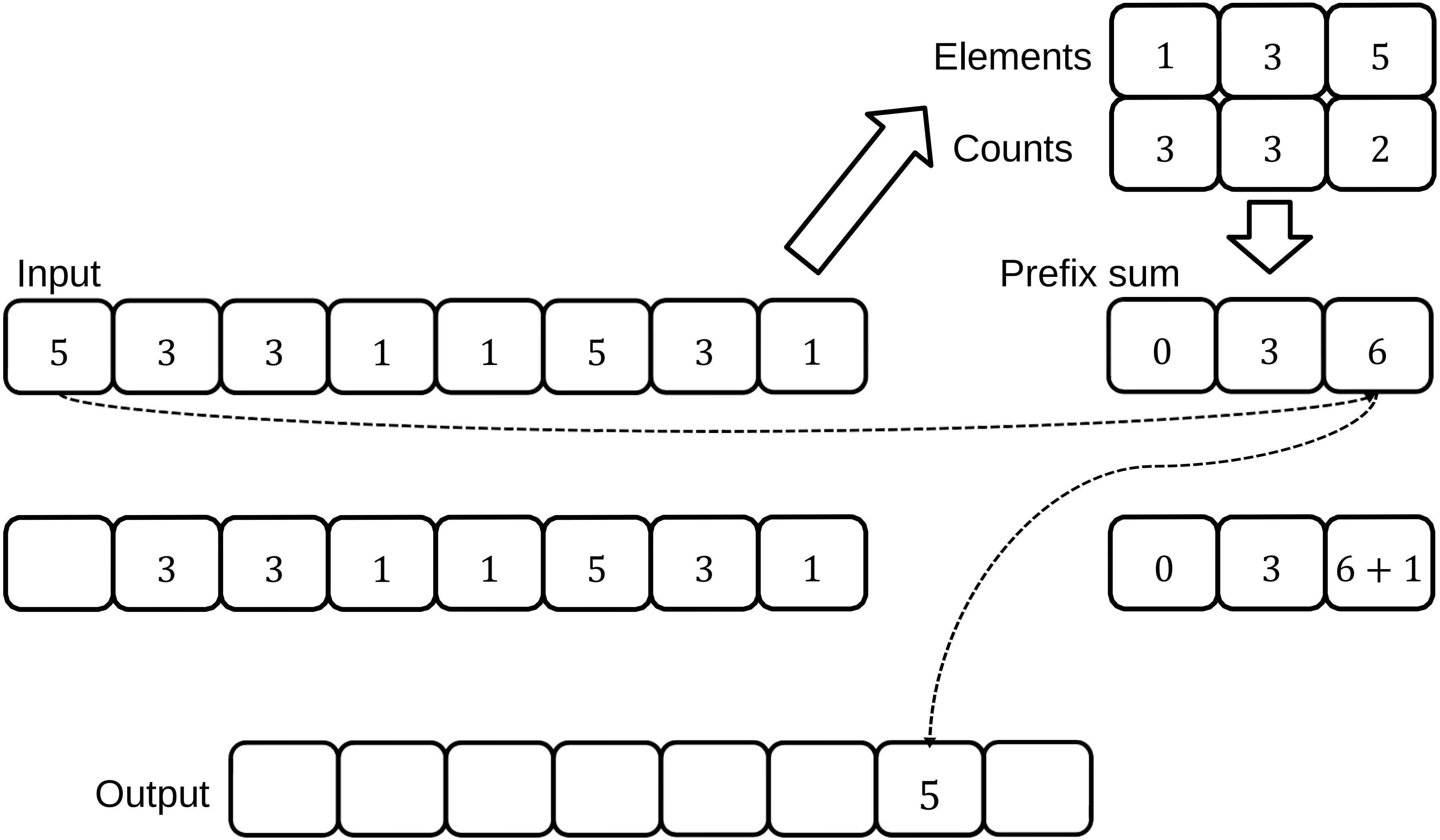
An Example of Split



Counting Sort Algorithm

- Sorting for equality-comparison elements, e.g., integers
- Assume
  - Constant number of possible values
  - Possible values known in advance
- Algorithm outline
  - Compute the histogram of each element
  - Prefix sum over the histogram
  - Move each element to its location
- Complexity (sequential) - O(n)

Counting Sort Example



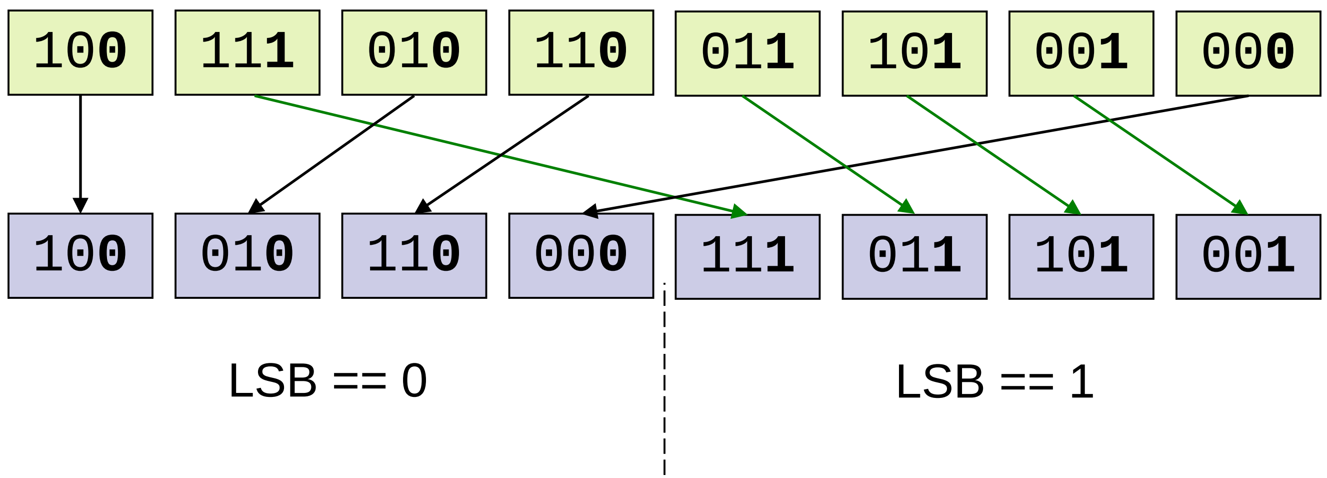
Radix Sort

- Iterated counting sort on individual digits (radix)
- Sort from the least significant bit (LSB) to the most significant bit (MSB)

Input	25	13	53	21	61	45	83	11
Sort 2nd digit	21	61	11	13	53	83	25	45
Sort 1st digit	11	13	21	25	45	53	61	83

