



CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

CUDA

Advanced Techniques 3

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>

Some slides are used and slightly
modified from:

NVIDIA teaching kit



In This Lecture ...

- Error Handling
- More about performance
- A common parallel pattern
- Atomic instructions in CUDA
- Memory Fencing
- Computational thinking 101
- A Full Example

Error Handling in *CUDA*

```
__global__ void foo(int *ptr)
{
    *ptr = 7;
}

int main(void)
{
    foo<<<1,1>>>(0);
    return 0;
}
```

What will happen when you compile and execute this piece of code?

Error Handling

- In a CUDA program, if we suspect an error has occurred during a kernel launch, then we **must explicitly check** for it after the kernel has executed.
- CUDA runtime will respond to questions ... But won't talk without asked!

`cudaError_t cudaGetLastError(void);`

- Called by the host
- returns a value encoding the kind of the last error it has encountered
- check for the error only after we're sure a kernel has finished executing → don't forget kernel calls are async!
 - What will you do?

```
#include <stdio.h>
#include <stdlib.h>
```

```
__global__ void foo(int *ptr)
{
    *ptr = 7;
}
```

```
int main(void)
{
    foo<<<1,1>>>(0);
```

```
    // make the host block until the device is finished with foo
    cudaDeviceSynchronize();
```

```
    // check for error
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess)
    {
        // print the CUDA error message and exit
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }
```

```
    return 0;
}
```

Same Technique with Synchronous Calls

```
cudaError_t error = cudaMalloc((void**)&ptr,  
                                10000000000000);  
  
if(error != cudaSuccess)  
{  
    // print the CUDA error message and exit  
    printf("CUDA error: %s\n",  
          cudaGetErrorString(error));  
    exit(-1);  
}
```

The output will be:

CUDA error: out of memory

Rules of Thumb

- Do not use `cudaDeviceSynchronize()` a lot in your code because it has a large performance penalty.
- You can enable it during debugging and disable it otherwise.

#ifdef DEBUG

```
cudaThreadSynchronize();  
cudaError_t error = cudaGetLastError();  
if(error != cudaSuccess)  
{  
    printf("CUDA error at %s:%i: %s\n", filename, line_number, cudaGetErrorString(error));  
    exit(-1);  
}
```

#endif

If debugging, compile with:
\$ nvcc **-DDEBUG** mycode.cu

More About Performance

Hardware configuration can be safely ignored when designing a software for correctness but must be considered in the code structure when designing for peak performance.

Throughput

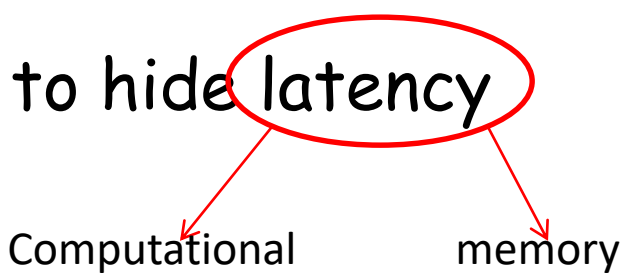
Latency

Some Insights About Performance

Occupancy

Utilization





It is a common belief that ...

- More threads is better
 - because it needs more threads to hide latency
- 
- ```
graph TD; latency((latency)) --> Computational[Computational]; latency --> memory[memory];
```





But is it always true?

## CUDA Basic Linear Algebra Subroutines

Multiplication of two large matrices, single precision (SGEMM):

|                   | CUBLAS 1.1                                                                            | CUBLAS 2.0                                                                              |                          |
|-------------------|---------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|--------------------------|
| Threads per block | 512  | 64   | 8x smaller thread blocks |
| Occupancy (G80)   | 67%  | 33%  | 2x lower occupancy       |
| Performance (G80) | 128 Gflop/s                                                                           | 204 Gflop/s                                                                             | 1.6x higher performance  |

Batch of 1024-point complex-to-complex FFTs, single precision:

|                   | CUFFT 2.2                                                                              | CUFFT 2.3                                                                                |                          |
|-------------------|----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|--------------------------|
| Threads per block | 256  | 64   | 4x smaller thread blocks |
| Occupancy (G80)   | 33%  | 17%  | 2x lower occupancy       |
| Performance (G80) | 45 Gflop/s                                                                             | 93 Gflop/s                                                                               | 2x higher performance    |

# Latency Vs Throughput

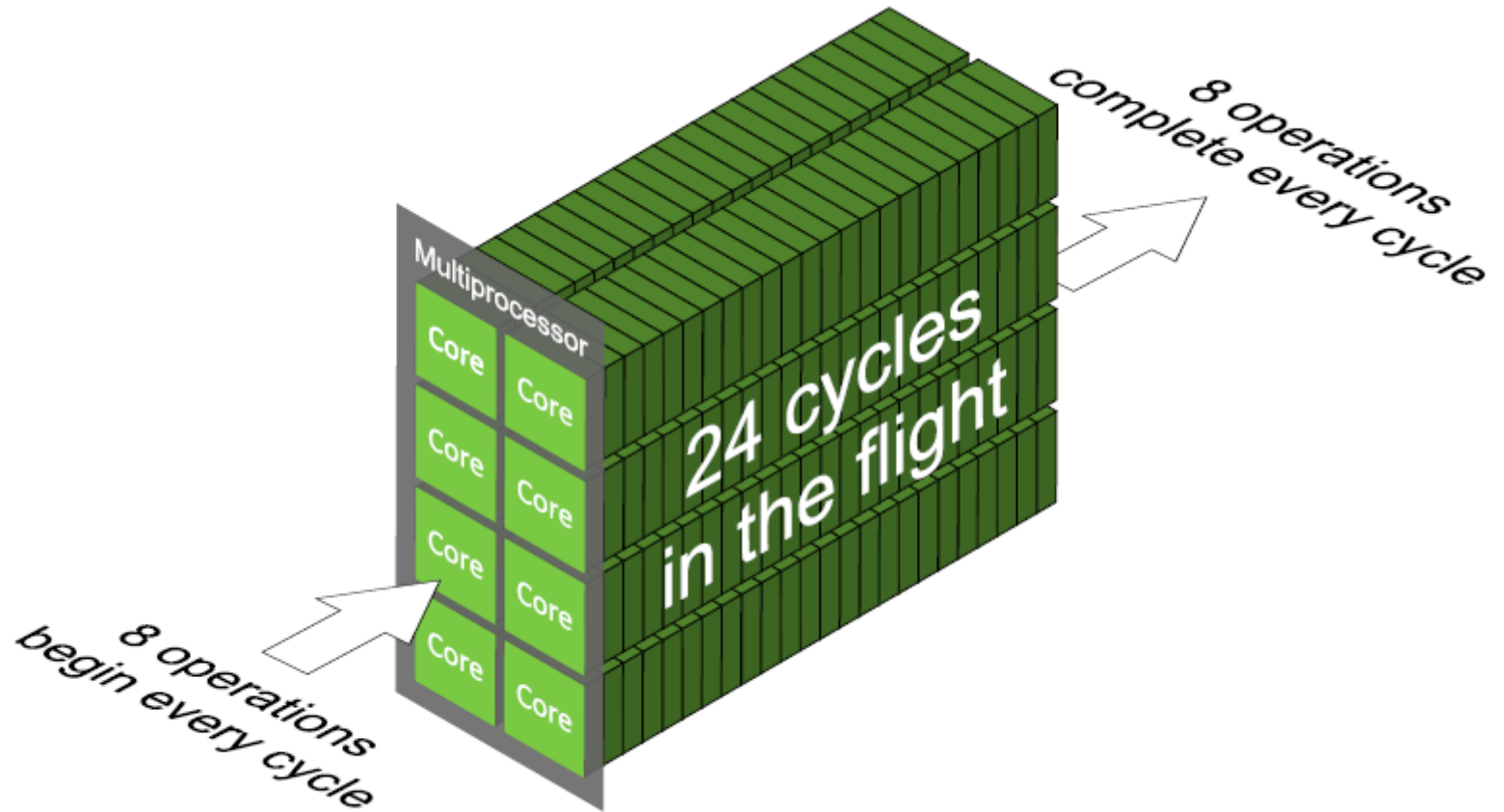
- Latency (how much time) is **time**
  - instruction takes 4 cycles per warp
  - memory takes 400 cycles
- Throughput (how many operations per cycle or second) is **rate**
  - Arithmetic:  $1.3 \text{ Tflop/s} = 480 \text{ ops/cycle}$  (op=multiply-add)
  - Memory:  $177 \text{ GB/s} \approx 32 \text{ ops/cycle}$  (op=32-bit load)

# Hiding Latency is ...

- Doing other operations while waiting
- This will make the kernel runs faster
- **But** not at the peak performance

What can we do??

