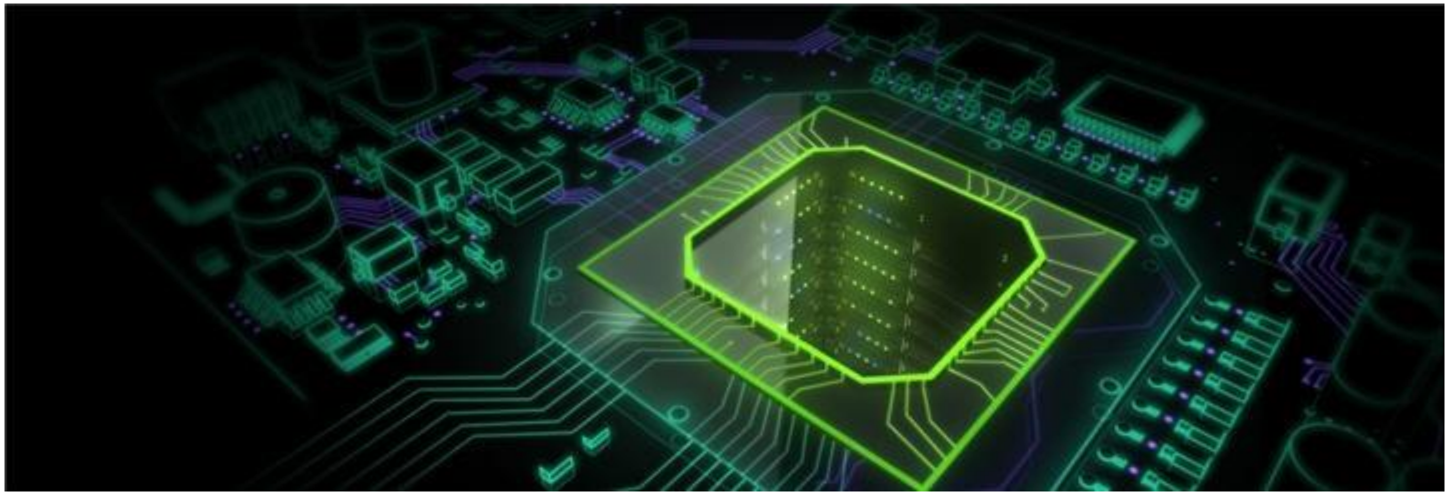
CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

CUDA
Advanced Techniques 4

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



This Lectures

- Texture memory
- A glimpse on libraries
- Power-aware programming
- Putting it all together

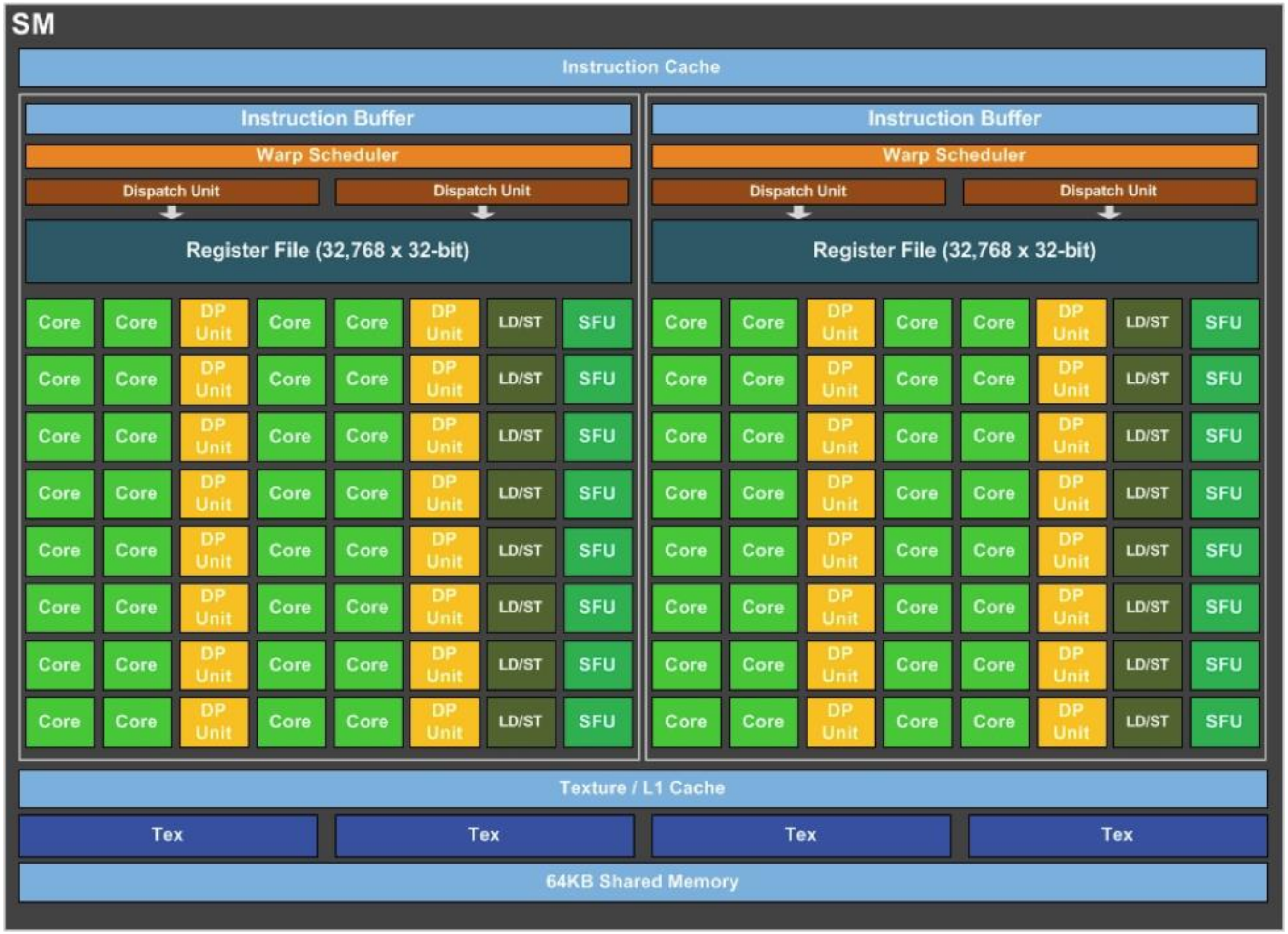
Texture Memory

Texture Memory

- read-only memory
- Can improve performance and reduce memory traffic when reads have certain access patterns.
- Originally designed for the classical OpenGL and DirectX rendering pipelines.
- But has some properties that make it extremely useful for computing, especially computer vision application.

Texture Memory

- Texture memory is cached on chip
  - In KB range in every SM
  - In some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM.
- Texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality.
- Every SM has several texture fetch units

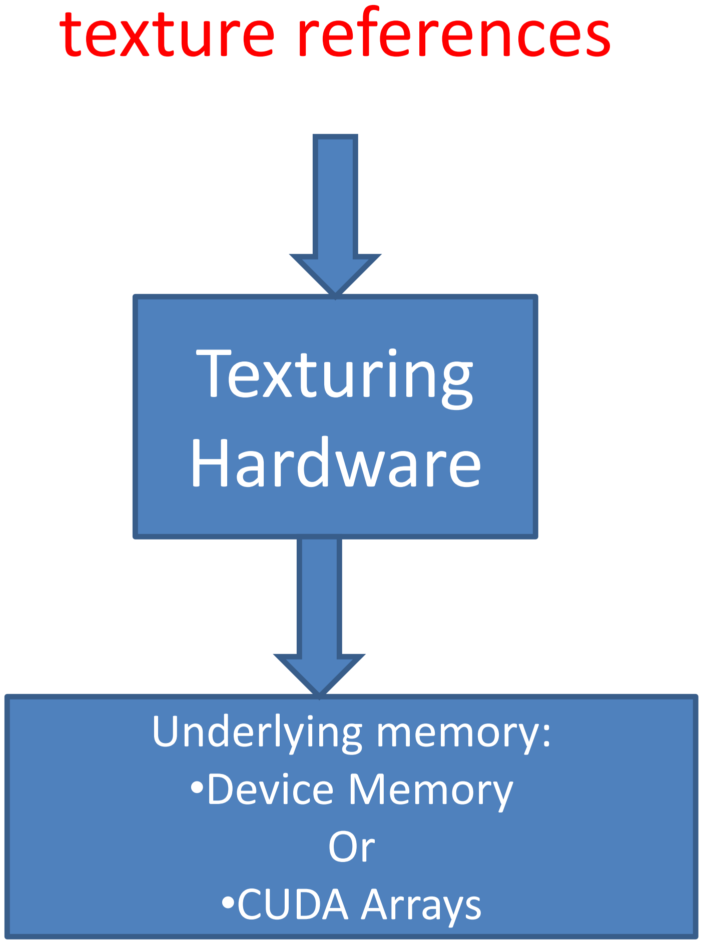


One SM from Pascal architecture (GP100)