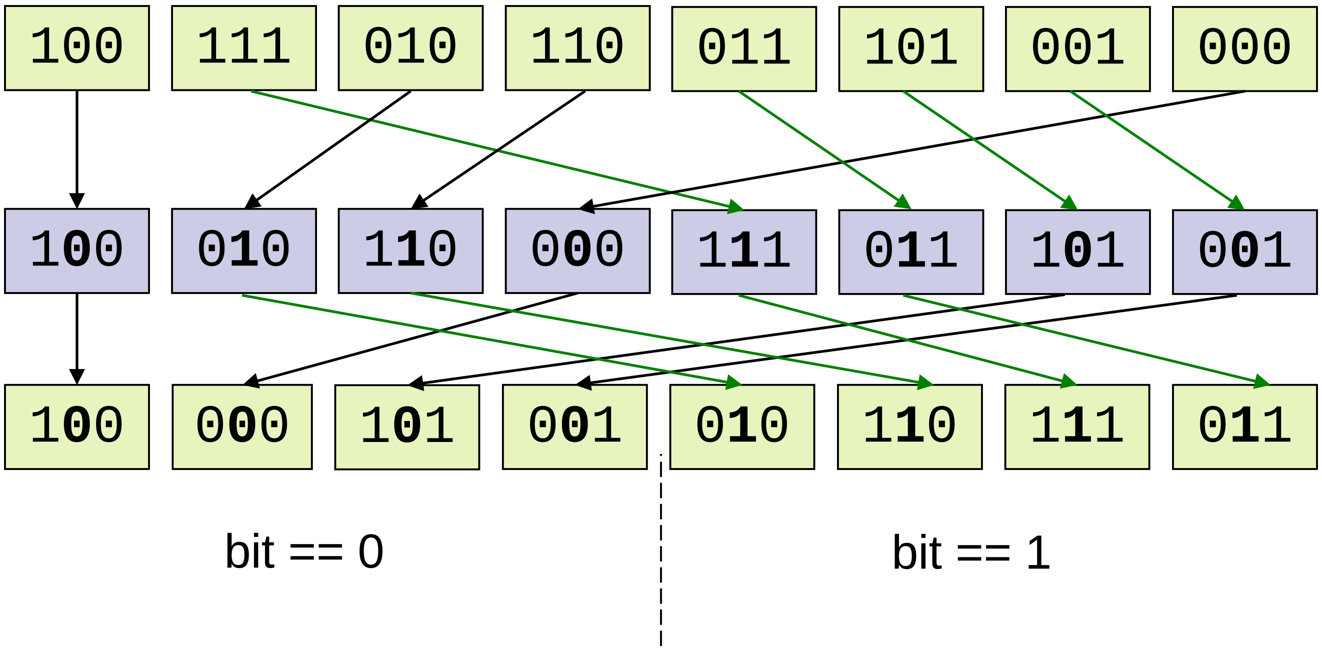
Radix Sort Example

- First pass: partition based on LSB



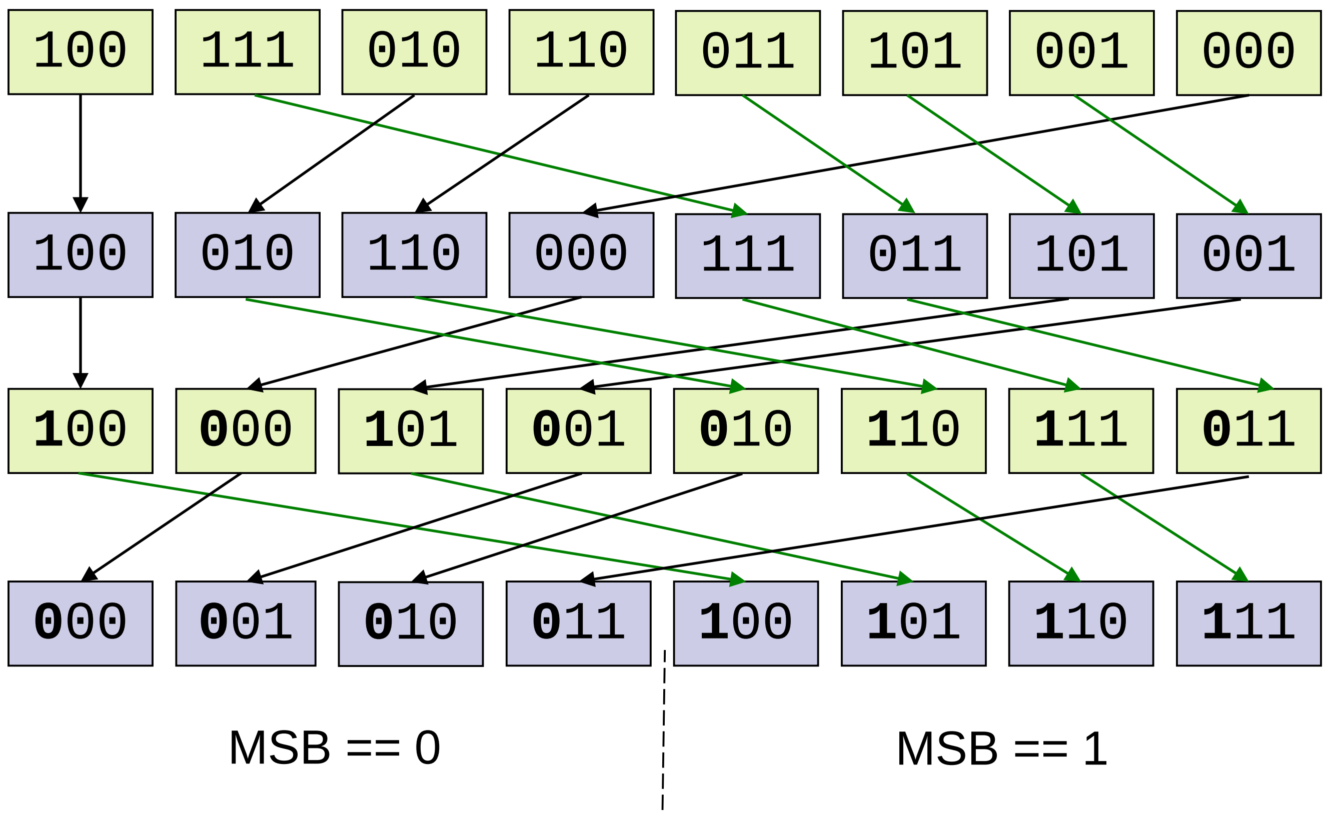
Radix Sort Example

- Second pass: partition based on second LSB



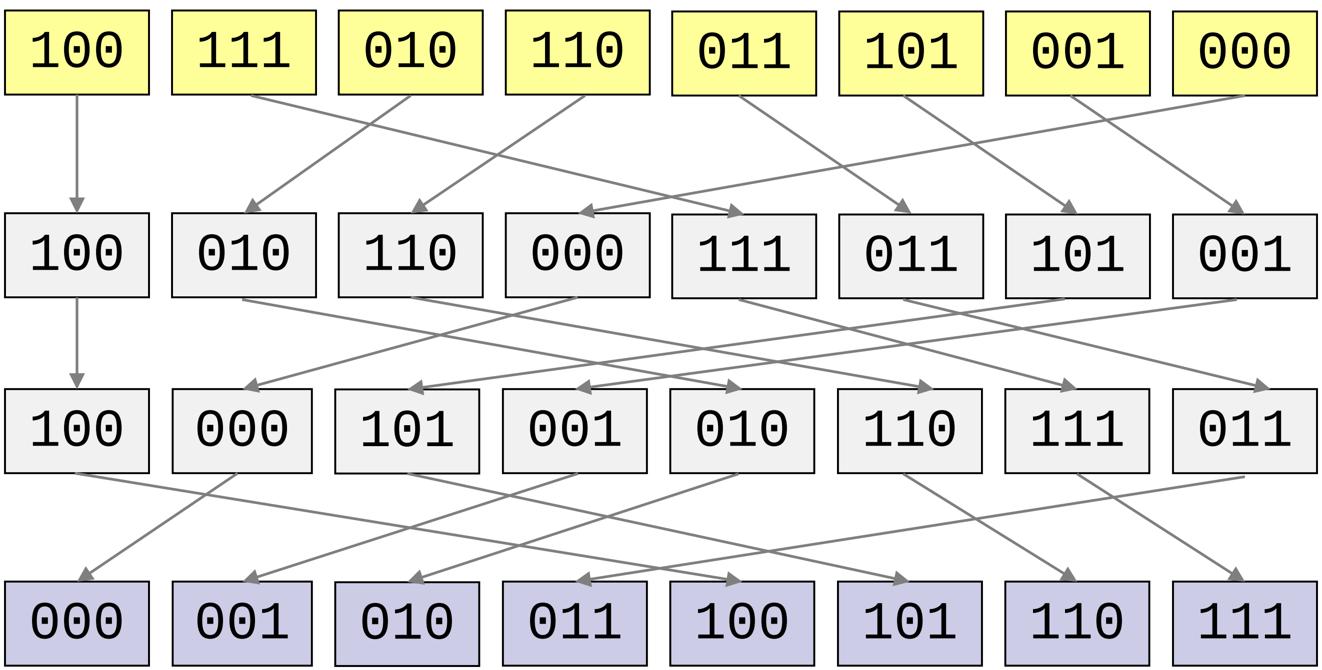
Radix Sort Example

- Final pass: partition based on MSB



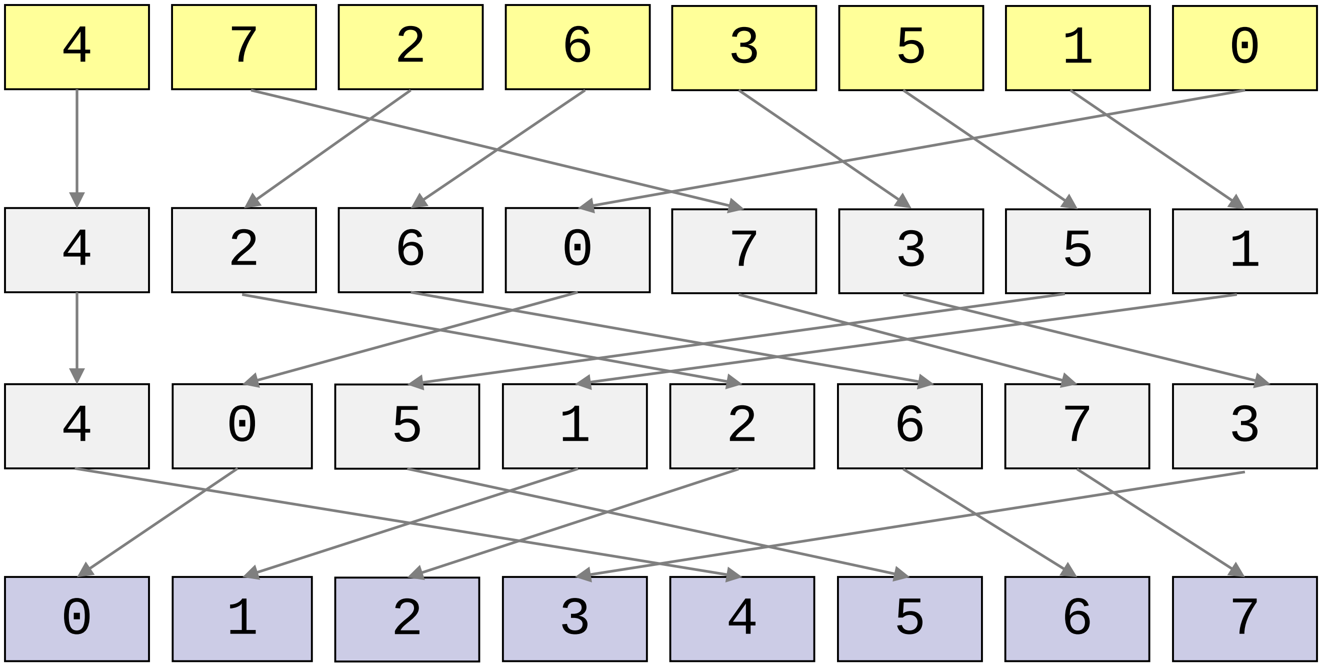
Radix Sort Example

- Completed:



Radix Sort Example

- Completed:



Parallel Radix Sort

1. Break input array into tiles
   – Each tile fits into shared memory for a thread block
2. Sort tiles in *parallel* with *radix sort*
3. Merge pairs of tiles using a *parallel bitonic merge* until all tiles are merged.

Our focus is on Step 2

Parallel Radix Sort

- Where is the parallelism?
  - Each tile is sorted in parallel
  - Where is the parallelism within a tile?
    - Each pass is done in sequence after the previous pass. No parallelism
    - Can we parallelize an individual pass? How?
  - Merge also has parallelism

Parallel Radix Sort in Each Pass

- Implement Split on the current bit under comparison
  - Histogram-based computation to count the number of 0/1s
  - Prefix scan to determine the output position of each element.
  - Efficient scatter of elements to target locations
  - Shared memory optimization
    - Histograms are stored in the shared memory.

Parallel Radix Sort

- Implement *split*. Given:
  - Array, *i*, at pass *n*:

100	111	010	110	011	101	001	000

  - Array, *b*, which is true/false for bit *n*:

0	1	0	0	1	1	1	0

- Output array with false keys before true keys:

100	010	110	000	111	011	101	001

Parallel Radix Sort

- Step 1: Compute *e* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array

Parallel Radix Sort

- Step 2: Exclusive Scan *e*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array

Parallel Radix Sort

- Step 3: Compute *totalFalses*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	$\boxed{1}$	e array
0	1	1	2	3	3	3	$\boxed{3}$	f array

totalFalses = e[n – 1] + f[n – 1]
totalFalses = 1 + 3
totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
								t array

t[i] = i – f[i] + totalFalses

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4								t array

t[0] = 0 – f[0] + totalFalses
t[0] = 0 – 0 + 4
t[0] = 4

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	$\boxed{1}$	1	2	3	3	3	3	f array
4	$\boxed{4}$							t array

t[1] = 1 – f[1] + totalFalses
t[1] = 1 – 1 + 4
t[1] = 4

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5						t array

t[2] = 2 – f[2] + totalFalses
t[2] = 2 – 1 + 4
t[2] = 5

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute *t* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array

t[i] = i – f[i] + totalFalses

totalFalses = 4

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0								d[i] = b[i] ? t[i] : f[i]

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0	4							d[i] = b[i] ? t[i] : f[i]

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0	4	1						d[i] = b[i] ? t[i] : f[i]

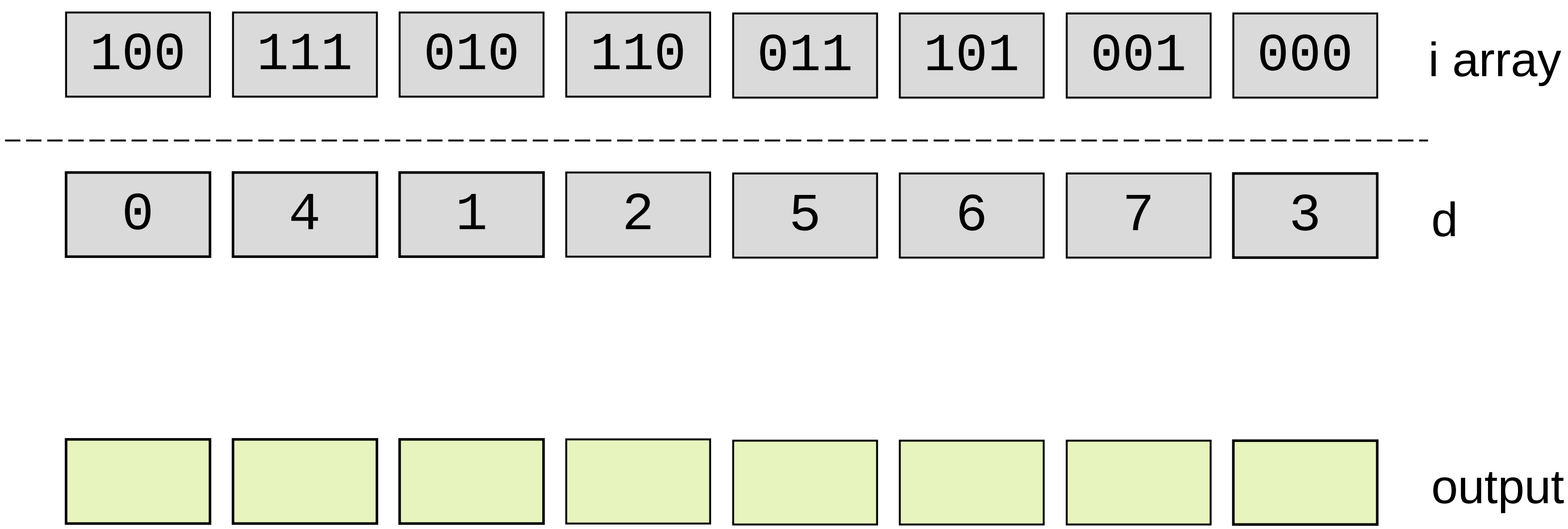
Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0	4	1	2	5	6	7	3	d[i] = b[i] ? t[i] : f[i]

