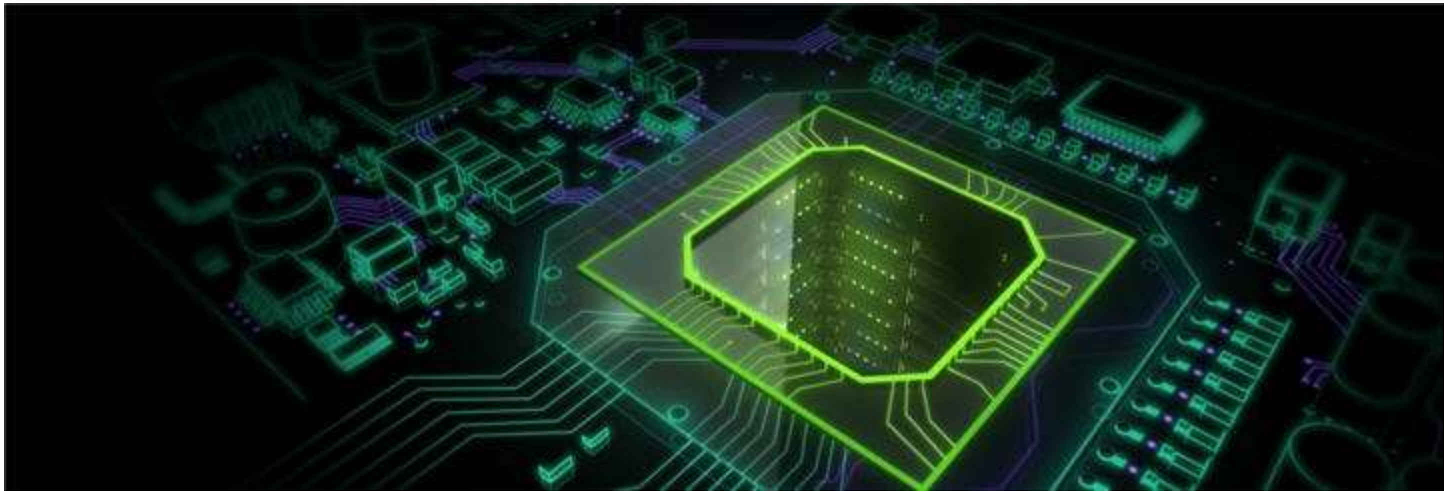
CSCI-GA.3033-012

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 13: Putting It All Together