Source: https://www.extremetech.com/extreme/226032-nvidias-pascal-gp100-gpu-massive-bandwidth-enormous-double-precision-performance

Texture Memory

- The texture cache is optimized for 2D spatial locality.
- Part of DRAM
- The process of reading a texture is called a *texture fetch*.
- Can be addressed as 1D, 2D, or 3D dimensional arrays.
- Elements of the array are called *texels*.


texture references
Texturing Hardware
Underlying memory:
•Device Memory
Or
•CUDA Arrays

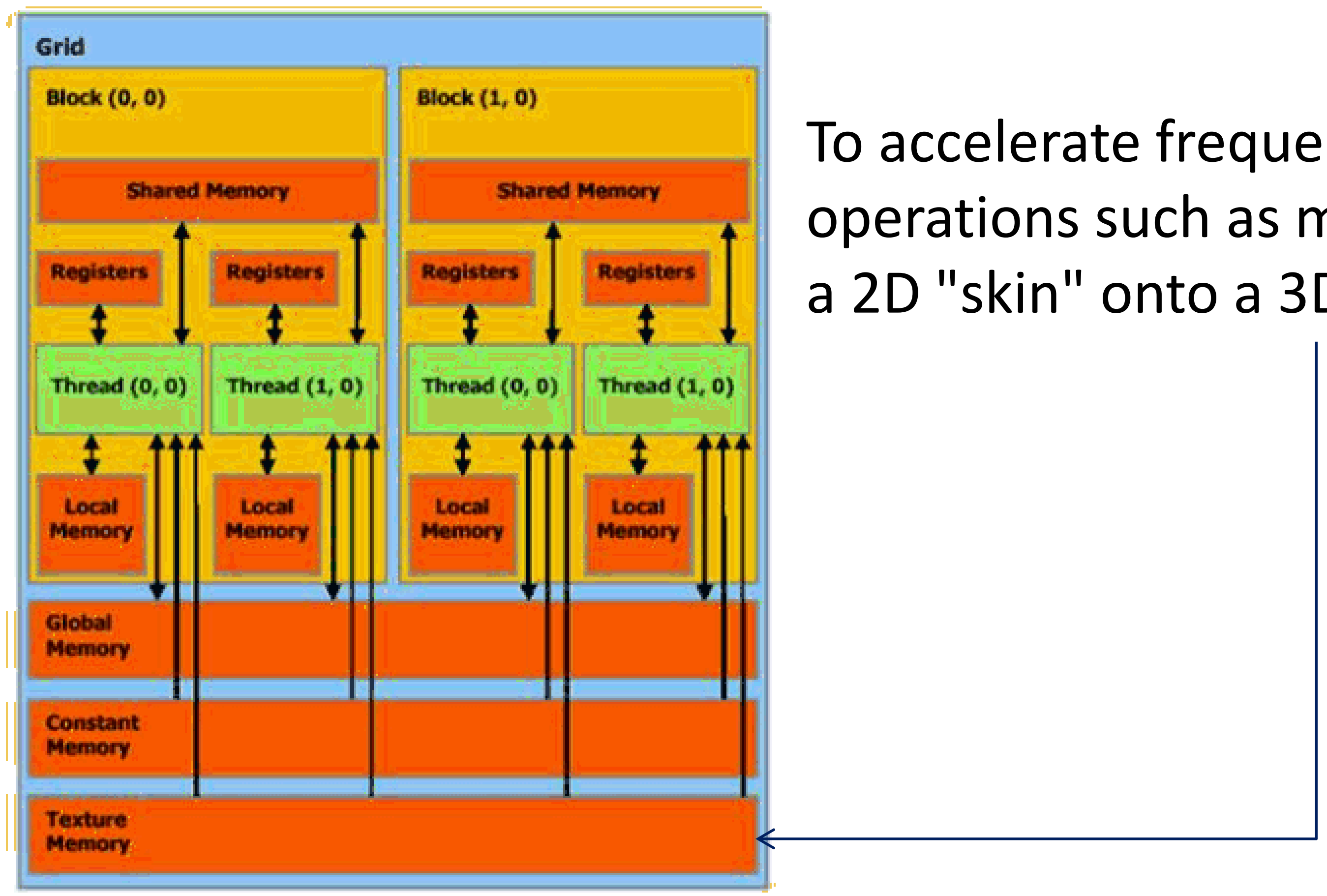
CUDA Arrays

- Designed specifically to support texturing.
- Allocated from device memory
- Do not consume any CUDA address space.
- Have an opaque layout
- Cannot be addressed by pointers
- Memory locations addressed using two things:
  - array handle
  - set of 1D, 2D, or 3D coordinates

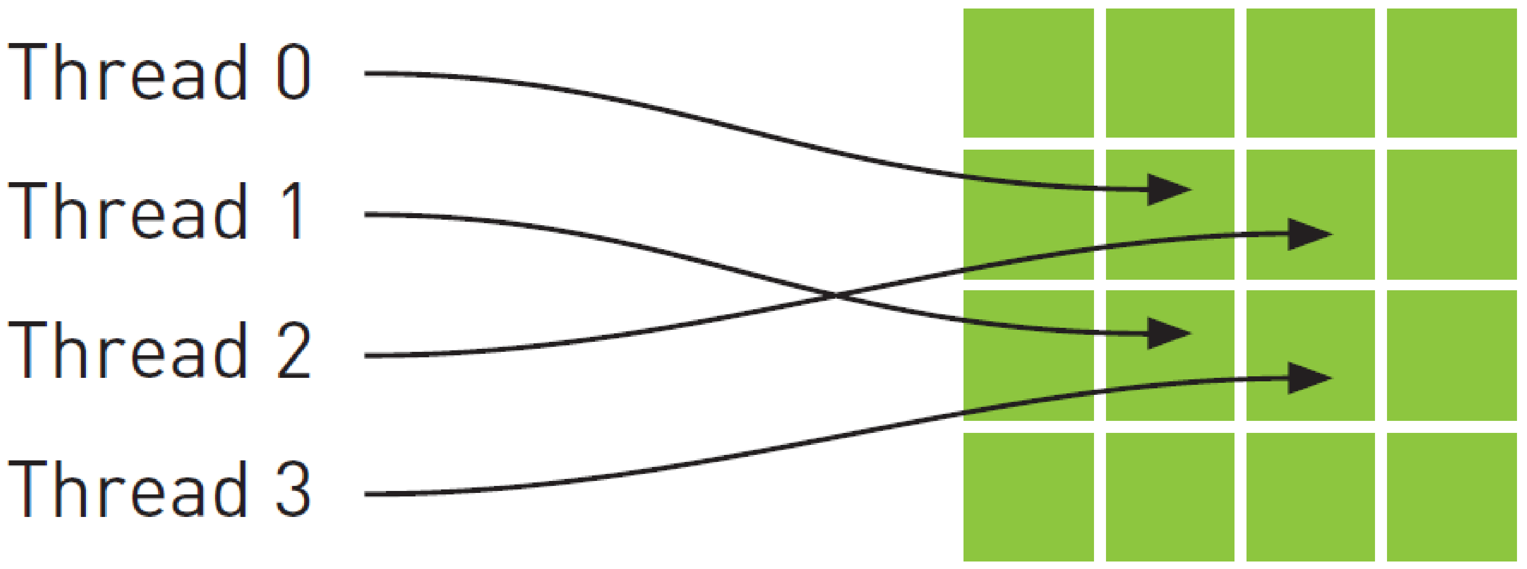
Why CUDA Arrays?

- Designed so that contiguous addresses exhibit 2D or 3D locality.