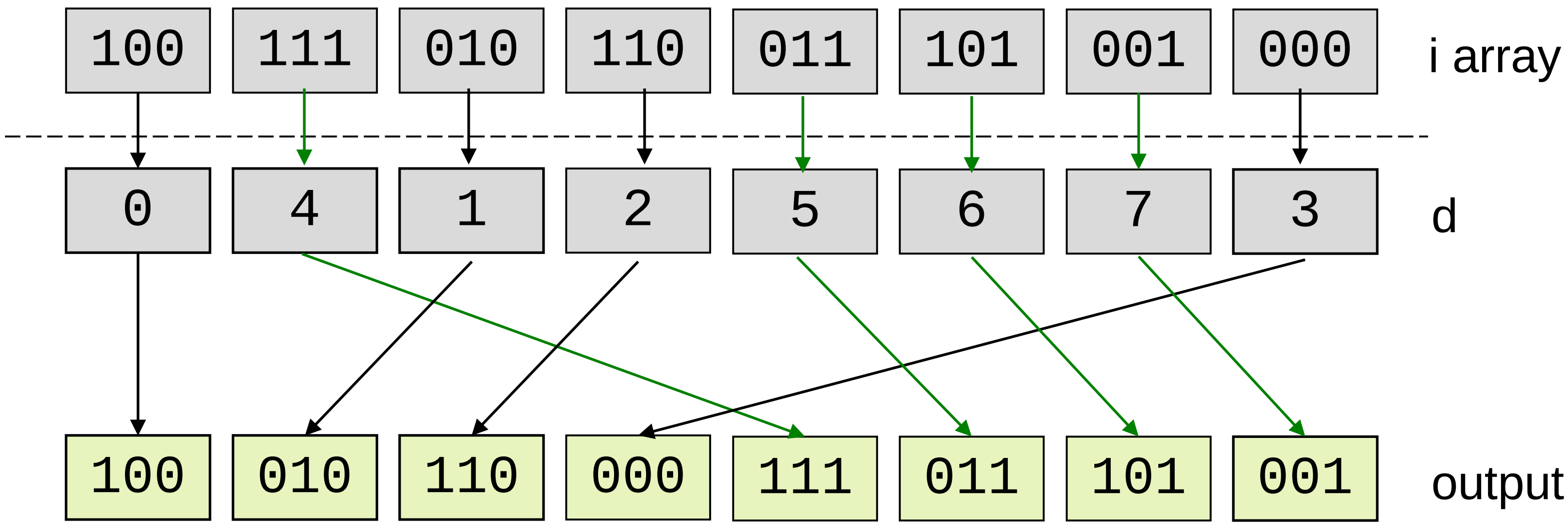
Parallel Radix Sort

- Step 5: Scatter based on address *d*



Parallel Radix Sort

- Step 5: Scatter based on address *d*



Radix Sort Analysis

- Integer sort
- Complexity $O(kn)$
  - n – number of elements
  - k– number of digits (constant)
- Parallel implementation (naïve)
  - For each radix (digit)
    - Compute radix histogram
    - Scan the histogram to compute offset for each element
    - Write the sorted elements
  - Work $O(kn)$
  - Time $O(k \log n)$

Summary

- Split and Sort are two common data-parallel primitives.
- They can be composed using simpler primitives.
- They are used widely in higher-level applications.
- Radix sort is currently the fastest sorting algorithm on the GPU.

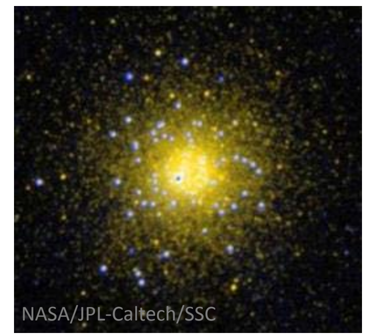
Parallel Programming

N-Body Simulation in CUDA

Slides based on Martin Burtscher's tutorial

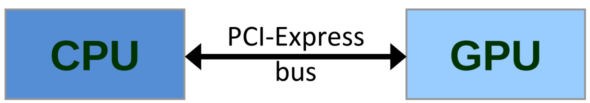
Outline

- Review: GPU programming
- N-body example
- Porting and tuning



CUDA Programming Model

- Non-graphics programming
  - Uses GPU as massively parallel co-processor



- SIMT (single-instruction multiple-threads) model
  - Thousands of threads needed for full efficiency

- C/C++ with extensions
  - Function launch
    - Calling functions on GPU
  - Memory management
    - GPU memory allocation, copying data to/from GPU
  - Declaration qualifiers
    - Device, shared, local, etc.
  - Special instructions
    - Barriers, fences, etc.
  - Keywords
    - threadIdx, blockIdx

Calling GPU Kernels

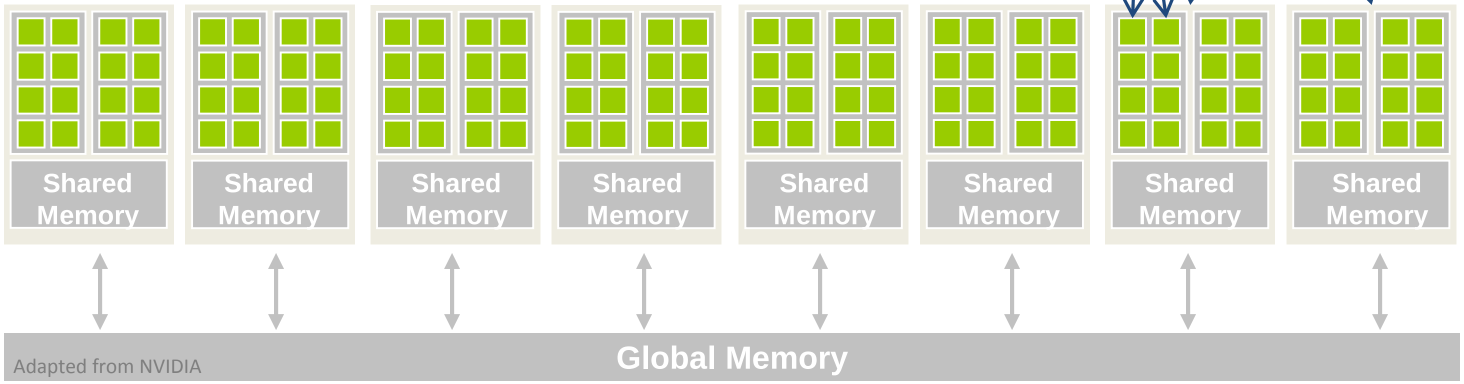
- Kernels are functions that run on the GPU
  - Callable by CPU code
  - CPU can continue processing while GPU runs kernel

```
KernelName<<<m, n>>>(arg1, arg2, ...);
```

- Launch configuration (programmer selectable)
  - GPU spawns m blocks of n threads per block (i.e., m*n threads total) that run a copy of the same function
  - Normal function parameters: passed conventionally
    - Different address space