# Little's Law



Needed parallelism = **Latency** x **Throughput**



# Examples from GPU

| GPU model | Latency<br>(cycles) | Throughput<br>(cores/SM) | Parallelism<br>(operations/SM) |
|-----------|---------------------|--------------------------|--------------------------------|
| G80-GT200 | $\approx 24$        | 8                        | $\approx 192$                  |
| GF100     | $\approx 18$        | 32                       | $\approx 576$                  |
| GF104     | $\approx 18$        | 48                       | $\approx 864$                  |

Average latency of a computational operation



Less operations means idle cycle



# So ...

- Higher performance does not mean more threads but **higher utilization**
- Utilization is related to **parallelism**
- We can increase utilization by
  - increasing throughput
    - Instruction level parallelism
    - Thread level parallelism
  - decreasing latency

**Occupancy is not utilization, but one of the contributing factors.**

# Occupancy Calculator API

```
__global__ void MyKernel(int *d, int *a, int *b) {
 int idx = threadIdx.x + blockIdx.x * blockDim.x;
 d[idx] = a[idx] * b[idx];
}

int main() {
 int numBlocks;
 int blockSize = 32;
 int device;
 cudaDeviceProp prop;
 int activeWarps;
 int maxWarps;

 cudaGetDevice(&device);
 cudaGetDeviceProperties(&prop, device);

 cudaOccupancyMaxActiveBlocksPerMultiprocessor(
 &numBlocks,
 MyKernel,
 blockSize,
 0);

 activeWarps = numBlocks * blockSize / prop.warpSize;
 maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;
}
```

## cudaOccupancyMaxActiveBlocksPerMultiprocessor

- From CUDA 6.5
- Produces an occupancy prediction based on:
  - the block size
  - shared memory usage of a kernel
- Reports occupancy in terms of the number of concurrent thread blocks per multiprocessor
- Don't forget: it is just a prediction!
- Arguments:
  1. pointer to an integer (where #blocks will be reported)
  2. kernel
  3. block size
  4. dynamic shared memory per block in bytes

# How about memory?

maximizing overall memory throughput for  
the application

=

minimize data transfers  
with low bandwidth

host  $\leftrightarrow$  device

Global mem access

# This means ...Typically

1. Load data from device memory to shared memory.
2. Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads.
3. Process the data in shared memory.
4. Synchronize again if necessary to make sure that shared memory has been updated with the results.
5. Write the results back to device memory.

But accessing global memory is a  
necessary evil ... So:

- Can we apply the same technique (i.e. Little's law) to memory?

Needed parallelism = **Latency** x **Throughput**

|            | <b>Latency</b>   | <b>Throughput</b> | <b>Parallelism</b> |
|------------|------------------|-------------------|--------------------|
| Arithmetic | ≈18 cycles       | 32 ops/SM/cycle   | 576 ops/SM         |
| Memory     | < 800 cycles (?) | < 177 GB/s        | < 100 KB           |

This means that to hide memory latency you need to keep 100KB in flight.  
But less if the kernel is compute bound!

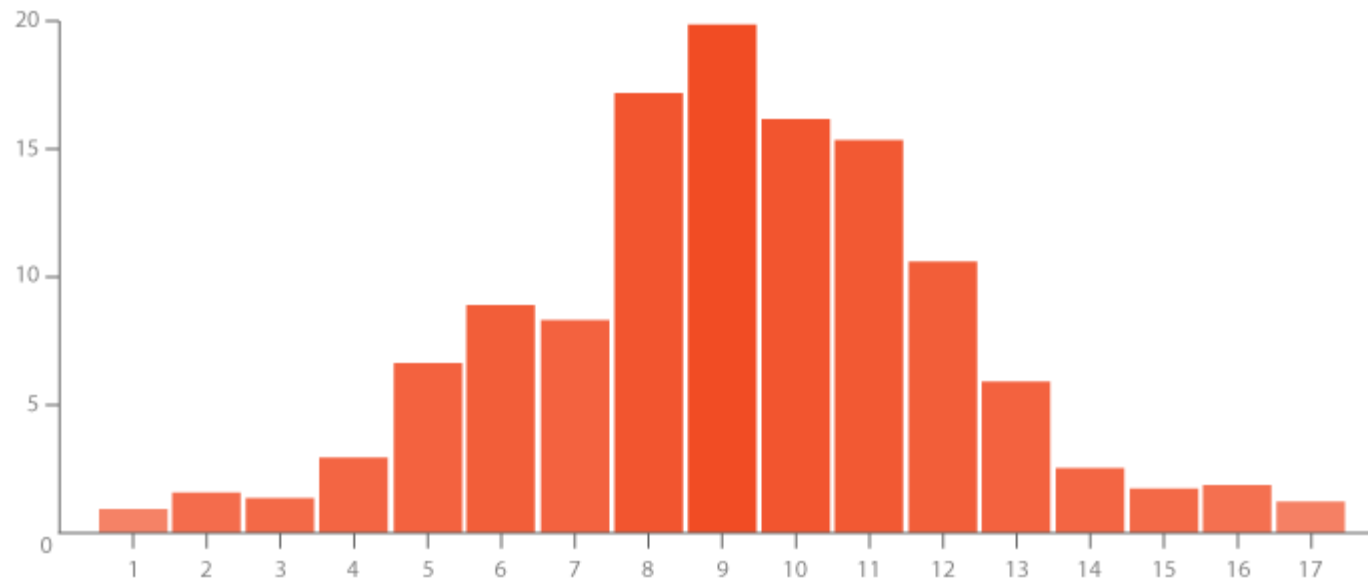
# How Can You Get 100KB From Threads?

- Use more threads
- Use more instructions per thread
- Use more data per thread



*A Common Parallel Pattern*

# Histogram

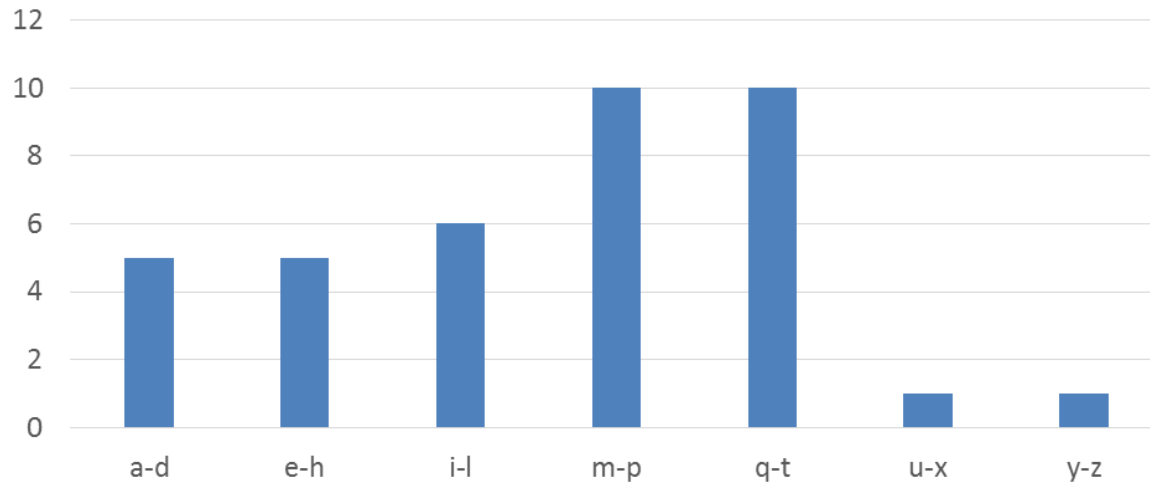


# In a Nutshell

- Important and very useful computation:
  - For each element in the data set, use the value to identify a “bin counter” to increment.
- A good example for understanding output interference in parallel computation