Texture Memory

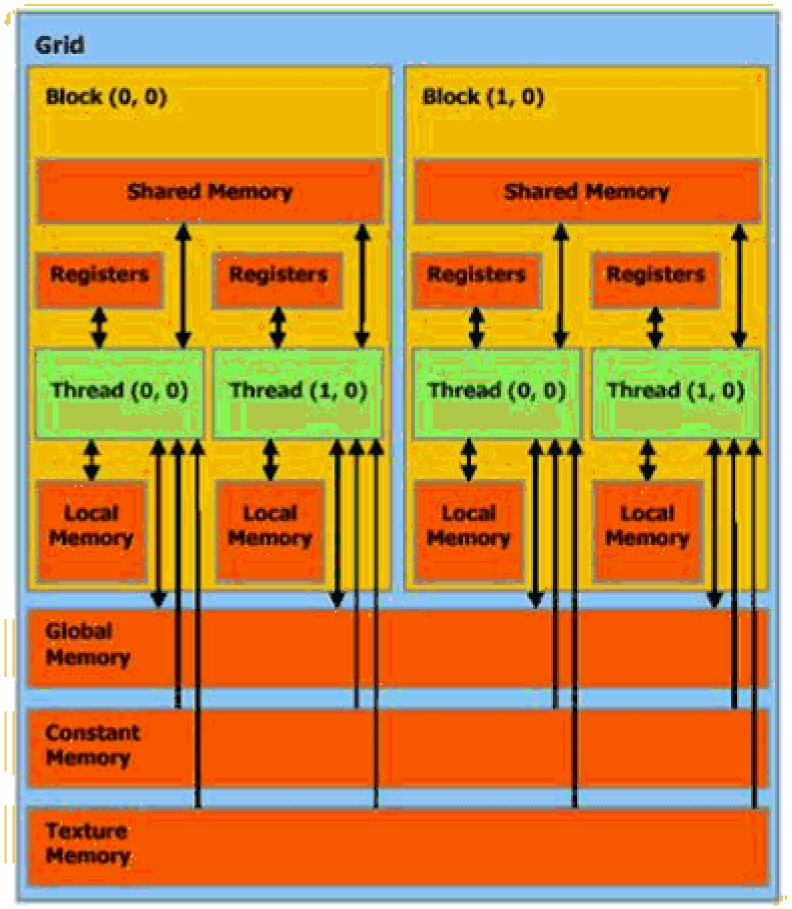


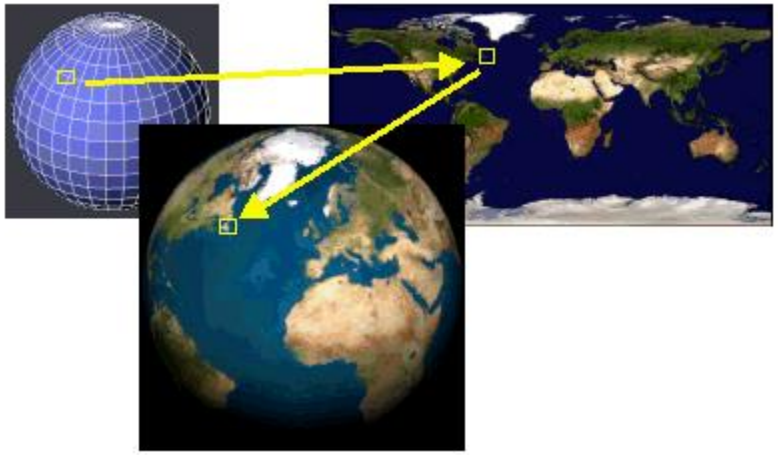
To accelerate frequently performed operations such as mapping
a 2D "skin" onto a 3D polygonal model



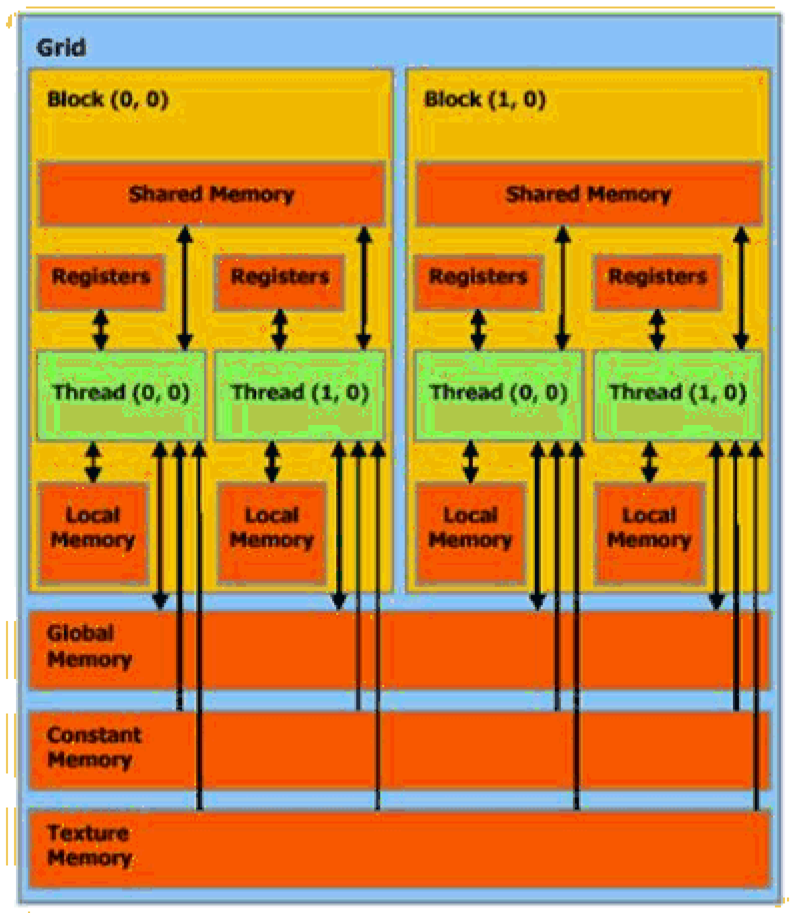
source: http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

Texture Memory





Texture Memory



Capabilities:

- Ability to cache global memory
- Dedicated interpolation hardware
- Provides a way to interact with the display capabilities of the GPU.

The best performance is achieved when the threads of a warp read locations that are close together from a spatial locality perspective.

Allocating CUDA Arrays

```
cudaError_t cudaMallocArray              (
            struct cudaArray **          array,
            const struct cudaChannelFormatDesc *        desc,
            size_t      width,
            size_t      height,
            unsigned int  flags
);
```

array: pointer to allocated array in device memory
width: array width in bytes
height: default is 0 → 1D array
flags:

- cudaArrayDefault: default array allocation
- cudaArraySurfaceLoadStore

Allocating CUDA Arrays

```
cudaError_t cudaMallocArray              (
            struct cudaArray **  array,
            const struct cudaChannelFormatDesc *  desc,
            size_t      width,
            size_t      height,
            unsigned int  flags
);
```

```
struct cudaChannelFormatDesc
cudaCreateChannelDesc(x,y,z,w,f);
```

```
struct cudaChannelFormatDesc {
    int x, y, z, w;   ← number of bits in each member of the texture element
    enum cudaChannelFormatKind f;
};
```

cudaChannelFormatKindSigned	Signed channel format
cudaChannelFormatKindUnsigned	Unsigned channel format
cudaChannelFormatKindFloat	Float channel format
cudaChannelFormatKindNone	No channel format

Texture Fetch

- First parameter is texture reference
  - defines which part of texture memory is fetched
  - must be bound through runtime functions to texture memory
  - Attribute:
    - texture is addressed as 1D, 2D, or 3D
    - the input and output data types of the texture fetch
    - the input coordinates are interpreted
    - what processing should be done
  - Type of texels are the basic: integer, single/double precision floating point, … .

Steps for Using Texture Memory in Your CUDA Code

1. Declare the texture memory in CUDA.
2. Bind the texture memory to your texture reference in CUDA.
3. Read the texture memory from your texture reference in CUDA Kernel.
4. Unbind the texture memory from your texture reference in CUDA.

Step 1: Declare

texture <type, dim, readmode> texture_reference;

- texture_reference: the handle to be used
- type: type of texel data returned from an access to the texture: int, float, … .
- dim: 1 (default), 2, or 3
- readmode: controls conversion of texel returned by an access
  - cudaReadModeElementType (default) no conversion
  - cudeReadModeNormalizedFloat
    - if type is integer, value returned is mapped to [-1.0,1.0] for signed, and [0.0, 1.0] for unsigned
- Example:

texture <float, 2, cudaReadModeElementType> mytex;

Step 2: Bind

cudaBindtexture (size *t offset,
& testure_reference , const void * devptr,
size_t size) ;

- Binds size bytes of the memory area pointed to by devPtr to texture reference texture_reference.
- offset parameter is an optional byte offset.
- devPtr: Memory area on device
- size: Size of the memory area pointed to by devPtr

Step 3: Read

- The easiest is: tex1Dfetch()

Example:

```
texture <int,1,cudaReadModeElementType> texref;
__global__
void textureTest(int *out){
    int  tid =  blockIdx.x * blockDim.x + threadIdx.x;
    float x;
    int i;
    for(i=0; i<30; i++)
        x = tex1Dfetch(texref, i);
}
```

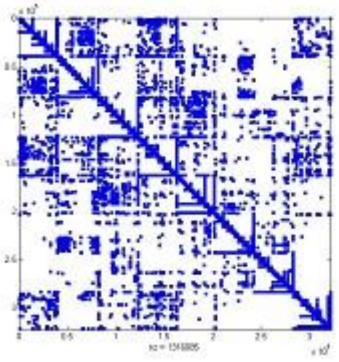
Step 4: Unbind

```
cudaUnbindTexture(texture_reference);
```