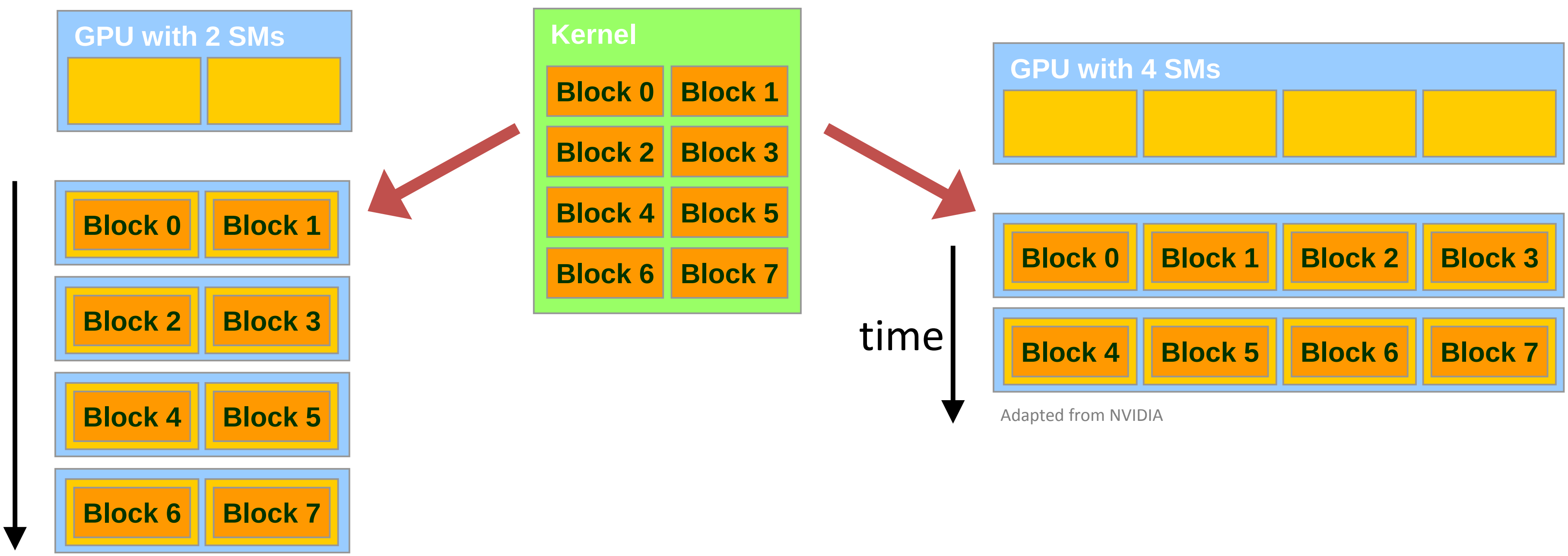
GPU Architecture

- GPUs consist of Streaming Multiprocessors (SMs)
  - 1 to 30 SMs per chip (run blocks)
- SMs contain Processing Elements (PEs)
  - 8, 32, or 192 PEs per SM (run threads)



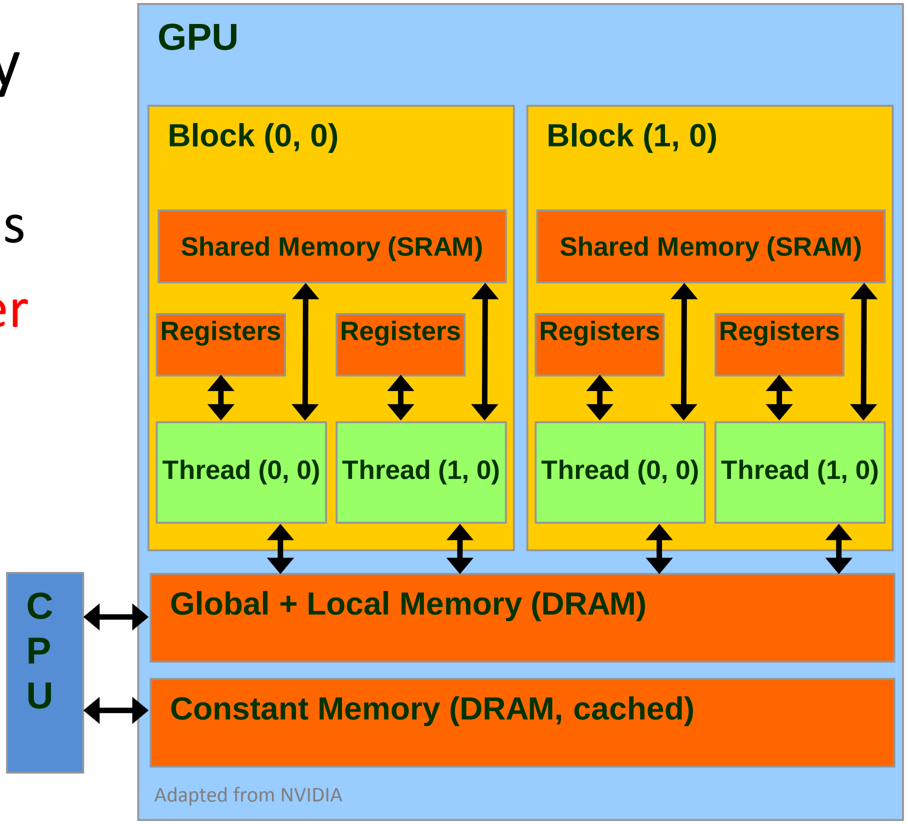
Block Scalability

- Hardware can assign blocks to SMs in any order
  - A kernel with enough blocks scales across GPUs
  - Not all blocks may be resident at the same time



GPU Memories

- Separate from CPU memory
  - CPU can access GPU's global & constant mem. via PCIe bus
  - Requires slow explicit transfer
- Visible GPU memory types
  - Registers (per thread)
  - Local mem. (per thread)
  - Shared mem. (per block)
    - Software-controlled cache
  - Global mem. (per kernel)
  - Constant mem. (read only)



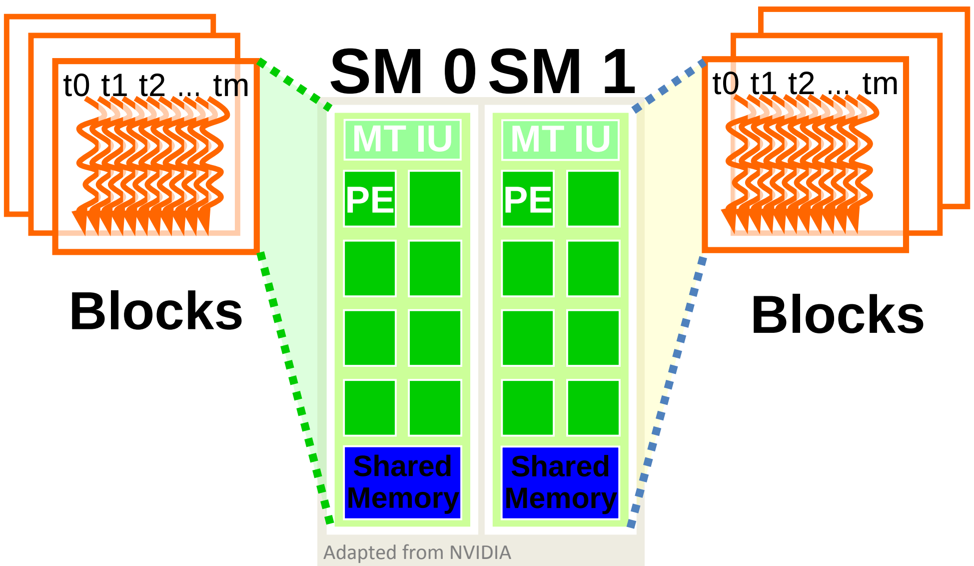
- Slow communic. between blocks

SM Internals (Fermi and Kepler)

- Caches
  - Software-controlled shared memory
  - Hardware-controlled incoherent L1 data cache
  - 64 kB combined size, can be split 16/48, 32/32, 48/16
- Synchronization support
  - Fast hardware barrier within block (*__syncthreads()*)
  - Fence instructions: memory consistency & coherency
- Special operations
  - Thread voting (warp-based reduction operations)

Block and Thread Allocation Limits

- Blocks assigned to SMs
  - Until first limit reached
- Threads assigned to PEs



- Hardware limits
  - 8/16 active blocks/SM
  - 1024, 1536, or 2048 resident threads/SM
  - 512 or 1024 threads/blk
  - 16k, 32k, or 64k regs/SM
  - 16 kB or 48 kB shared memory per SM
  - 2^{16}-1 or 2^{31}-1 blks/kernel

Warp-based Execution

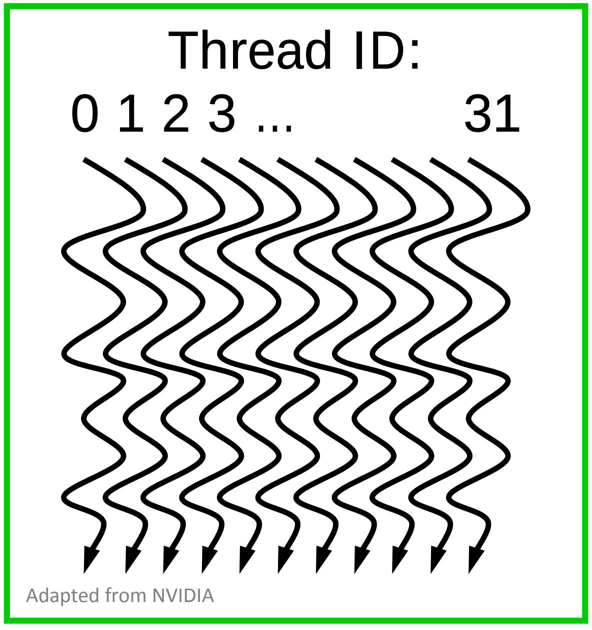
- 32 contiguous threads form a warp
  - Execute same instruction in same cycle (or disabled)
  - Warps are scheduled out-of-order with respect to each other to hide latencies