# Example

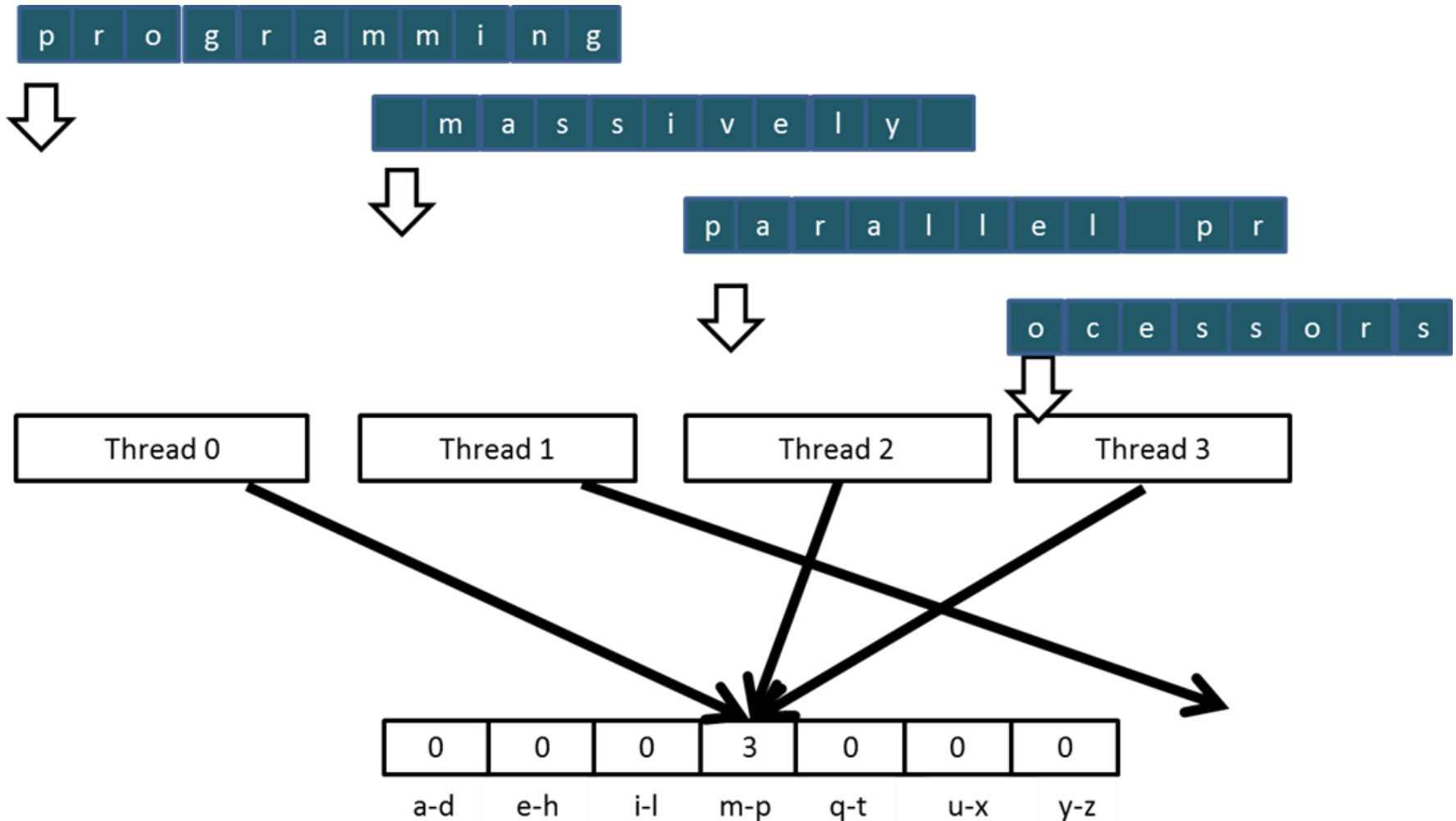
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, ...
- For each character in an input string, increment the appropriate bin counter.
- In the phrase "Programming Massively Parallel Processors" the output histogram is shown below:



# Implementation 1

- Partition the input into sections
- Have each thread to take a section of the input
- Each thread iterates through its section.
- For each letter, increment the appropriate bin counter

# Implementation 1



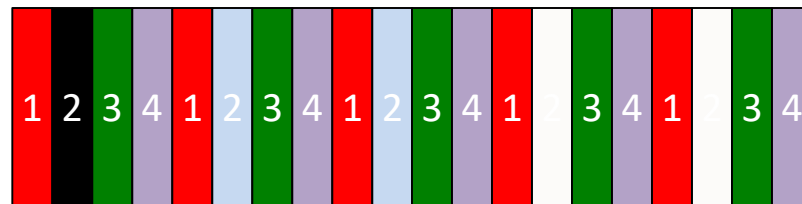
# Evaluation of Implementation 1

- Possible collision
- Poor memory access efficiency:
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized



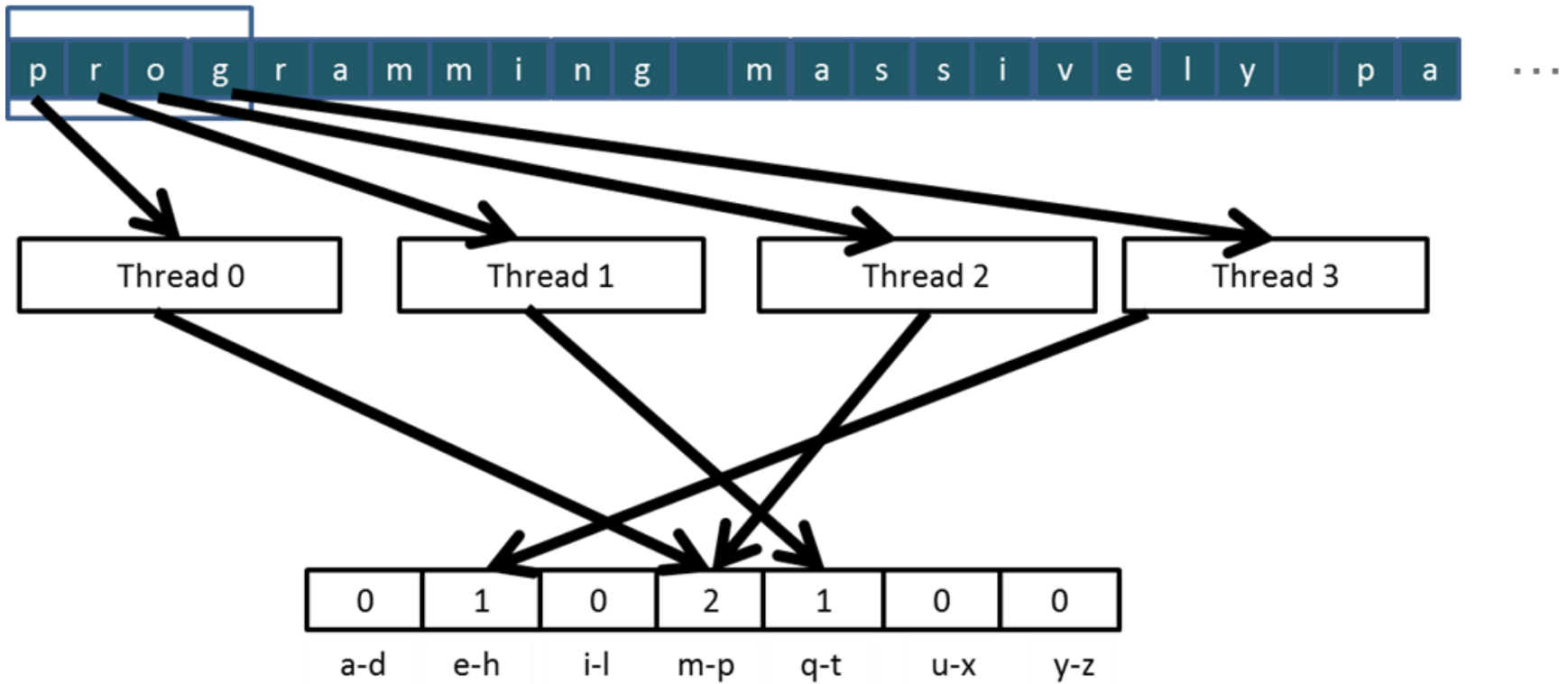
# Implementation 2

- Interleaved partitioning:
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat



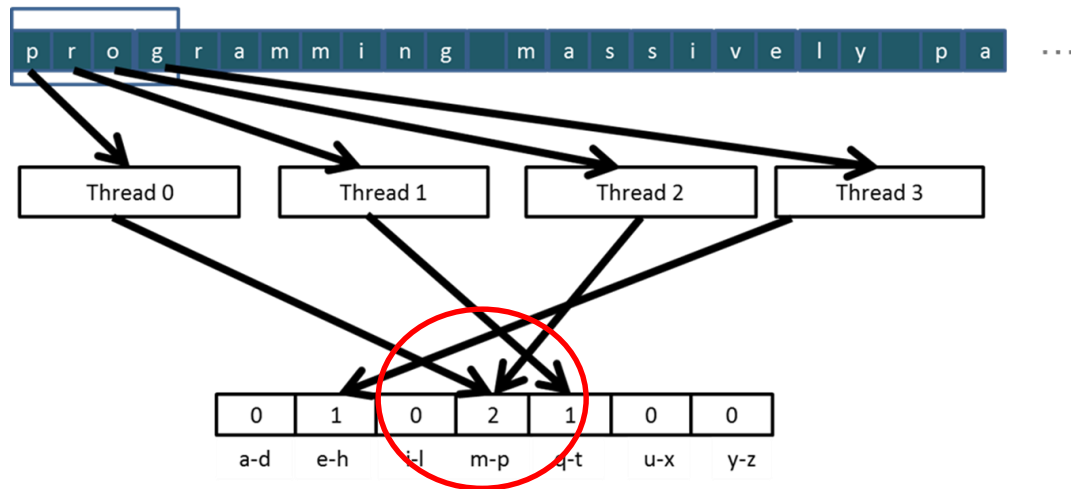


# Implementation 2



# Evaluation of Implementation 2

- + Better memory access patterns
- Still possibility of collision due to data races



# Implementation 3

- We need to deal with data races:
  - read-modify-write operations

thread1: Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

thread2: Old  $\leftarrow$  Mem[x]  
New  $\leftarrow$  Old + 1  
Mem[x]  $\leftarrow$  New

# Implementation 3

- We need **atomic** operation for **read-modify-write**.
- A read-modify-write operation performed by a single hardware instruction on a memory location address
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
  - Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
  - **All threads perform their atomic operations serially on the same location**

# Implementation 3

- Atomic operations in CUDA
  - Atomic add, sub, inc, dec, min, max, exch (exchange)
  - CAS (compare and swap)
    - 3 args: address, compare, val
    - reads a value from address (old value)
    - compute: Old = compare ? val : old
- Example:
  - `int atomicAdd(int* address, int val);`
  - `unsigned int atomicAdd(unsigned int* address, unsigned int val);`
  - `atomicSub, atomicExch, atomicMin, ... atomicAnd, AtomicOr, ...`
- `atomicAdd` for double precision floating-point numbers requires CC 6.0 or higher