So

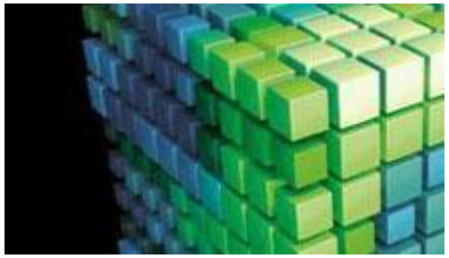
- Texture memory size is very small.
- We just scratched the surface of texture memory.
- Two usages of texture memory outside graphics applications:
  - Using texture cache to reduce bandwidth and work around coalescing constraints.
  - Make use of advanced fixed-function hardware put inside GPU for graphics applications

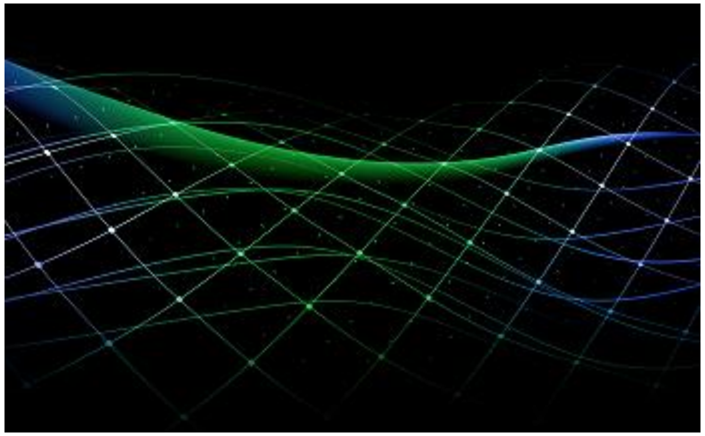
cuDNN


cuSPARSE

A Glimpse on Libraries

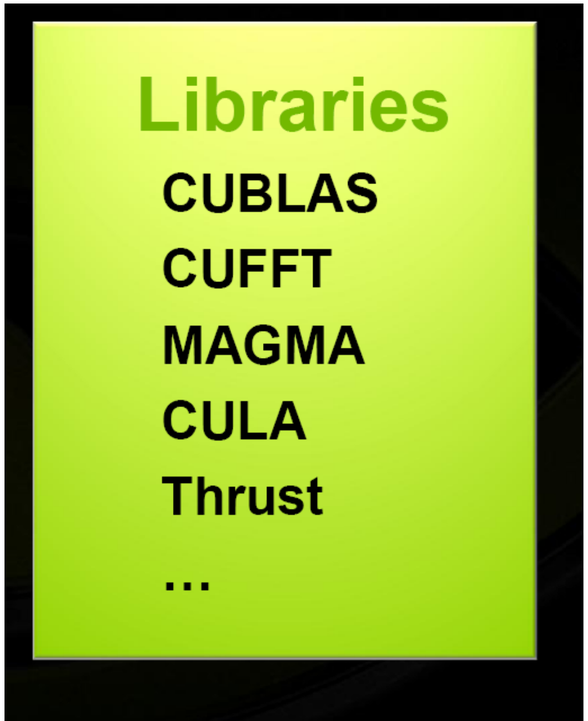
https://developer.nvidia.com/gpu-accelerated-libraries


cuBLAS


CUDA Math Library

CUDA Libraries

- CUDA provides a set of very useful libraries.
- This increases the programmer productivity.



Libraries

CUBLAS

CUFFT

MAGMA

CULA

Thrust

…

Example: CUBLAS

- Cuda Based Linear Algebra Subroutines
  - conjugate gradient, linear solvers, …
- Single GPU or Multiple GPUs
- Support CUDA Stream
- Basic preparation
  - Install CUDA Toolkit
  - Include cublas_v2.h
  - Link cublas.lib (-lcublas)

CUDA Libraries: CUBLAS

- Some basic tips
  - Every CUBLAS function needs a handle
  - The CUBLAS function must be written between cublasCreate() and cublasDestory()
  - Every CUBLAS function returns a cublasStatus_t to report the state of execution.
  - Column-major storage

Example: CUFFT

- Cuda Based Fast Fourier Transform Library.
- The FFT is a divide-and-conquer algorithm
- Computes FFT on Nvidia CUDA
- 1D, 2D, and 3D
- The CUFFT library is freely available as part of the CUDA Toolkit
- #include<cufft.h>

Power Aware Programming

What Can A Software Application Do?

- Use less expensive operations
- Less stress on power-hungry parts
- Access and make use of internal GPU performance counters
  - PAPI: https://developer.nvidia.com/papi-cuda-component
  - Nvidia Management Library (NVL)
- Interaction of three players:
  - The application software
  - The Compiler
  - The OS

Power-Aware Applications

- Applications must be Designed and tested for power management
- Applications must handle sleep transitions seamlessly
- You can differentiate your application with power management features
  - Handle power management events
  - Scale behavior based on user's power preference

NVIDIA NML

- Nvidia Management Library

https://developer.nvidia.com/nvidia-management-library-nvml

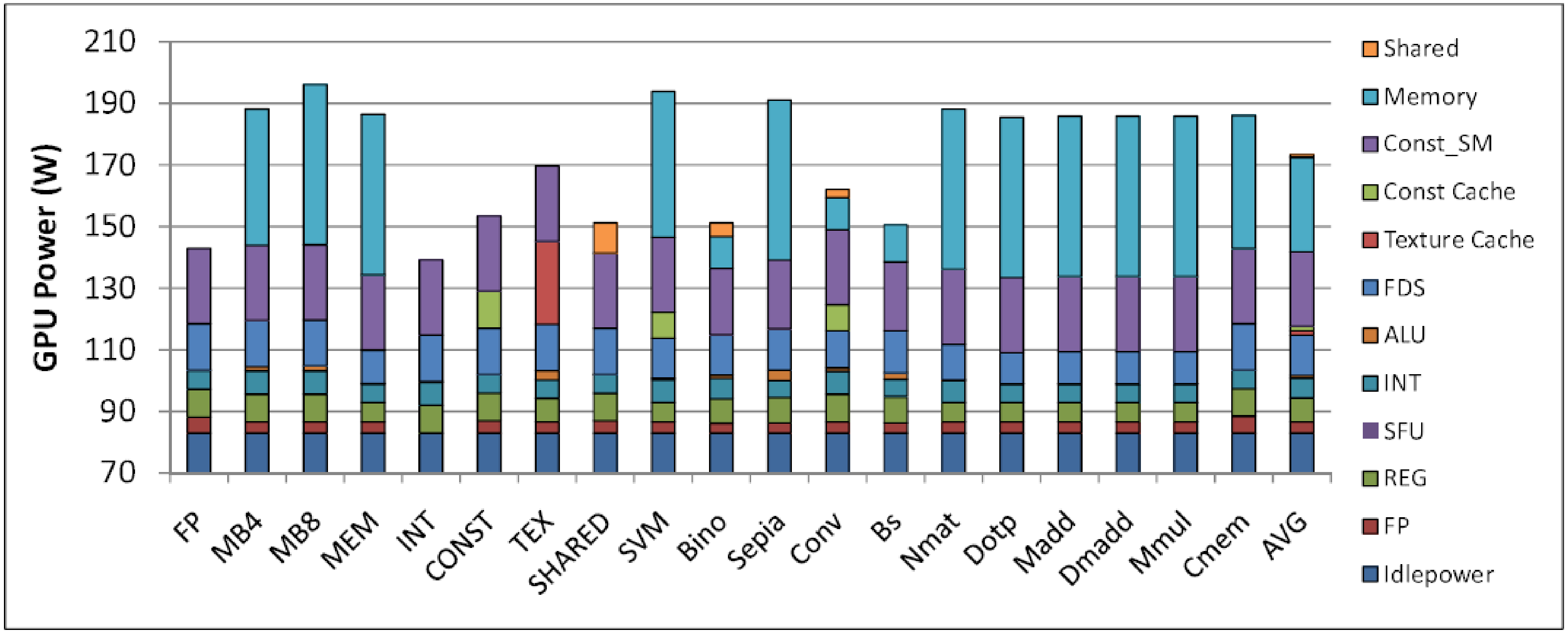
- C based interface for monitoring and managing various states within NVIDIA GPUs
- You call it from the host
- Compile with -lnvidia-ml
- You must have installed Nvidia CUDA toolkit and Nvidia development drivers

NVIDIA NML

```
#include <stdio.h>
#include <nvml.h>
int main()
{
nvmlReturn_t  result;
unsigned  int  device_count , i;
char version[80];
result =  nvmlInit ();
result =  nvmlSystemGetDriverVersion (version,80);
result = nvmlDeviceGetCount(&device_count );
}
```

What Can NVML Do?