- Thread divergence
  - Some threads in warp jump to different PC than others
  - Hardware runs subsets of warp until they re-converge
  - Results in reduction of parallelism (performance loss)

Thread Divergence

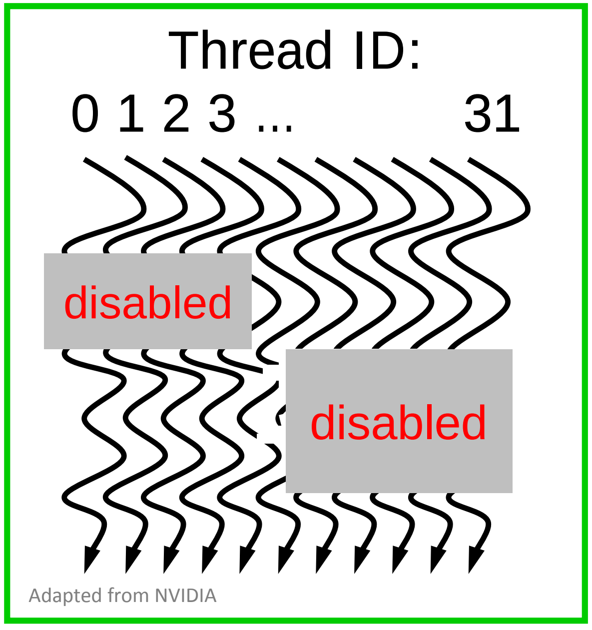
- Non-divergent code

```
if (threadID >= 32) {
   some_code;
} else {
   other_code;
}
```



- Divergent code

```
if (threadID >= 13) {
   some_code;
} else {
   other_code;
}
```



Parallel Memory Accesses

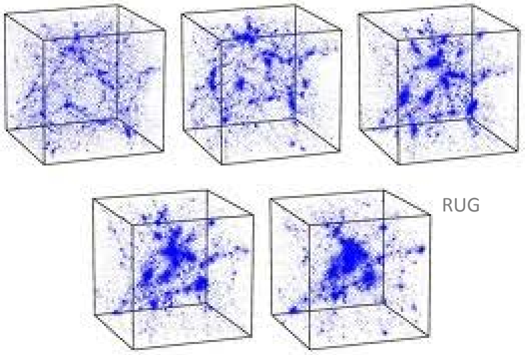
- Coalesced main memory access
  - Under some conditions, HW combines multiple (half) warp memory accesses into a single coalesced access
- Bank-conflict-free shared memory access
  - No superword alignment or contiguity requirements

Warnings for GPU Programming

- GPUs can only execute some types of code fast
  - Need lots of data parallelism, data reuse, & regularity
- GPUs are harder to program and tune than CPUs
  - poor tool support
  - architecture
  - poor support for irregular code

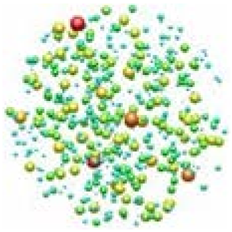
N-body Simulation

- Time evolution of physical system
  - System consists of bodies
  - "n" is the number of bodies
  - Bodies interact via pair-wise forces



RUG

- Many systems can be modeled in this way
  - Star/galaxy clusters (gravitational force)
  - Particles (electric force, magnetic force)



Simple N-body Algorithm

- Algorithm

  Initialize body masses, positions, and velocities
  Iterate over time steps {
      Accumulate forces acting on each body
      Update body positions and velocities based on force
  }
  Output result

- More sophisticated n-body algorithms exist
  - Barnes Hut algorithm
  - Fast Multipole Method (FMM)

Key Loops (Pseudo Code)

```
bodySet = ...;  // input
for timestep do {  // sequential
  foreach Body b1 in bodySet { // O(n²) parallel
    foreach Body b2 in bodySet {
      if (b1 != b2) {
        b1.addInteractionForce(b2);
      }
    }
  }
  foreach Body b in bodySet {  // O(n) parallel
    b.Advance();
  }
}
// output result
```

Force Calculation C Code

```
struct Body {
  float mass, posx, posy, posz; // mass and 3D position
  float velx, vely, velz, accx, accy, accz; // 3D velocity & accel
} *body;

for (i = 0; i < nbodies; i++) {
  . . .
  for (j = 0; j < nbodies; j++) {
    if (i != j) {
      dx = body[j].posx - px; // delta x
      dy = body[j].posy - py; // delta y
      dz = body[j].posz - pz; // delta z
      dsq = dx*dx + dy*dy + dz*dz; // distance squared
      dinv = 1.0f / sqrtf(dsq + epssq); // inverse distance
      scale = body[j].mass * dinv * dinv * dinv; // scaled force
      ax += dx * scale; // accumulate x contribution of accel
      ay += dy * scale;  az += dz * scale; // ditto for y and z
    }
  }
. . .
```

N-body Algorithm Suitability for GPU

- Lots of data parallelism
  - Force calculations are independent
  - Should be able to keep SMs and PEs busy
- Sufficient memory access regularity
  - All force calculations access body data in same order
  - Should have lots of coalesced memory accesses
- Sufficient code regularity
  - All force calculations are identical
  - There should be little thread divergence
- Plenty of data reuse
  - $O(n^2)$ operations on $O(n)$ data
  - CPU/GPU transfer time is insignificant

C to CUDA Conversion

- Two CUDA kernels
  - Force calculation
  - Advance position and velocity

- Benefits
  - Force calculation requires over 99.9% of runtime
    - Primary target for acceleration
  - Advancing kernel unimportant to runtime
    - But allows to keep data on GPU during entire simulation
    - Minimizes GPU/CPU transfers

C to CUDA Conversion

```
__global__ void ForceCalcKernel(int nbodies, struct Body *body, ...) {
  . . .
}
__global__ void AdvancingKernel(int nbodies, struct Body *body, ...) {
  . . .
}

int main(...) {
  Body *body, *bodyl;
  . . .
  cudaMalloc((void**)&bodyl, sizeof(Body)*nbodies);
  cudaMemcpy(bodyl, body, sizeof(Body)*nbodies, cuda…HostToDevice);
  for (timestep = ...) {
    ForceCalcKernel<<<1, 1>>>(nbodies, bodyl, ...);
    AdvancingKernel<<<1, 1>>>(nbodies, bodyl, ...);
  }
  cudaMemcpy(body, bodyl, sizeof(Body)*nbodies, cuda…DeviceToHost);
  cudaFree(bodyl);
  . . .
}
```

Evaluation Methodology

- Systems and compilers
  - CC 1.3: Quadro FX 5800, nvcc 3.2
    - 30 SMs, 240 PEs, 1.3 GHz, 30720 resident threads
  - CC 2.0: Tesla C2050, nvcc 3.2
    - 14 SMs, 448 PEs, 1.15 GHz, 21504 resident threads
  - CC 3.0: GeForce GTX 680, nvcc 4.2
    - 8 SMs, 1536 PEs, 1.0 GHz, 16384 resident threads
- Input and metric
  - 1k, 10k, or 100k star clusters (Plummer model)
  - Median runtime of three experiments, excluding I/O

1-Thread Performance

- Problem size
  - n=10000, step=1
  - n=10000, step=1
  - n=3000, step=1
- Slowdown rel. to CPU
  - CC 1.3: 72.4
  - CC 2.0: 36.7
  - CC 3.0: 68.1

  (Note: comparing different GPUs to different CPUs)
- Performance
  - 1 thread is one to two orders of magnitude slower on GPU than CPU
- Reasons
  - No caches (CC 1.3)
  - Not superscalar
  - Slower clock frequency
  - No SMT latency hiding