# Implementation 3

```
__global__ void histo_kernel(unsigned char *buffer,
 long size, unsigned int *histo)
{
 int i = threadIdx.x + blockIdx.x * blockDim.x;

 // stride is total number of threads
 int stride = blockDim.x * gridDim.x;

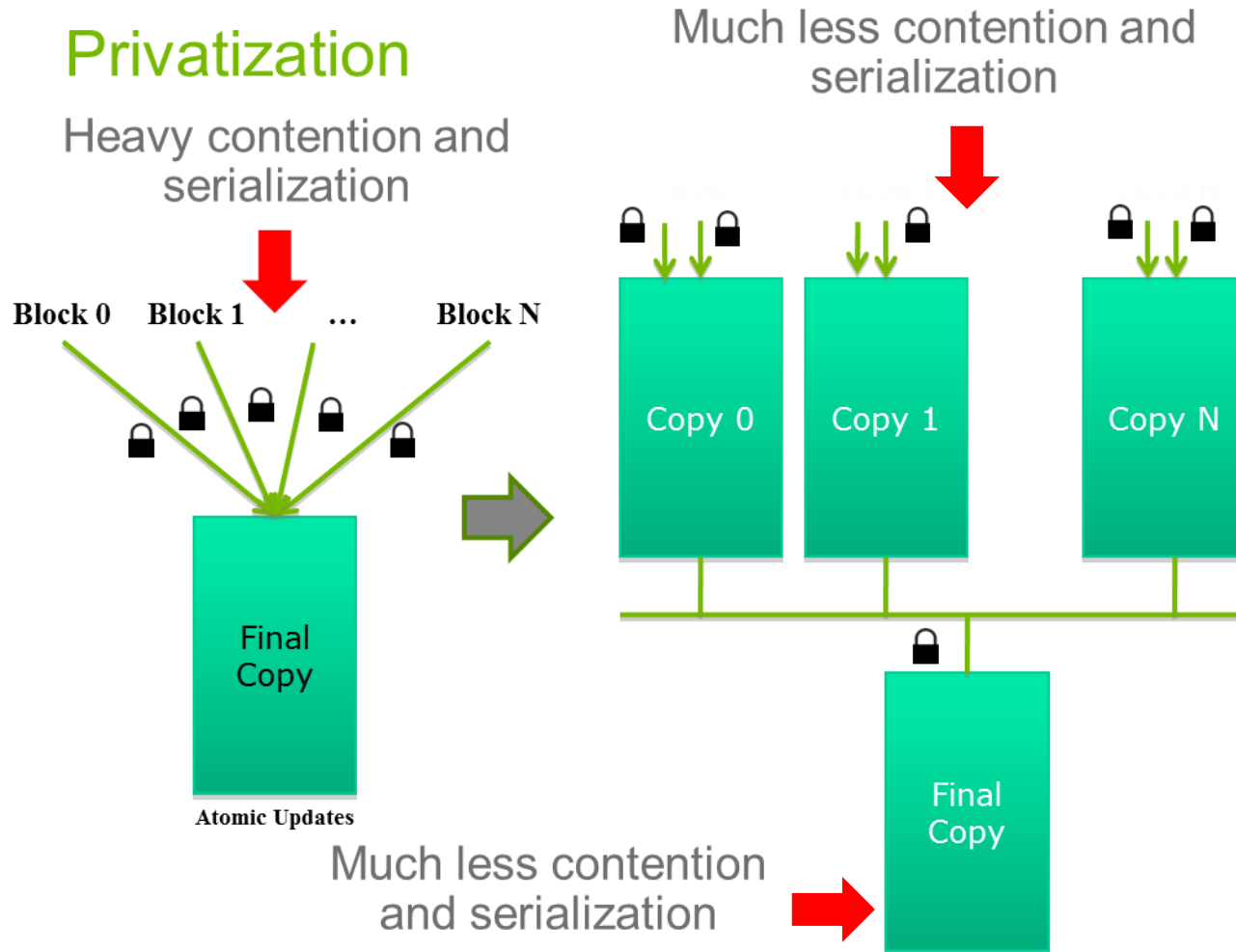
 // All threads handle blockDim.x * gridDim.x
 // consecutive elements
 while (i < size) {
 int alphabet_position = buffer[i] - "a";
 if (alphabet_position >= 0 && alphabet_position < 26)
 atomicAdd(&(histo[alphabet_position/4]), 1);
 i += stride;
 }
}
```

# Evaluation of Implementation 3

- + No data race
- + Coalesced memory access
- Performance loss due to serialization

# Implementation 4

## Privatization





# Implementation 4


- Privatization: a very important use case for shared memory
- Cost
  - Overhead for creating and initializing private copies
  - Overhead for accumulating the contents of private copies into the final copy
- Benefit
  - Much less contention and serialization in accessing both the private copies and the final copy
  - The overall performance can often be improved more than 10x

# Implementation 4

- Create private copies of the histo[] array for each thread block

```
__global__ void histo_kernel(unsigned char *buffer,
 long size, unsigned int *histo)
{
 __shared__ unsigned int histo_private[7];

 if (threadIdx.x < 7) histo_private[threadIdx.x] = 0;
 __syncthreads();
```



Initialize the bin counters in  
the private copies of histo[]

# Implementation 4

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
// stride is total number of threads
int stride = blockDim.x * gridDim.x;
while (i < size) {
 atomicAdd(&(amp;private_histo[buffer[i]/4]), 1);
 i += stride;
}
```



Build the private histogram

# Implementation 4

```
// wait for all other threads in the block to finish
__syncthreads();
```

```
if (threadIdx.x < 7) {
 atomicAdd(&(histo[threadIdx.x]),
 private_histo[threadIdx.x]);
}
}
```

Build the final histogram



# About Privatization

- Privatization is a powerful and frequently used technique for parallelizing applications
- The operation needs to be associative and commutative
- The private histogram size needs to be small
  - Fits into shared memory
- What if the histogram is too large to privatize?
  - Sometimes one can partially privatize an output histogram and use range testing to go to either global memory or shared memory

# What we learned from the histogram example

- **Atomic operations** may be needed → sacrificing some performance for correctness
- **Privatization** can sometimes reduce the performance loss due to serialization caused by atomic operations.

# Atomic Operations in CUDA

# Definition

An atomic function performs a **read-modify-write** atomic operation on one 32-bit or 64-bit word residing in **global or shared memory**.

**Atomic** = guaranteed to be performed without interference from other threads  
= no other thread can access this address until the operation is complete

**Atomic functions can only be used in device functions.**



# Atomic Operations Do not Imply:

- synchronization
- ordering constraints for memory operations

# Be Careful

- **Before CC < 6.x**
  - Atomic operations done from the GPU are atomic only with respect to that GPU.
  - GPU's atomic operation to a peer GPU's memory appears as a regular read followed by a write to the peer GPU, and the two operations are not done as one single atomic operation.
  - Atomic operations from the GPU to CPU memory will not be atomic with respect to the CPU.

# Be Careful

- From CC 6.x:
  - New type of atomics which allows developers to **widen or narrow the scope** of an atomic operation
  - Example "widen": **atomicAdd\_system** guarantees that the instruction is atomic with respect to other CPUs and GPUs in the system.
  - Example "narrow": **atomicAdd\_block** implies that the instruction is atomic only with respect to atomics from other threads in the same thread block.

# atomicCAS(int \*p, int cmp, int y)

## Compare And Swap

```
atomic
{
 if (*p == cmp)
 *p = y;
 return *p;
}
```

Instead of int, you can use unsigned, long, long long, float, double, ... .

# int atomicOP(int \*p, int v)

```
atomic
{
 int old = *p;
 *p = old OP v;

 return old;
}
```

OP can be:

- Add
- Sub
- Min
- Max
- Inc [ $*p = (old \geq v) ? 0 : (old+1)$ ]
- Dec [ $*p = (old \geq v) ? 0 : (old-1)$ ]
- And
- Or
- Xor

Instead of int, you can use unsigned, long, long long, float, double, ... .

# int atomicExch(int \*p, int v)

```
atomic
{
 int old = *p;
 *p = v;

 return old;
}
```

Instead of int, you can use unsigned, long, long long, float, double, ... .

# Memory Fencing

Suppose that:

Thread 1 executes this

And

Thread 2 executes this

```
__device__ volatile int X = 1, Y = 2;
```

```
__device__ void writeXY() {
 X = 10;
 Y = 20; }
```

**Q:** What are the possible values that A and B can take?

**A:** In **strongly-ordered** memory model we have 3 possibilities:

- A = 1 B = 2
- A = 10 B = 2
- A = 10 B = 20