- Get you the temperature of the device in Celsius.
  - nvmlDeviceGetTemperature()
  - nvmlDeviceGetPowerUsage()
- Reduce the clock frequency (throttling)
  - nvmlDeviceGetApplicationsClock()
  - nvmlDeviceGetAPIRestriction()
  - nvmlDeviceSetApplicationsClock()



Power Breakdown: GeForce 285 GTX

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!

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 at the same time.

The Process of Parallel Programming

- Problem decomposition
- Algorithm selection
- Implementation in a language
- Performance tuning

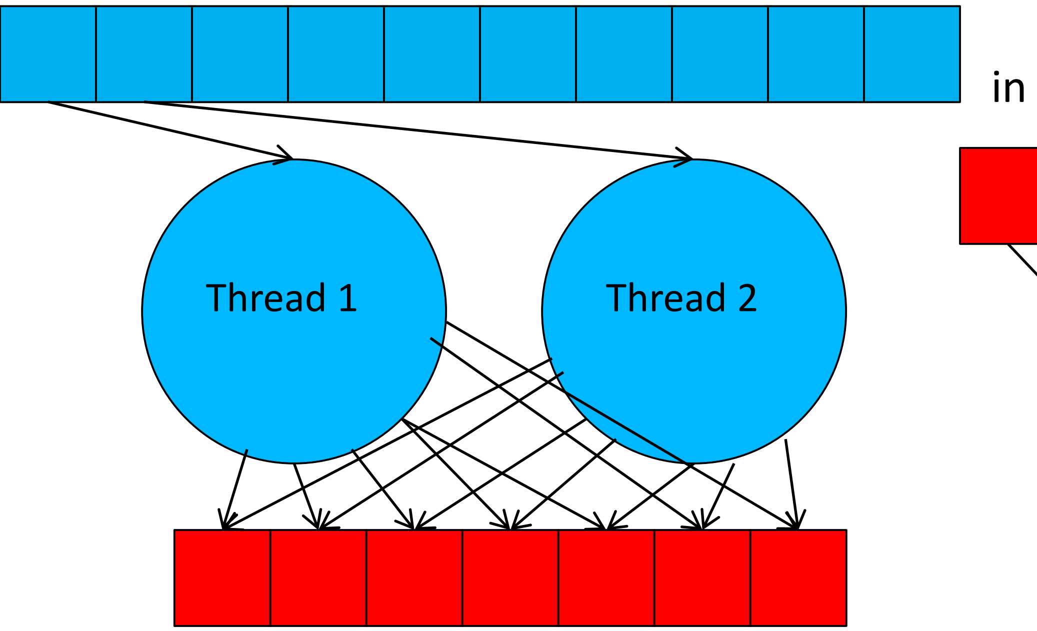
This is what we have been doing till now!

Problem Decomposition

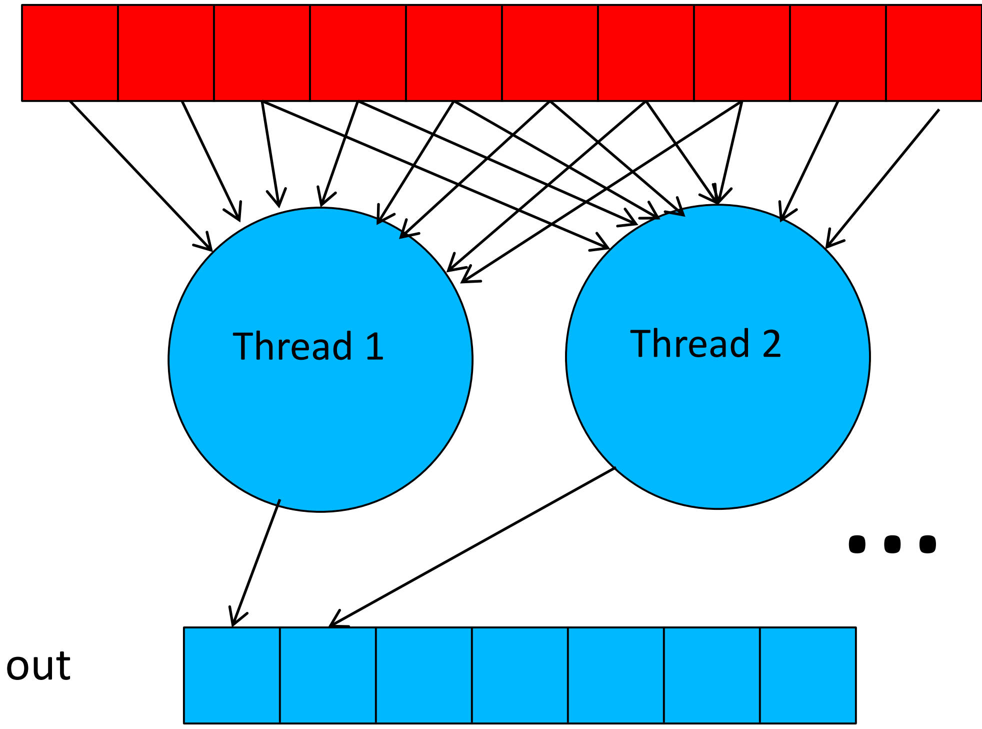
Identify the work to be performed by each unit of parallel execution → thread in CUDA

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
- Algorithm 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!

Putting It All Together

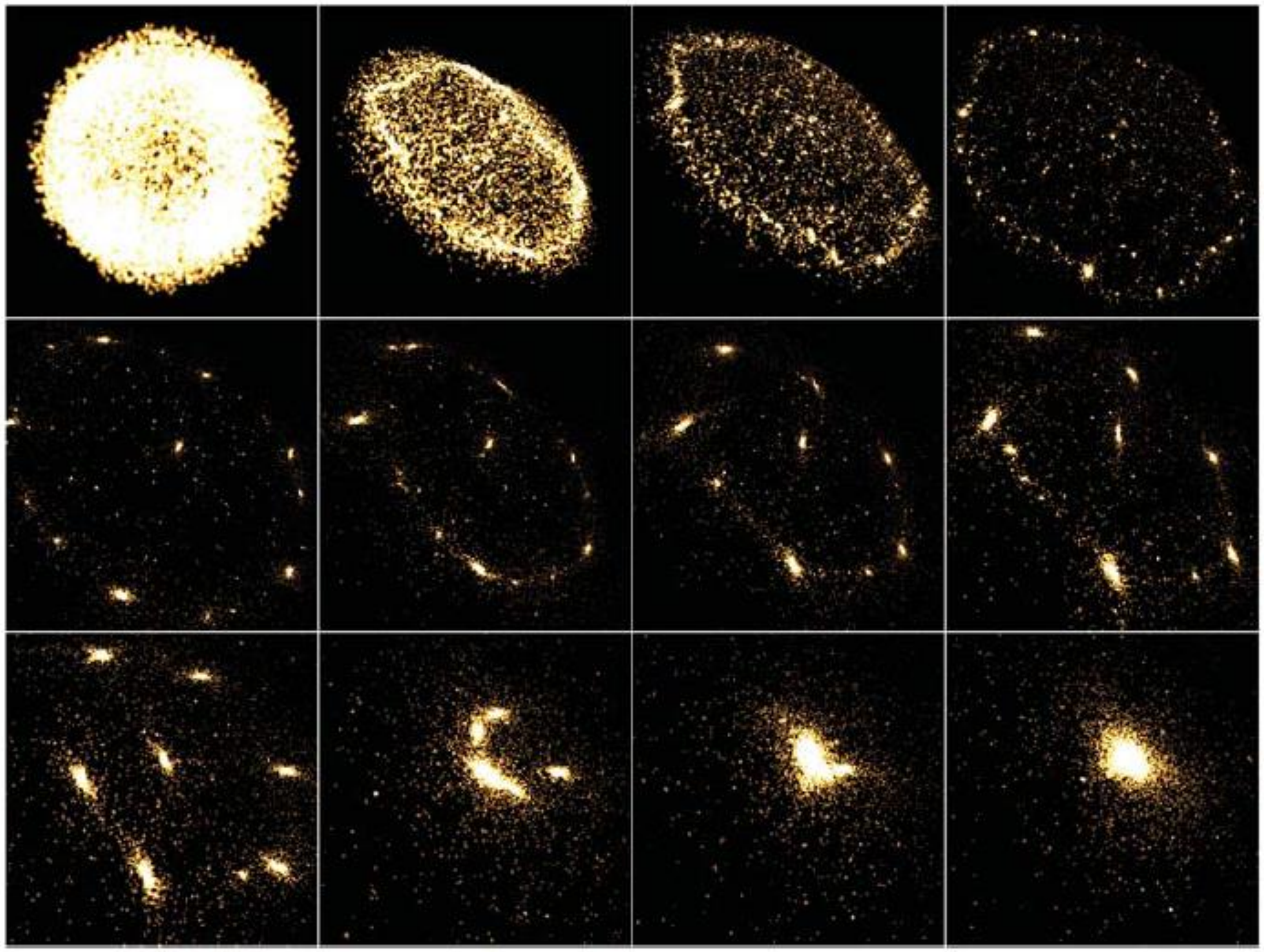
What will we do?