Using N Threads

- Approach
  - Eliminate outer loop
  - Instantiate n copies of inner loop, one per body
- Threading
  - Blocks can only hold 512 or 1024 threads
    - Up to 8/16 blocks can be resident in an SM at a time
    - SM can hold 1024, 1536, or 2048 threads
    - Use 256 threads per block (works for all three GPUs)
  - Need multiple blocks
    - Last block may not need all of its threads

Using N Threads

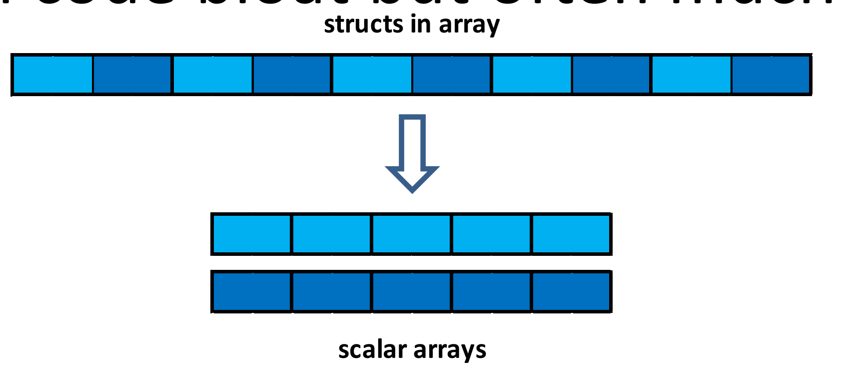
```
__global__ void ForceCalcKernel(int nbodies, struct Body *body, ...) {
  for (i = 0; i < nbodies; i++) {
  i = threadIdx.x + blockIdx.x * blockDim.x; // compute i
  if (i < nbodies) { // in case last block is only partially used
    for (j = ...) {
      . . .
    }
  }
}
__global__ void AdvancingKernel(int nbodies,...) // same changes

#define threads 256
int main(...) {
  . . .
  int blocks = (nbodies + threads - 1) / threads; // compute block cnt
  for (timestep = ...) {
    ForceCalcKernel<<<1, 1blocks, threads>>>(nbodies, bodyl, ...);
    AdvancingKernel<<<1, 1blocks, threads>>>(nbodies, bodyl, ...);
  }
}
```

N Thread Speedup

- Relative to 1 GPU thread
  - CC 1.3: 7781 (240 PEs)
  - CC 2.0: 6495 (448 PEs)
  - CC 3.0: 12150 (1536 PEs)
- Relative to 1 CPU thread
  - CC 1.3: 107.5
  - CC 2.0: 176.7
  - CC 3.0: 176.2
- Performance
  - Speedup much higher than number of PEs (32, 14.5, and 7.9 times)
  - Due to SMT latency hiding
- Per-core performance
  - CPU core delivers up to 4.4, 5, and 8.7 times as much performance as a GPU core (PE)

Using Scalar Arrays

- Data structure conversion
  - Arrays of structs are bad for coalescing
  - Bodies' elements (e.g., mass fields) are not adjacent
- Optimize data structure
  - Use multiple scalar arrays, one per field (need 10)
  - Results in code bloat but often much better speed



Using Scalar Arrays

```
__global__ void ForceCalcKernel(int nbodies, float *mass, ...) {
  // change all “body[k].blah” to “blah[k]”
}
__global__ void AdvancingKernel(int nbodies, float *mass, ...) {
  // change all “body[k].blah” to “blah[k]”
}

int main(...) {
  float *mass, *posx, *posy, *posz, *velx, *vely, *velz, *accx, *accy,*accz;
  float *massl, *posxl, *posyl, *poszl, *velxl, *velyl, *velzl, ...;
  mass = (float *)malloc(sizeof(float) * nbodies); // etc
  . . .
  cudaMalloc((void**)&massl, sizeof(float)*nbodies); // etc
  cudaMemcpy(massl, mass, sizeof(float)*nbodies, cuda…HostToDevice); // etc
  for (timestep = ...) {
    ForceCalcKernel<<<blocks, threads>>>(nbodies, massl, posxl, ...);
    AdvancingKernel<<<blocks, threads>>>(nbodies, massl, posxl, ...);
  }
  cudaMemcpy(mass, massl, sizeof(float)*nbodies, cuda…DeviceToHost); // etc
  . . .
}
```

Scalar Array Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Relative to struct
  - CC 1.3: 0.83
  - CC 2.0: 0.96
  - CC 3.0: 0.82
- Performance
  - Threads access same memory locations, not adjacent ones
    - Always combined but not really coalesced access
  - Slowdowns may be due to DRAM page/TLB misses
- Scalar arrays
  - Still needed (see later)

Constant Kernel Parameters

- Kernel parameters
  - Lots of parameters due to scalar arrays
  - All but one parameter never change their value
- Constant memory
  - “Pass” parameters only once
  - Copy them into GPU’s constant memory
- Performance implications
  - Reduced parameter passing overhead
  - Constant memory has hardware cache

Constant Kernel Parameters

```
__constant__ int nbodiesd;
__constant__ float dthfd, epssqd, float *massd, *posxd, ...;

__global__ void ForceCalcKernel(int step) {
  // rename affected variables (add “d” to name)
}

__global__ void AdvancingKernel() {
  // rename affected variables (add “d” to name)
}

int main(...) {
  . . .
  cudaMemcpyToSymbol(massd, &massl, sizeof(void *)); // etc
  . . .
  for (timestep = ...) {
    ForceCalcKernel<<<1, 1>>>(step);
    AdvancingKernel<<<1, 1>>>();
  }
  . . .
}
```

Constant Mem. Parameter Speedup

- Problem size
  - n=1000, step=10000
  - n=1000, step=10000
  - n=3000, step=10000
- Speedup
  - CC 1.3: 1.015
  - CC 2.0: 1.016
  - CC 3.0: 0.971
- Performance
  - Minimal perf. impact
  - May be useful for very short kernels that are often invoked
- Benefit
  - Less shared memory used on CC 1.3 devices

Using the RSQRTF Instruction

- Slowest kernel operation
  - Computing one over the square root is very slow
  - GPU has slightly imprecise but fast 1/sqrt instruction
    (frequently used in graphics code to calculate inverse of distance to a point)
- IEEE floating-point accuracy compliance
  - CC 1.x is not entirely compliant
  - CC 2.x and above are compliant but also offer faster non-compliant instructions

Using the RSQRT Instruction

```
for (i = 0; i < nbodies; i++) {
  . . .
  for (j = 0; j < nbodies; j++) {
    if (i != j) {
      dx = body[j].posx - px;
      dy = body[j].posy - py;
      dz = body[j].posz - pz;
      dsq = dx*dx + dy*dy + dz*dz;
      dinv = 1.0f / sqrtf(dsq + epssq);
      dinv = rsqrtf(dsq + epssq);
      scale = body[j].mass * dinv * dinv * dinv;
      ax += dx * scale;
      ay += dy * scale;
      az += dz * scale;
    }
  }
  . . .
}
```

RSQRT Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 0.99
  - CC 2.0: 1.83
  - CC 3.0: 1.64
- Performance
  - Little change for CC 1.3
    - Compiler automatically uses less precise RSQRTF as most FP ops are not fully precise anyhow
  - 83% speedup for CC 2.0
    - Over entire application
    - Compiler defaults to precise instructions
    - Explicit use of RSQRTF indicates imprecision okay

Using 2 Loops to Avoid If Statement

- “if (i != j)” creates code divergence
  - Break loop into two loops to avoid if statement

```
for (j = 0; j < nbodies; j++) {
  if (i != j) {
    dx = body[j].posx - px;
    dy = body[j].posy - py;
    dz = body[j].posz - pz;
    dsq = dx*dx + dy*dy + dz*dz;
    dinv = rsqrtf(dsq + epssq);
    scale = body[j].mass * dinv * dinv * dinv;
    ax += dx * scale;
    ay += dy * scale;
    az += dz * scale;
  }
}
```