```
__device__ void readXY() {
 int A = X;
 int B = Y; }
```

But in a **weakly-ordered** memory model, beside the above possibilities thread 2 can see:

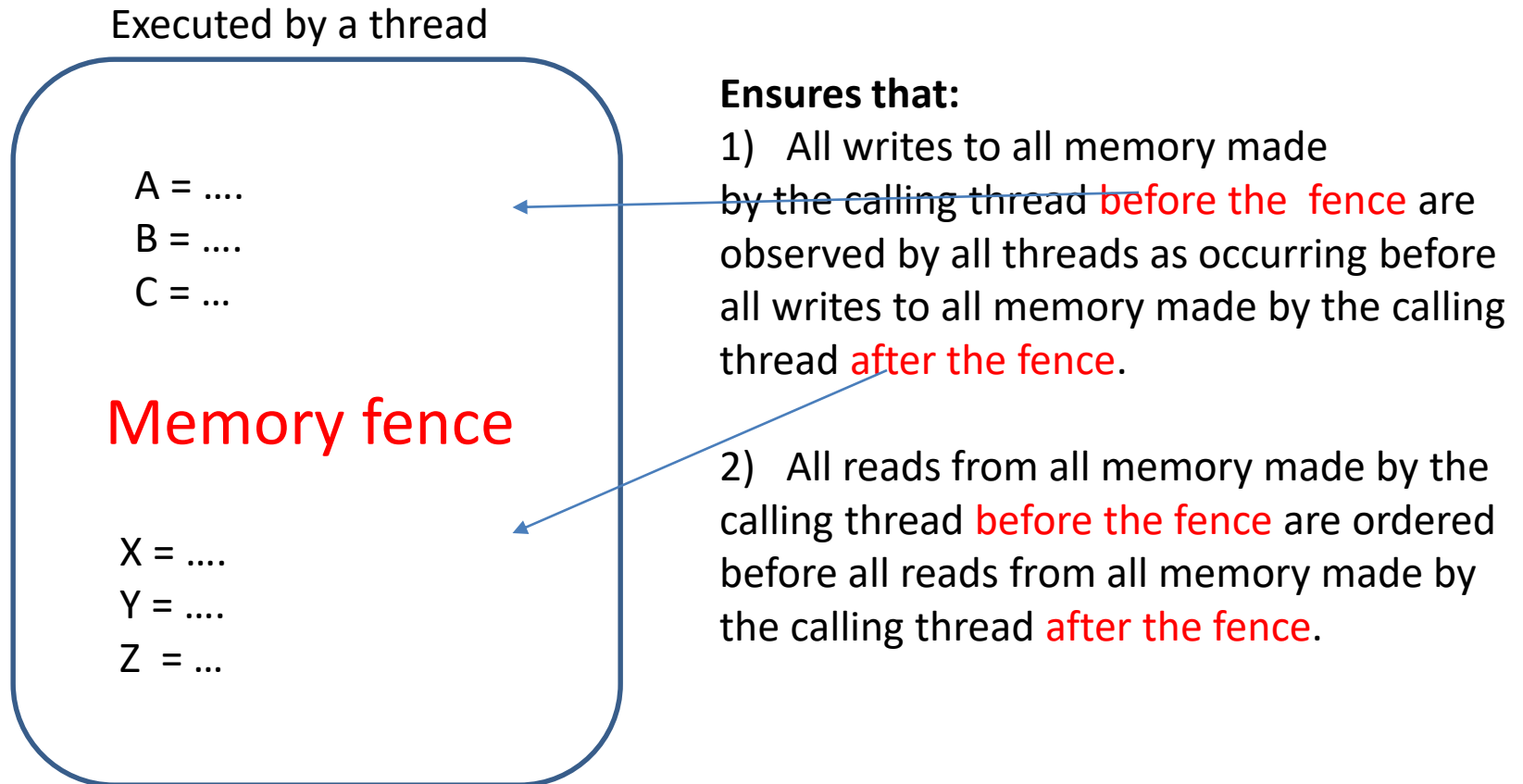
- A = 1 B = 20



# CUDA Memory Model

- The CUDA programming model assumes a **device with a weakly-ordered memory model**.
- **Definition:** the order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread.

# How to deal with this situation?



# Three Flavors of Fencing

`void __threadfence_block();`

Ensures 1) and 2) for threads in the same block.

`void __threadfence();`

Ensures 1) and 2) for all threads in the block and ordered writes (before/after fence) for all threads in the device.

`void __threadfence_system();`

For all peer devices and CPUs.

# Computational Thinking 101

# Computational Thinking 101

***Computational Thinking is arguably the most important aspect of parallel Application development!***

J. Wing Communications of the ACM, 49(3), 2006

## **What is it?**

Decomposing a domain problem into well-defined, coordinated work units that can Each be realized with different numerical methods and well-known algorithms.

# Why Do We Need Parallel Computing in the First place?

To solve a given problem in less time

To solve bigger problems

To achieve better solutions for a given problem and a given amount of time



Increased Speed!

The diagram consists of three text blocks on the left, each with a red arrow pointing to a central red oval on the right. The oval contains the text 'Increased Speed!' in red. The arrows originate from the right side of each text block and point towards the left side of the oval.

# Applications that are good candidates for parallel computing:

- Involve large problem sizes
- Involve high modeling complexity

Formulating the problem is crucial!!



The problem must be formulated in a such a way that it can be decomposed into subproblems that can be executed in parallel.

# The Process of Parallel Programming

1. Problem decomposition
  2. Algorithm selection
  3. Implementation in a language
  4. Performance tuning
- This is what we have been doing till now!

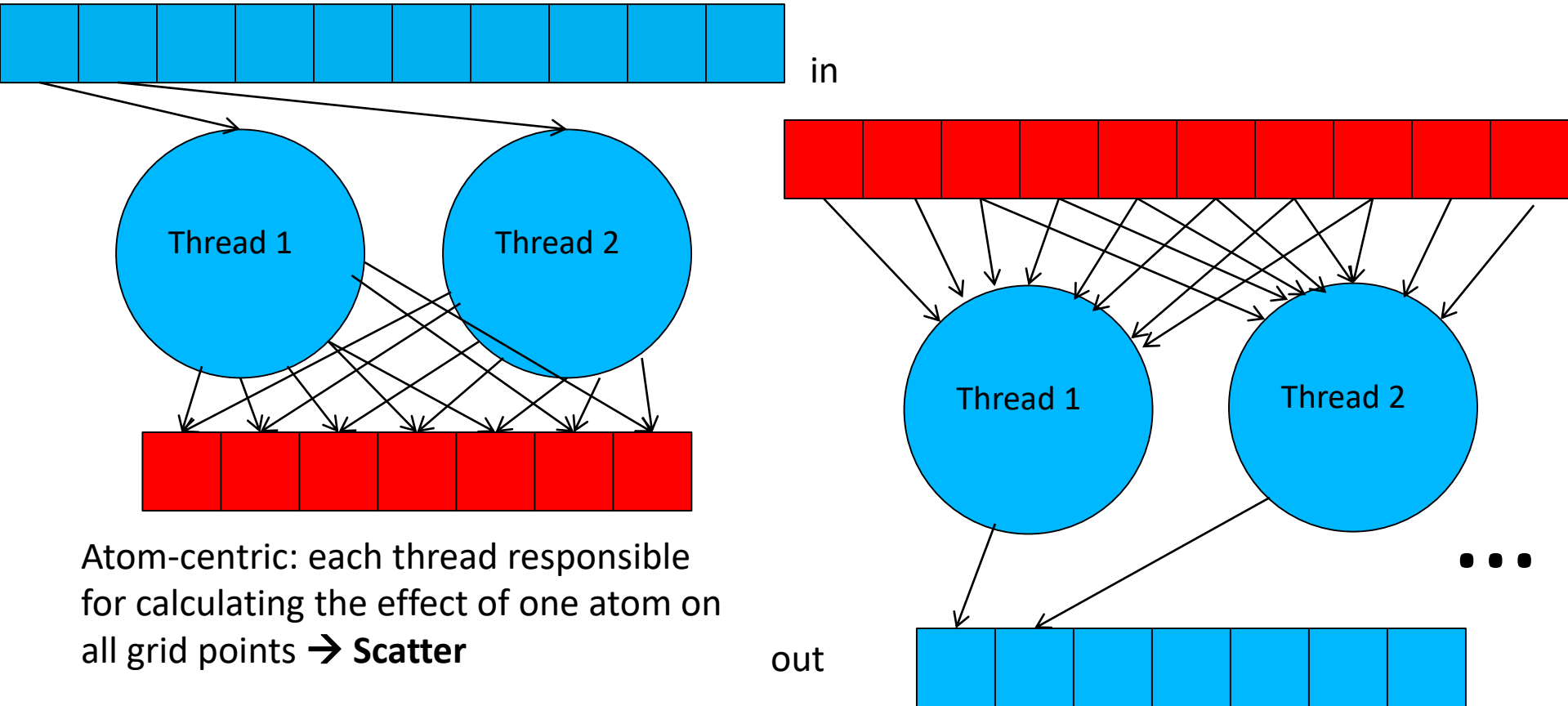


# Problem Decomposition

Identify the work to be performed  
by each unit of parallel execution →  
thread in CUDA

# Problem Decomposition: Thread Arrangement

Example: Electrostatic Map Problem



Atom-centric: each thread responsible for calculating the effect of one atom on all grid points → **Scatter**

Grid-centric: each thread calculates the effect of all atoms on a grid point → **Gather**

# Which is better?

- Gather is desirable
  - Threads can accumulate their results in their private registers.
  - Multiple threads share input atom values.

# Problem Decomposition

- Picking the best thread arrangement requires the understanding of the underlying hardware.
- A real application consists of several modules that work together
  - Amount of work per module can vary dramatically
  - You need to decide if a module is worth implementing in CUDA
- Amdahl's law

# Algorithm Selection

- An algorithm must exhibit three essential properties:
  - definiteness = no ambiguity
  - effective computability = each step can be carried by a computer
  - finiteness = guaranteed to terminate
- When comparing several algorithms, take the following factors into account:
  - Steps of computation
  - Degree of parallel execution
  - Numerical stability
  - Memory bandwidth

Skills needed to go  
from: Parallel Programmer  
to: Computational Thinker

- Computer Architecture
- Programming models and compilers
- Algorithmic techniques: (e.g. tiling)
- Domain knowledge

# So

- Computational thinking is an art but a very crucial one.
- Jumping from problem definition to coding right away is the worst thing you can do!

A Full Example



# What will we do?

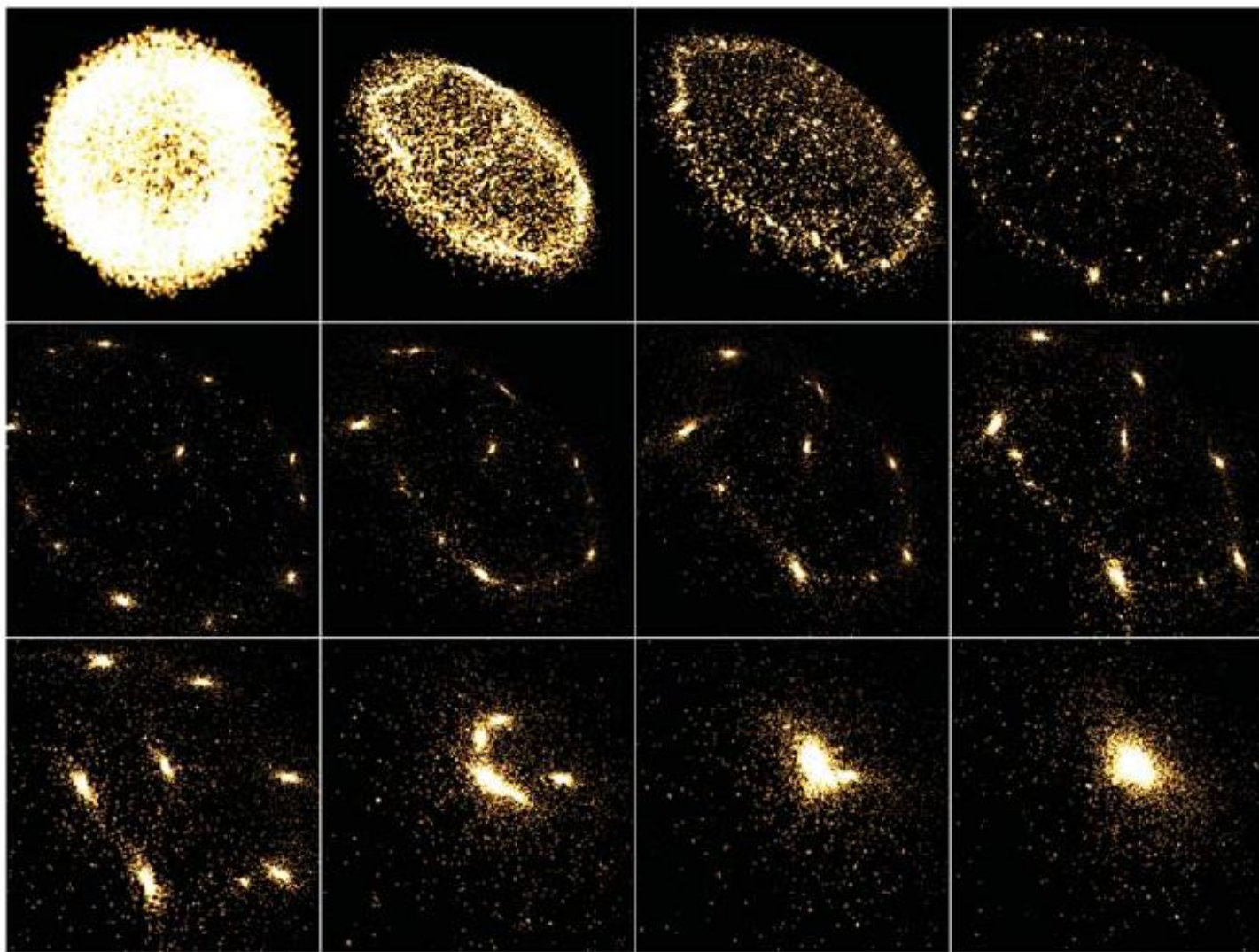
We will pick a problem, **analyze** it, and see how it can be **written** and **optimized** for GPU.



- Minimize memory access
- Minimize thread divergence
- Fully parallelized

# N-Body Problem

An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body.






**Frames from an Interactive 3D Rendering of a 16,384-Body System**

# N-Body Problem

- Manifests itself in many domains: physics, astronomy, electromagnetics, molecules, etc.
- N points
- The answer at each point depends on data at all the other points
- $O(n^2)$
- To reduce complexity: compress data of groups of nearby points
  - A well-known algorithm to do this: Barnes Hut

# Challenges of CUDA

## Implementation of Barnes Hut

- Repeatedly builds and traverse an irregular tree-based data structure.  • Results in thread divergence
- Performs a lot of pointer-chasing memory operations.  • Many slow uncoalesced accesses
- Typically expressed recursively.  • Must use iterations

# Barnes Hut n-Body Algorithm

Divided into 3 steps

1. Building the tree -  $O(n * \log n)$
2. Computing cell centers of mass -  $O(n)$
3. Computing Forces -  $O(n * \log n)$

# Barnes Hut n-Body Algorithm

0. Read input data and transfer to GPU

for each timestep do {

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into octree
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

7. Transfer result to CPU and output

**Executed on GPU**

# Barnes Hut n-Body Algorithm

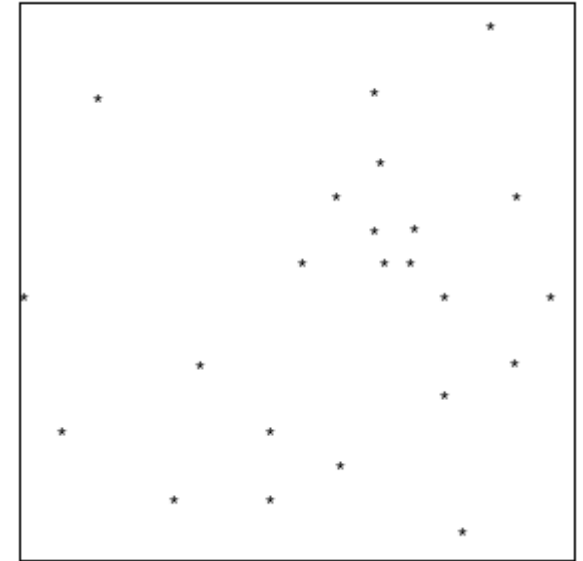
0. Read input data and transfer to GPU

for each timestep do {

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into octree
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

7. Transfer result to CPU and output





# Barnes Hut n-Body Algorithm

0. Read input data and transfer to GPU

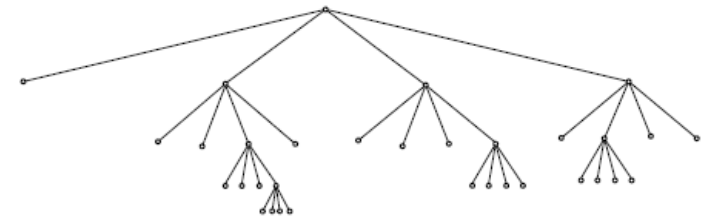
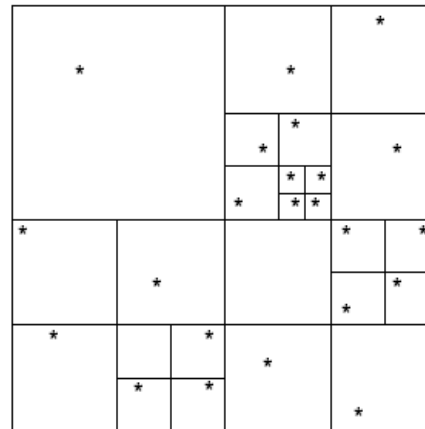
for each timestep do {

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into octree
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

7. Transfer result to CPU and output

**cells:** Internal tree nodes  
**Bodies:** leaves



# Barnes Hut n-Body Algorithm

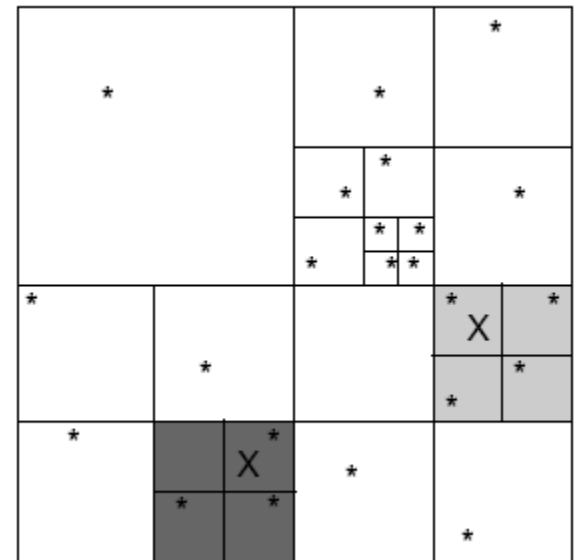
0. Read input data and transfer to GPU

for each timestep do {

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into octree
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

7. Transfer result to CPU and output



Calculate for each cell:

- Center of gravity
- Cumulative mass

# Barnes Hut n-Body Algorithm

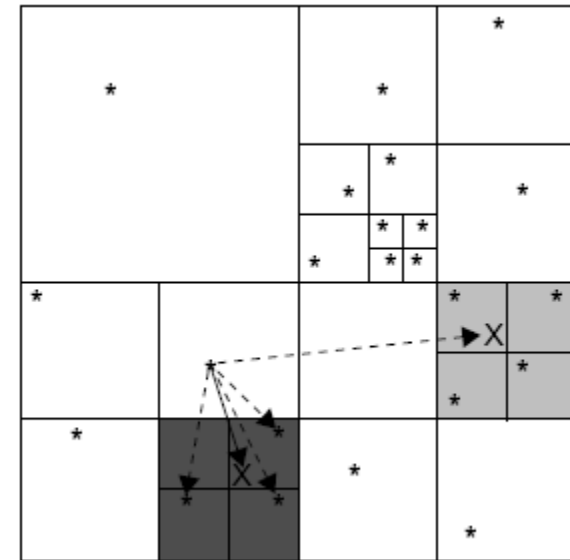
0. Read input data and transfer to GPU

for each timestep do {

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into octree
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

7. Transfer result to CPU and output