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)

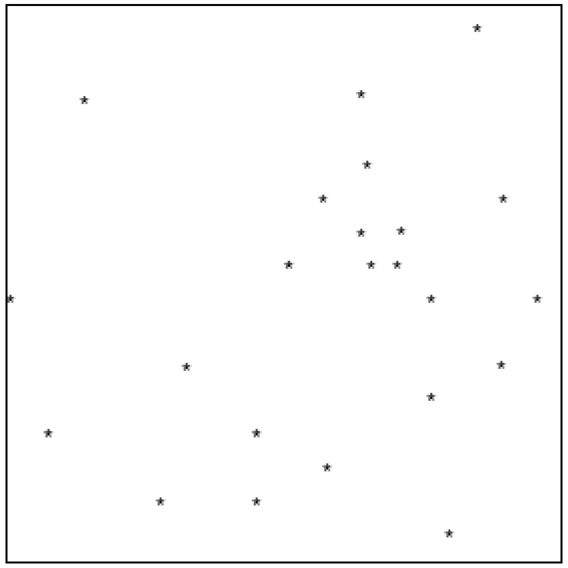
Barnes Hut n-Body Algorithm

```
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

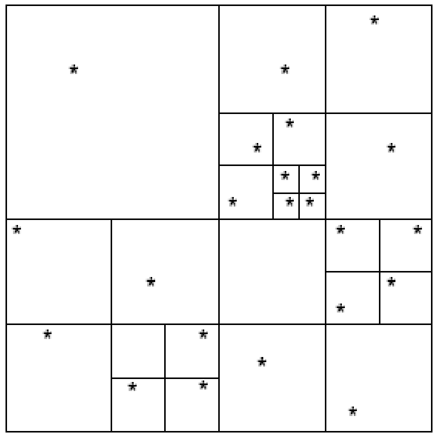
```
0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance
    5. Compute forces acting on each body with help of octree
    6. Update body positions and velocities
}
7. Transfer result to CPU and output
```

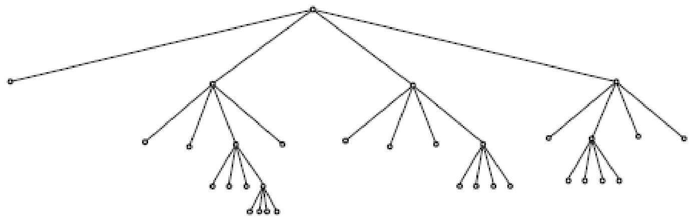


Barnes Hut n-Body Algorithm