Using 2 Loops to Avoid If Statement

```
for (j = 0; j < i; j++) {
  dx = body[j].posx - px;
  dy = body[j].posy - py;
  dz = body[j].posz - pz;
  dsq = dx*dx + dy*dy + dz*dz;
  dinv = rsqrtf(dsq + epssq);
  scale = body[j].mass * dinv * dinv * dinv;
  ax += dx * scale;
  ay += dy * scale;
  az += dz * scale;
}
for (j = i+1; j < nbodies; j++) {
  dx = body[j].posx - px;
  dy = body[j].posy - py;
  dz = body[j].posz - pz;
  dsq = dx*dx + dy*dy + dz*dz;
  dinv = rsqrtf(dsq + epssq);
  scale = body[j].mass * dinv * dinv * dinv;
  ax += dx * scale;
  ay += dy * scale;
  az += dz * scale;
}
```

Loop Duplication Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 0.55
  - CC 2.0: 1.00
  - CC 3.0: 1.00
- Performance
  - No change for 2.0 & 3.0
    - Divergence moved to loop
  - 45% slowdown for CC 1.3
    - Unclear reason
- Discussion
  - Not a useful optimization
  - Code bloat
  - A little divergence is okay (only 1 in 3125 iterations)

Blocking using Shared Memory

- Code is memory bound
  - Each warp streams in all bodies' masses and positions
- Use shared memory in inner loop
  - Read block of mass & position info into shared mem
  - Requires barriers (fast hardware barrier within SM)
- Advantage
  - A lot fewer main memory accesses
  - Remaining main memory accesses are fully coalesced (due to usage of scalar arrays)

Blocking using Shared Memory

```
__shared__ float posxs[threads], posys[threads], poszs[…], masss[…];
j = 0;
for (j1 = 0; j1 < nbodiesd; j1 += THREADS) { // first part of loop
  idx = tid + j1;
  if (idx < nbodiesd) { // each thread copies 4 words (fully coalesced)
    posxs[id] = posxd[idx];  posys[id] = posyd[idx];
    poszs[id] = poszd[idx];  masss[id] = massd[idx];
  }
  __syncthreads(); // wait for all copying to be done
  bound = min(nbodiesd - j1, THREADS);
  for (j2 = 0; j2 < bound; j2++, j++) { // second part of loop
    if (i != j) {
      dx = posxs[j2] – px;  dy = posys[j2] – py;  dz = poszs[j2] - pz;
      dsq = dx*dx + dy*dy + dz*dz;
      dinv = rsqrtf(dsq + epssqd);
      scale = masss[j2] * dinv * dinv * dinv;
      ax += dx * scale;  ay += dy * scale;  az += dz * scale;
    }
  }
  __syncthreads(); // wait for all force calculations to be done
}
```

Blocking Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 3.7
  - CC 2.0: 1.1
  - CC 3.0: 1.6
- Performance
  - Great speedup for CC 1.3
  - Some speedup for others
    - Has hardware data cache
- Discussion
  - Very important optimization for memory bound code
  - Even with L1 cache

Loop Unrolling

- CUDA compiler
  - Generally good at unrolling loops with fixed bounds
  - Does not unroll inner loop of our example code
- Use pragma to unroll (and pad arrays)

```
#pragma unroll 8
for (j2 = 0; j2 < bound; j2++, j++) {
  if (i != j) {
    dx = posxs[j2] – px;  dy = posys[j2] – py;  dz = poszs[j2] - pz;
    dsq = dx*dx + dy*dy + dz*dz;
    dinv = rsqrtf(dsq + epssqd);
    scale = masss[j2] * dinv * dinv * dinv;
    ax += dx * scale;  ay += dy * scale;  az += dz * scale;
  }
}
```

Loop Unrolling Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 1.07
  - CC 2.0: 1.16
  - CC 3.0: 1.07
- Performance
  - Insignificant speedup
  - All three GPUs
- Discussion
  - Can be useful
  - May increase register usage, which may lower maximum number of threads per block and result in slowdown

CC 2.0 Absolute Performance

- Problem size
  - n=100000, step=1
- Runtime
  - 612 ms
- FP operations
  - 326.7 GFlop/s
- Main mem throughput
  - 1.035 GB/s

- Not peak performance
  - Only 32% of 1030 GFlop/s
    - Peak assumes FMA every cycle
  - 3 sub (1c), 3 fma (1c), 1 rsqrt (8c), 3 mul (1c), 3 fma (1c) = 20c for 20 Flop
  - 63% of realistic peak of 515.2 GFlop/s
    - Assumes no non-FP operations
  - With int ops = 31c for 20 Flop
  - 99% of actual peak of 330.45 GFlop/s

Eliminating the If Statement

- Algorithmic optimization
  - Potential softening parameter avoids division by zero
  - If-statement is not necessary and can be removed
    - Eliminates thread divergence

```
for (j2 = 0; j2 < bound; j2++, j++) {
  if (i != j) {
    dx = posxs[j2] – px;  dy = posys[j2] – py;  dz = poszs[j2] - pz;
    dsq = dx*dx + dy*dy + dz*dz;
    dinv = rsqrtf(dsq + epssqd);
    scale = masss[j2] * dinv * dinv * dinv;
    ax += dx * scale;  ay += dy * scale;  az += dz * scale;
  }
}
```

If Elimination Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 1.38
  - CC 2.0: 1.54
  - CC 3.0: 1.64
- Performance
  - Large speedup
  - All three GPUs
- Discussion
  - No thread divergence
  - Allows compiler to schedule code much better

Rearranging Terms

- Generated code is suboptimal
  - Compiler does not emit as many fused multiply-add (FMA) instructions as it could
  - Rearrange terms in expressions to help compiler
    - Need to check generated assembly code

```
for (j2 = 0; j2 < bound; j2++, j++) {
  dx = posxs[j2] – px;  dy = posys[j2] – py;  dz = poszs[j2] - pz;
  dsq = dx*dx + dy*dy + dz*dz;
  dinv = rsqrtf(dsq + epssqd);
  dsq = dx*dx + (dy*dy + (dz*dz + epssqd));
  dinv = rsqrtf(dsq);
  scale = masss[j2] * dinv * dinv * dinv;
  ax += dx * scale;  ay += dy * scale;  az += dz * scale;
}
```

FMA Speedup

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 1.03
  - CC 2.0: 1.05
  - CC 3.0: 1.06
- Performance
  - Small speedup
  - All three GPUs
- Discussion
  - Seemingly needless transformations may make a difference

Higher Unroll Factor

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 1.01
  - CC 2.0: 1.04
  - CC 3.0: 0.93
- Unroll 128 times
  - Avoid looping overhead
  - Now that there are no *if*s
- Performance
  - Little speedup/slowdown
- Discussion
  - Carefully choose unroll factor (manually tune)

Compiler Flags

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 1.00
  - CC 2.0: 1.18
  - CC 3.0: 1.15

- -use_fast_math
  - "-ftz=true" suffices (flush denormals to zero)
  - Makes SP FP operations faster except on CC 1.3
- Performance
  - Significant speedup
- Discussion
  - Use faster but less precise operations when prudent

Final Absolute Performance

- CC 2.0 Fermi GTX 480
  - Problem size
    - n=100000, step=1
  - Runtime
    - 296.1 ms
  - FP operations
    - 675.6 GFlop/s (SP)
    - 66% of peak performance
    - 261.1 GFlops/s (DP)
  - Main mem throughput
    - 2.139 GB/s
- CC 3.0 Kepler GTX 680
  - Problem size
    - n=300000, step=1
  - Runtime
    - 1073 ms
  - FP operations
    - 1677.6 GFlop/s (SP)
    - 54% of peak performance
    - 88.7 GFlops/s (DP)
  - Main mem throughput
    - 5.266 GB/s

Hybrid Execution

- CPU always needed for program launch and I/O
  – CPU much faster on serial program segments
- GPU 10 times faster than CPU on parallel code
  – Running 10% of problem on CPU is hardly worthwhile
  – Complicates programming and requires data transfer
    - Best CPU data structure is often not best for GPU
- PCIe bandwidth much lower than GPU bandwidth
  – 1.6 to 6.5 GB/s versus 192 GB/s
  – But can send data while CPU & GPU are computing
  – Merging CPU and GPU on same die (e.g., AMD's Fusion APU) makes finer grain switching possible

Summary

- Step-by-step porting and tuning of CUDA code
  - Example: n-body simulation
- GPUs have very powerful hardware
  - But only exploitable with some code
  - Harder to program and optimize than CPU hardware