Kernel 4 is not needed for correctness but for optimization.

- It is done by in-order traversal of the tree.
- Typically places spatially close bodies close together.

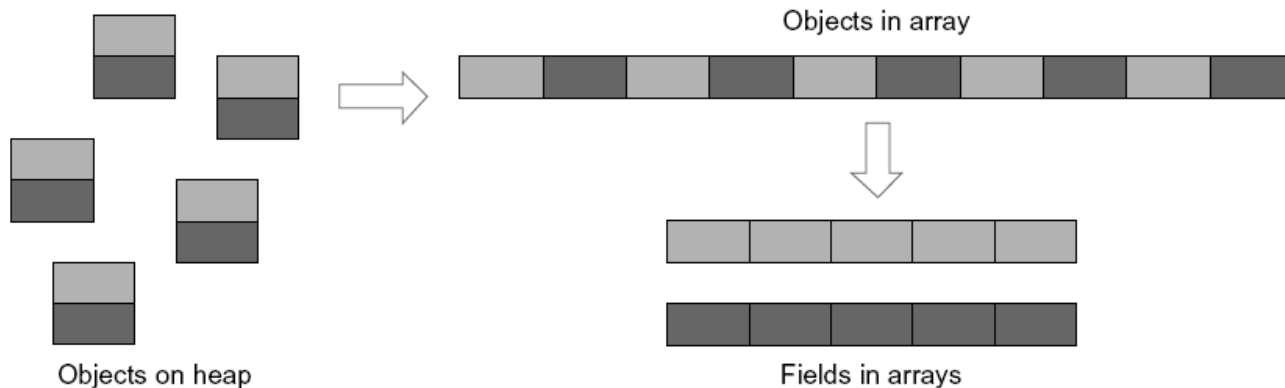
# First Step: Data Structure

- Dynamic data structures like trees are usually built using heap objects.
- Is it the best way to go?
- Drawbacks:
  - Access to heap objects is slow
  - Very hard to coalesce objects with multiple fields

How do we deal with this?

# First Step: Data Structure

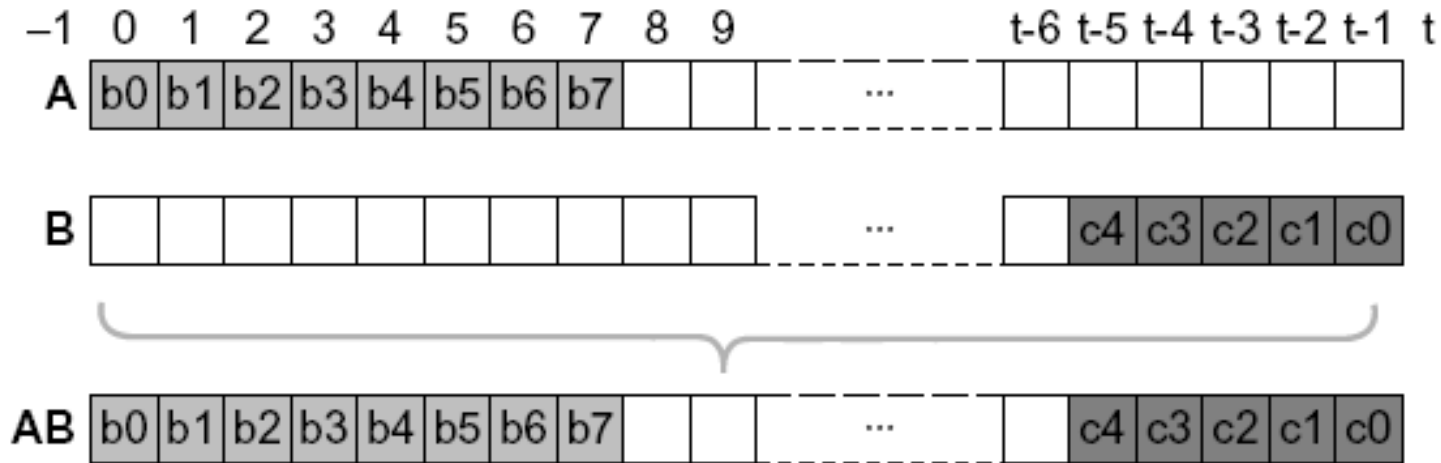
- Use an array-based data structure
- To be able to coalesce:
  - use several aligned scalar arrays, one per field
- Array indices instead of pointers makes a faster code



# First Step: Data Structure

- Allocate bodies at the beginning and the cells at the end of the arrays
- Use an index of -1 as a "null pointer."
- Advantages.
  - A simple comparison of the array index with the number of bodies determines whether the index points to a cell or a body.
  - In some code sections, we need to find out whether an index refers to a body or to null. Because -1 is also smaller than the number of bodies, a single integer comparison suffices to test both conditions.

# First Step: Data Structure



**b: body**

**c: cell**

**t: array length**

# Threads, Blocks, and Kernels

- The **thread count per block is maximized** and rounded down to the nearest multiple of the warp size for each kernel.
- All kernels **use at least as many blocks as there are streaming multiprocessors in the GPU**, which is automatically detected.
- Because all parameters passed to the kernels, such as the starting addresses of the various arrays, stay the same throughout the time step loop, we **copy them once into the GPU's constant memory**.
  - This is much faster than passing them with every kernel invocation.
- Data transferred from CPU to GPU only at the beginning of the program and at the end.
- code operates on octrees in which nodes can have up to eight children.
  - It contains many loops.
  - Loop unrolling is very handy here.



# Kernel 1

- computes a bounding box around all bodies
  - The root of the octree
  - has to find the minimum and maximum coordinates in the three spatial dimensions
- Implementation:
  - break up the data into equal sized chunks and assigns one chunk to each block
  - Each block then performs a reduction operation
  - The last block combine the results to generate the root node
  - reduction is performed in shared memory in a way that avoids bank conflicts and minimizes thread divergence

# Kernel 2

- Implements an iterative tree-building algorithm that uses **lightweight locks**
- Bodies are assigned to the blocks and threads within a block in round-robin fashion.
- Each thread inserts its bodies one after the other by:
  - traversing the tree from the root to the desired last-level cell
  - attempting to lock the appropriate child pointer (an array index) by writing an otherwise unused value to it using an atomic operation
  - If the lock succeeds, the thread inserts the new body and release the lock

# Kernel 2

- A handy group of functions to use are **atomicxxx** (must use -arch sm\_11 with nvcc)
  - Definition: An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

Synopsis of atomic function **atomicOP(a,b)** is typically