```
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 velocit
}
7. Transfer result to CPU and output
```

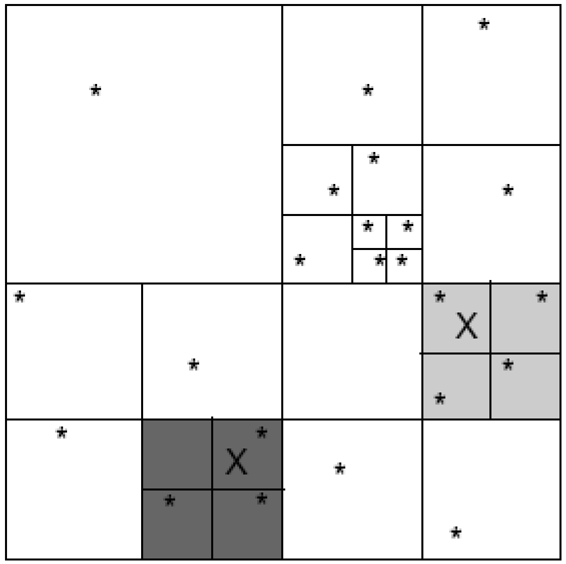
cells: Internal tree nodes
Bodies: leaves





Barnes Hut n-Body Algorithm

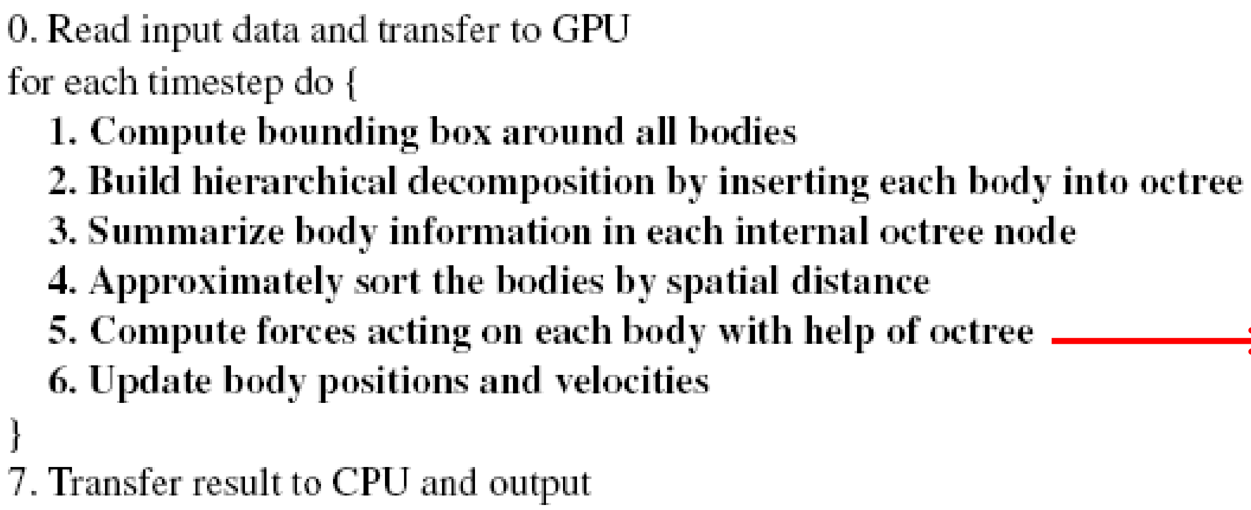
```
0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance
    5. Compute forces acting on each body with help of octree
    6. Update body positions and velocities
}
7. Transfer result to CPU and output
```



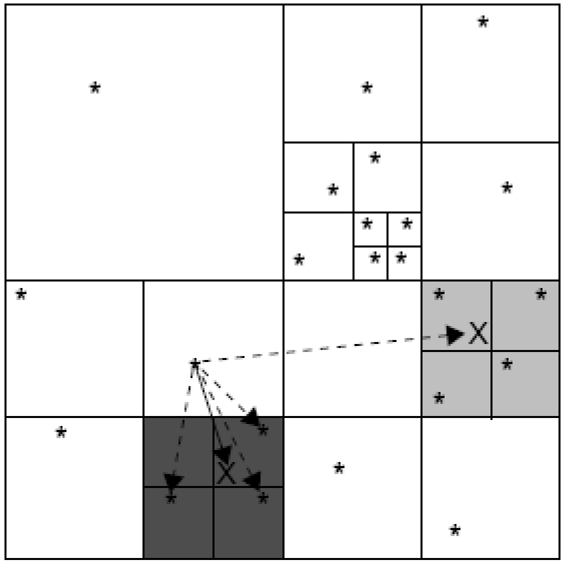
Calculate for each cell:

- 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.

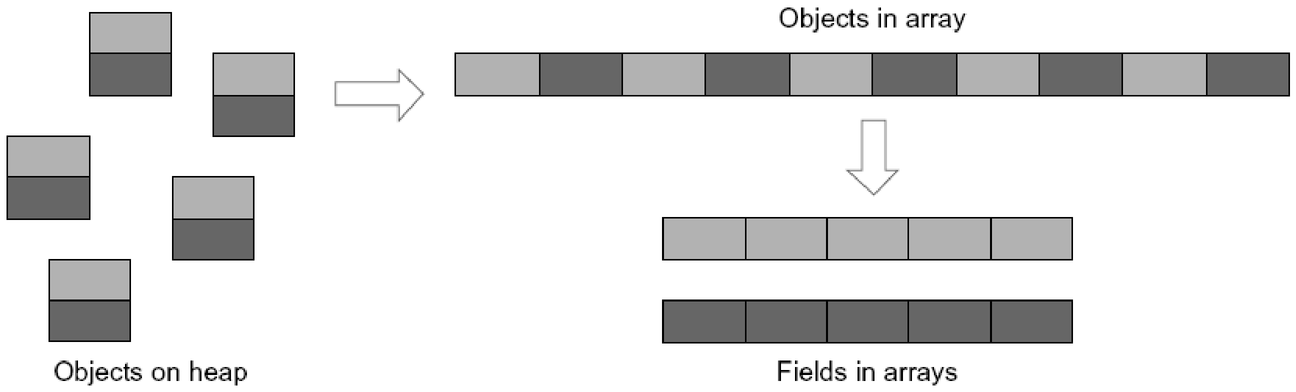
First Step: Data Structure

- 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

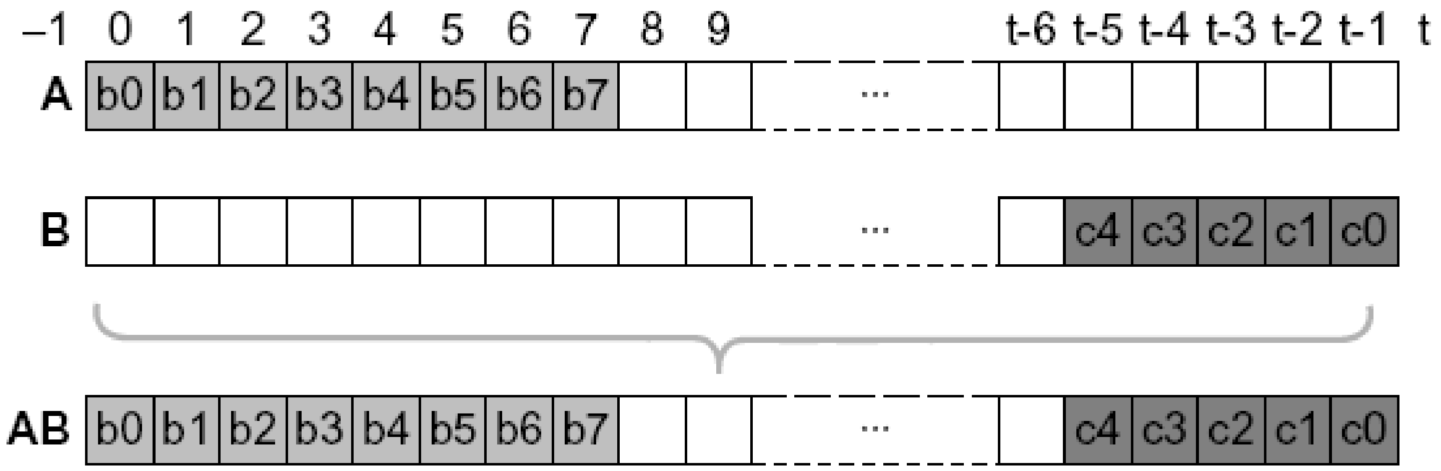
- Use an array-based data structure
- To be able to coalesce:
  - use several aligned scalar arrays, one per field
- Array indices instead of pointers makes a faster code



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

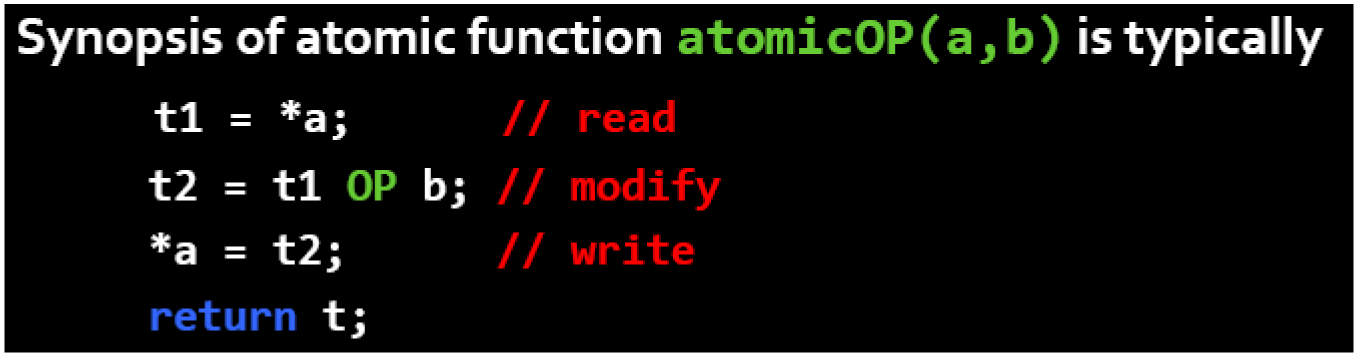
- computes a bounding box around all bodies
  - The root of the octree
  - has to find the minimum and maximum coordinates in the three spatial dimensions
- Implementation:
  - break up the data into equal sized chunks and assigns one chunk to each block
  - Each block then performs a reduction operation
  - 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

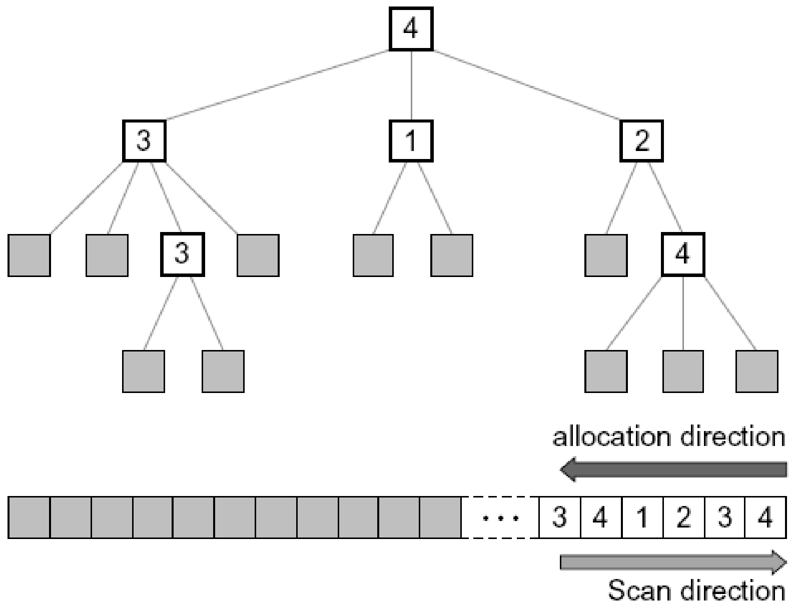
- If a body is already stored at this location, the thread:
  - creates a new cell by atomically requesting the next unused array index
  - inserts the original and the new body into this new cell
  - executes a memory fence (*__threadfence*) *to ensure the new subtree is visible to the rest of the cores*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

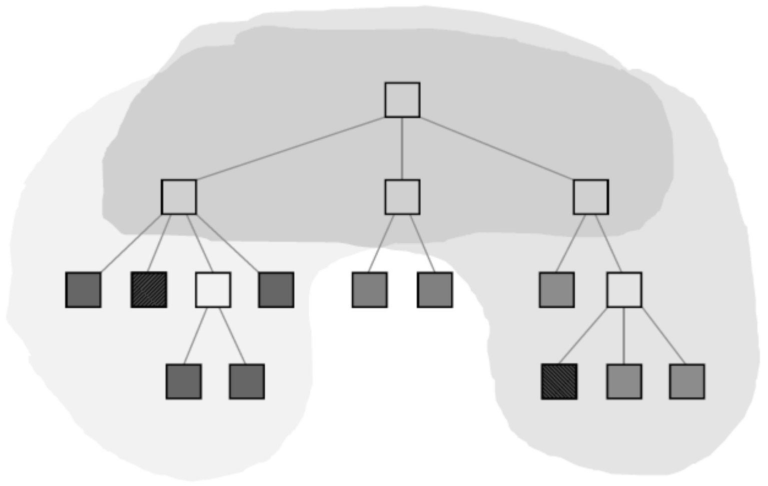


Kernel 3

- 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

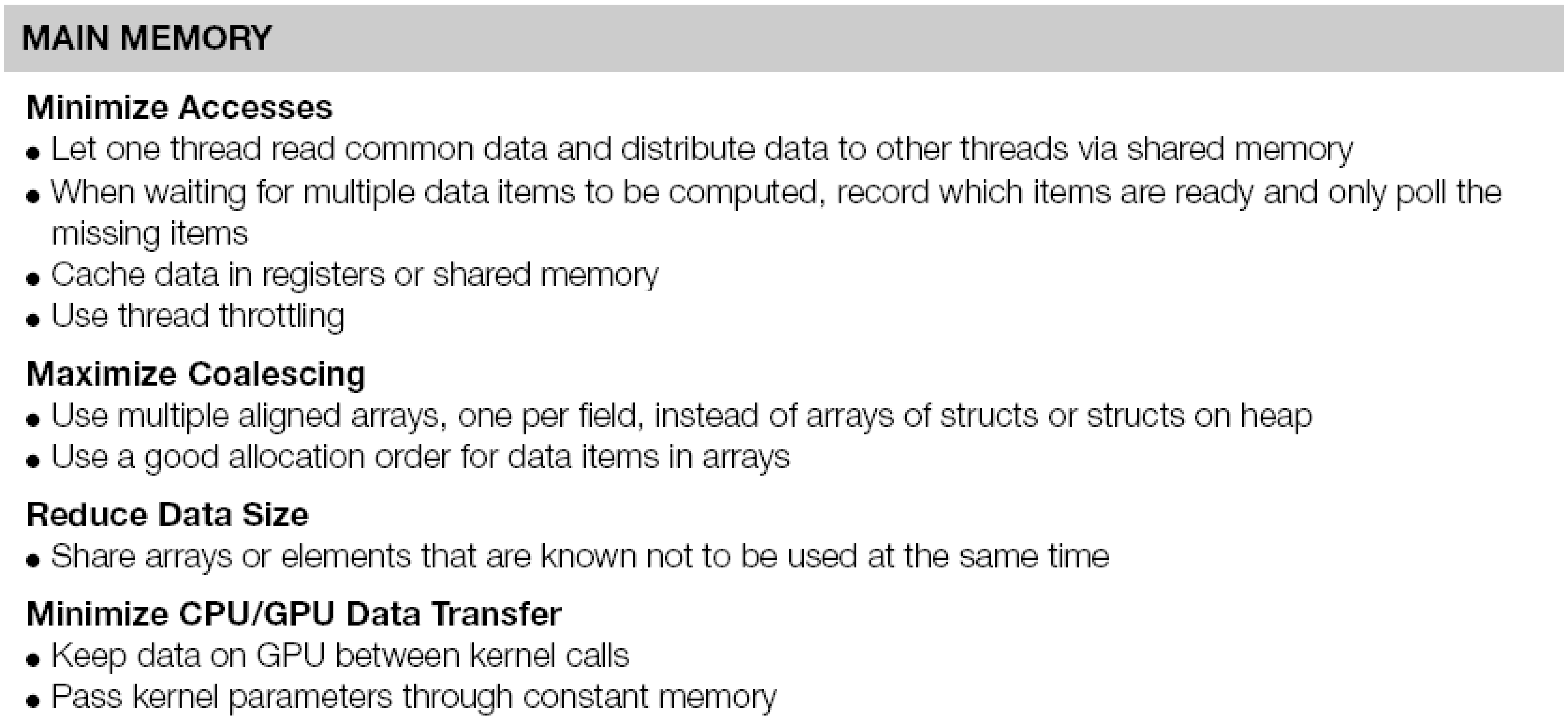
- Requires the vast majority of the runtime
- For each body, the corresponding thread traverses some prefix of the octree to compute the force acting upon this body.



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

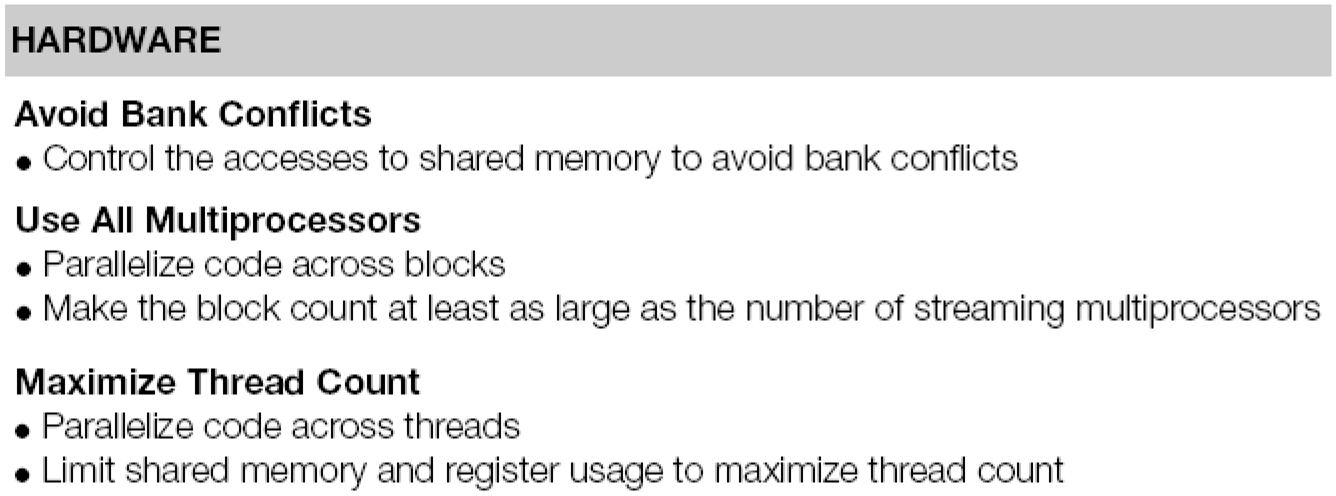
LOCKING

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



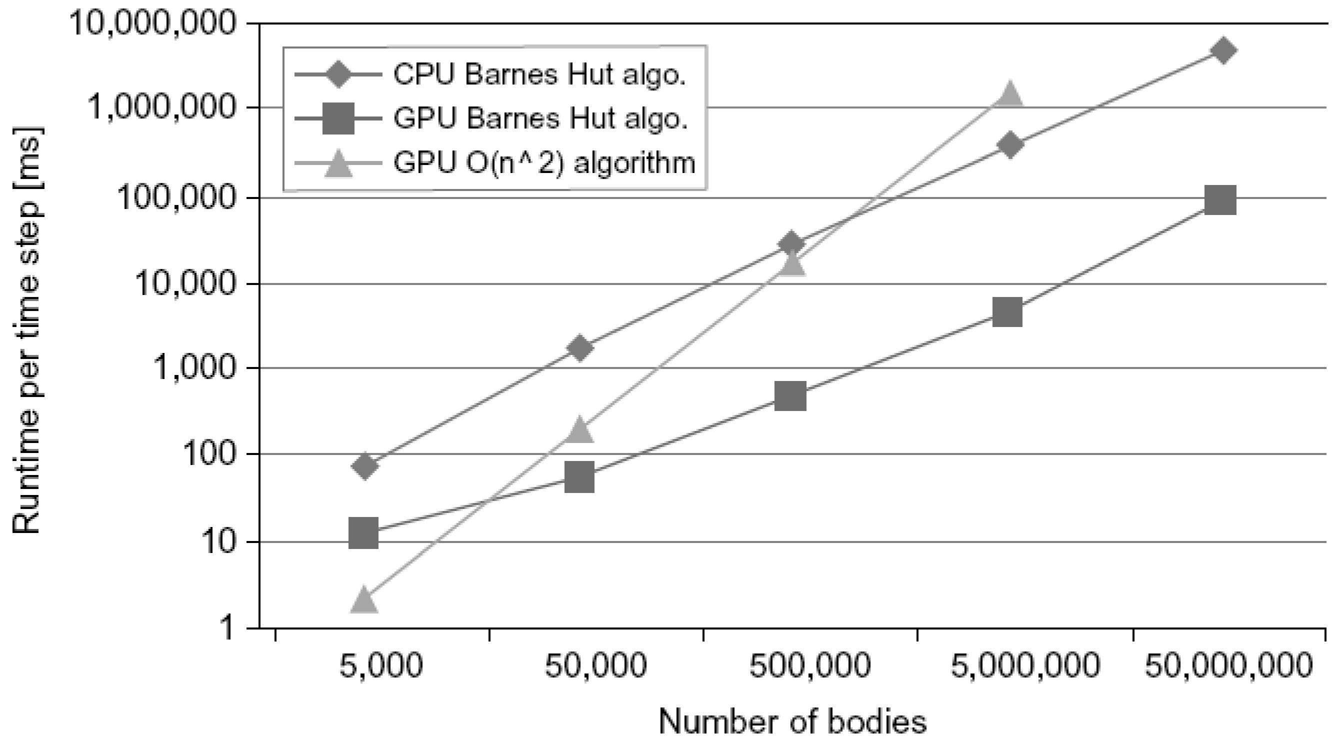
HARDWARE

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

- We did not cover all APIs of CUDA, but the main ones.
- Keep an eye on NVIDIA documentation, as more API are introduced and some are deprecated.
- To get the best performance:
    1. Pick the best (i.e. GPU friendly) algorithms
    2. Write a working problem
    3. Increase utilization
    4. Profile
    5. Optimize
    6. Goto step 4