```
t1 = *a; // read
t2 = t1 OP b; // modify
*a = t2; // write
return t;
```

# Kernel 2

- If a body is already stored at this location, the thread:
  - creates a new cell by atomically requesting the next unused array index
  - inserts the original and the new body into this new cell
  - executes a memory fence (*\_\_threadfence*) to ensure the new subtree is visible to the rest of the cores
  - attaches the new cell to the tree
  - releases the lock.

# Kernel 2

*// initialize*

*cell = find\_insertion\_point(body); // nothing is locked, cell cached for retries*

*child = get\_insertion\_index(cell, body);*

*if (child != locked) {*

*if (child == atomicCAS(&cell[child], child, lock)) {*

*if (child == null) {*

*cell[child] = body; // insert body and release lock*

*} else {*

*new\_cell = ...; // atomically get the next unused cell*

*// insert the existing and new body into new\_cell*

*\_\_threadfence(); // make sure new\_cell subtree is visible*

*cell[child] = new\_cell; // insert new\_cell and release lock*

*}*

*success = true; // flag indicating that insertion succeeded*

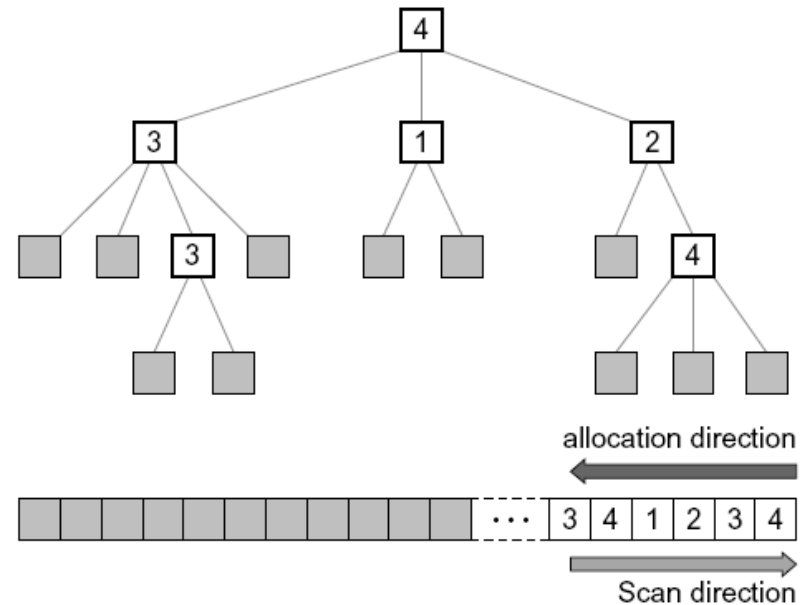
*}*

*}*

*\_\_syncthreads(); // wait for other warps to finish insertion*

# Kernel 3

- traverses the unbalanced octree from the bottom up to compute the center of gravity and the sum of the masses of each cell's children

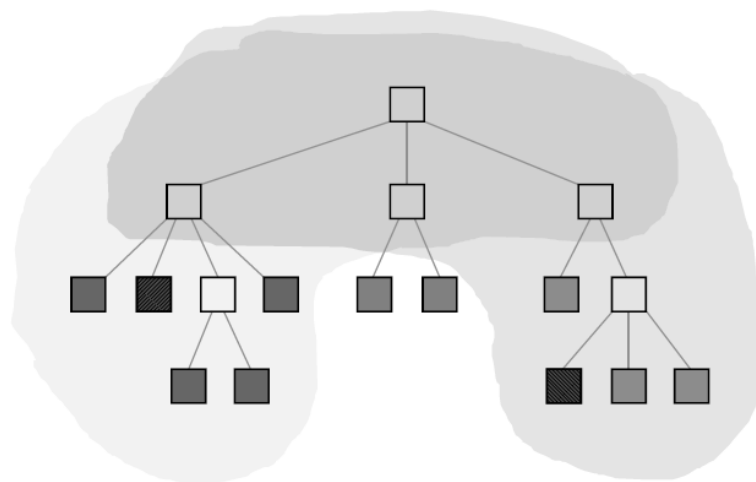


# Kernel 3

- Cells are assigned to blocks and threads in a round-robin fashion.
  - Ensure load-balance
  - Start from leaves so avoid deadlocks
  - Allow some coalescing

# Kernel 5

- Requires the vast majority of the runtime
- For each body, the corresponding thread traverses some prefix of the octree to compute the force acting upon this body.





# Kernel 5

- Optimization: whenever a warp traverses part of the tree that some of the threads do not need, those threads are disabled due to thread divergence.
  - Make the union of the prefixes in a warp as small as possible
    - group spatially nearby bodies together → kernel 4!
- Little computation to hide memory access
  - Optimization: Allow only one thread in a warp to read the pertinent data and cache them in shared memory.

# Summary of Optimizations

## MAIN MEMORY

### **Minimize Accesses**

- Let one thread read common data and distribute data to other threads via shared memory
- When waiting for multiple data items to be computed, record which items are ready and only poll the missing items
- Cache data in registers or shared memory
- Use thread throttling

### **Maximize Coalescing**

- Use multiple aligned arrays, one per field, instead of arrays of structs or structs on heap
- Use a good allocation order for data items in arrays

### **Reduce Data Size**

- Share arrays or elements that are known not to be used at the same time

### **Minimize CPU/GPU Data Transfer**

- Keep data on GPU between kernel calls
- Pass kernel parameters through constant memory

# Summary of Optimizations

## CONTROL FLOW

### Minimize Thread Divergence

- Group similar work together in the same warp

### Combine Operations

- Perform as much work as possible per traversal, i.e., fuse similar traversals

### Throttle Threads

- Insert barriers to prevent threads from executing likely useless work

### Minimize Control Flow

- Use compiler pragma to unroll loops

## LOCKING

### Minimize Locks

- Lock as little as possible (e.g., only a child pointer instead of entire node, only last node instead of entire path to node)

### Use Lightweight Locks

- Use flags (barrier/store and load) where possible
- Use atomic operation to lock but barrier/store or just store to unlock

### Reuse Fields

- Use existing data field instead of separate lock field

# Summary of Optimizations

## HARDWARE

### **Avoid Bank Conflicts**

- Control the accesses to shared memory to avoid bank conflicts

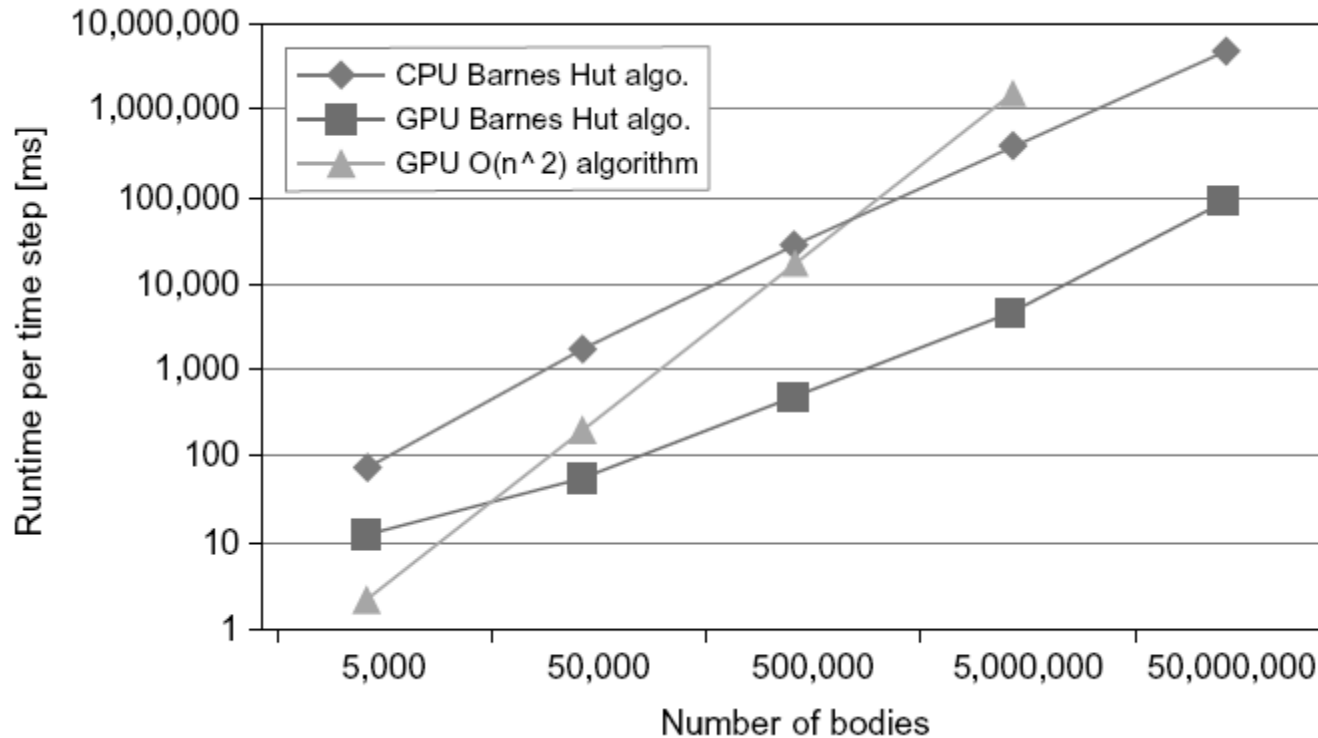
### **Use All Multiprocessors**

- Parallelize code across blocks
- Make the block count at least as large as the number of streaming multiprocessors

### **Maximize Thread Count**

- Parallelize code across threads
- Limit shared memory and register usage to maximize thread count

# Results



**CPU: 2.53GHz Xeon E5540 CPU**

**GPU: 1.3 GHz Quadro FX 5800**

# Conclusions

- Performance is related to how you keep the GPU and its memory busy → does not necessarily mean higher occupancy.
- We looked at some of the common parallel patterns used in many GPU kernels. These are tools that you can use in your own kernels.