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

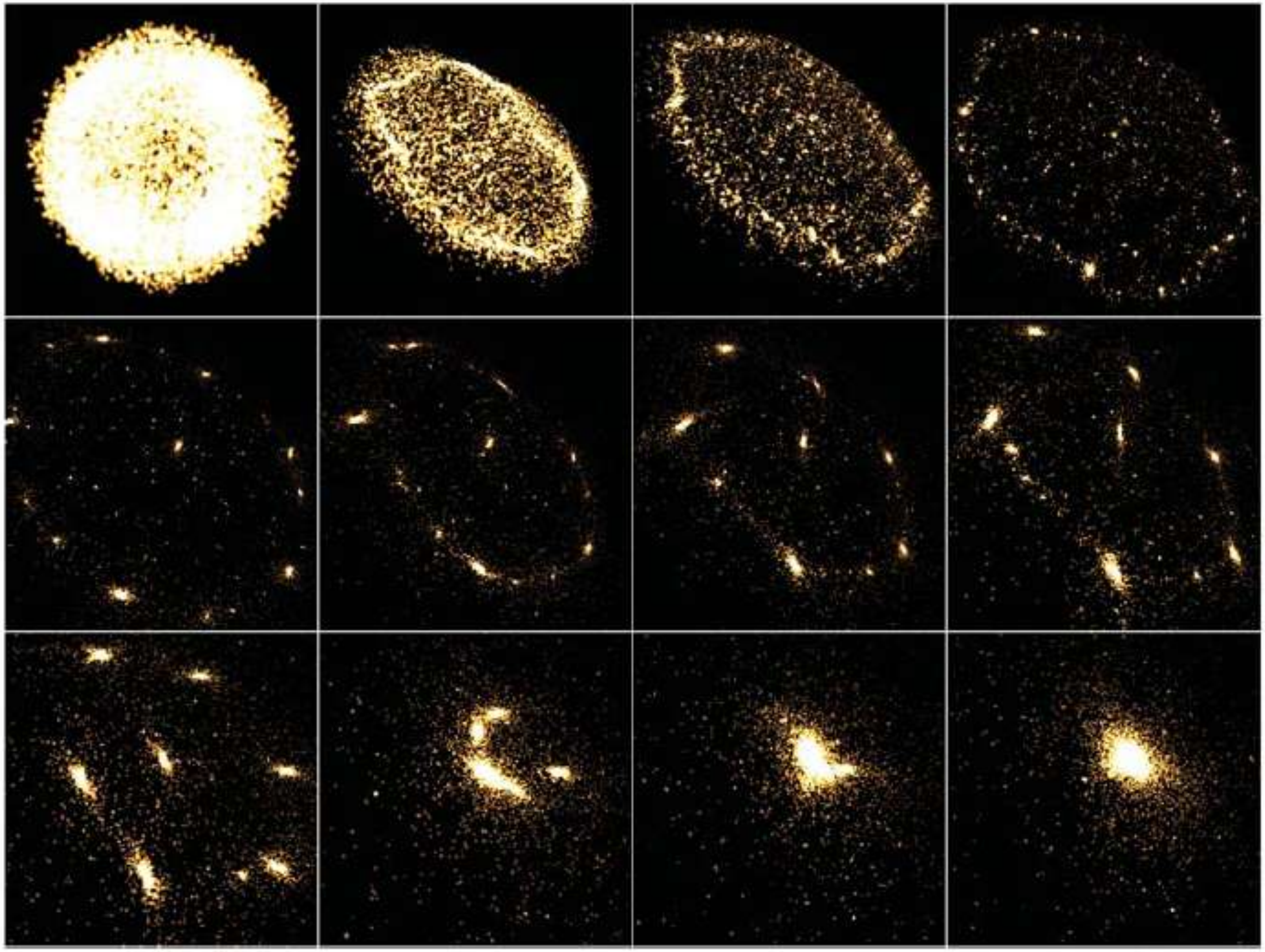


This Lecture

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

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

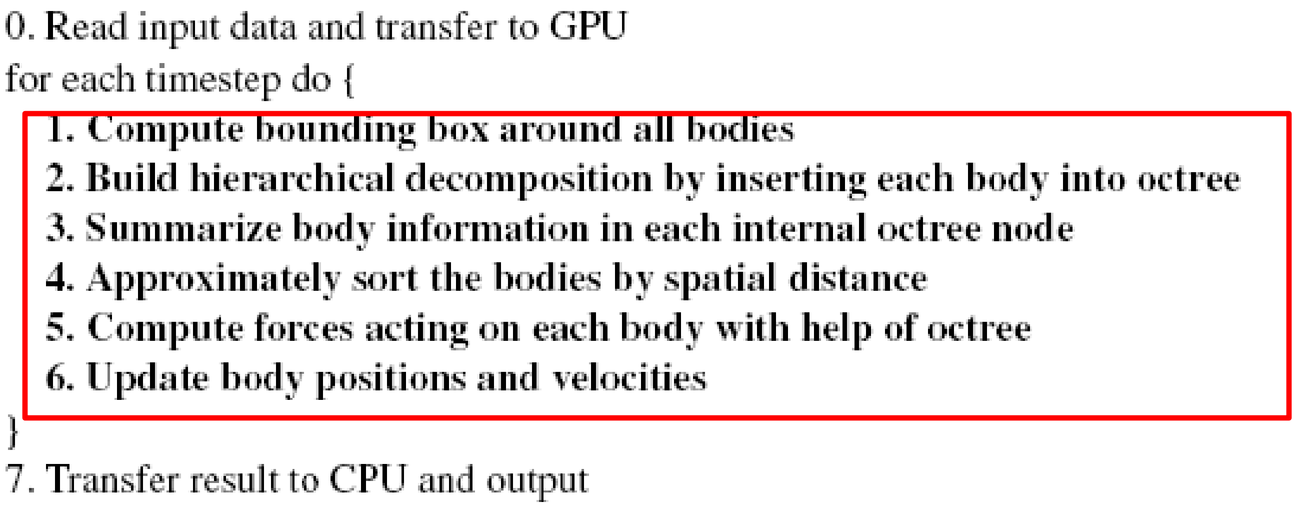
- Manifests itself in many domains: physics, astronomy, electromagnetics, molecules, etc.
- N points
- The answer at each point depends on data at all the other points
- $O(n^2)$
- To reduce complexity: compress data of groups of nearby points
  - A well-known algorithm to do this: Barnes Hut

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

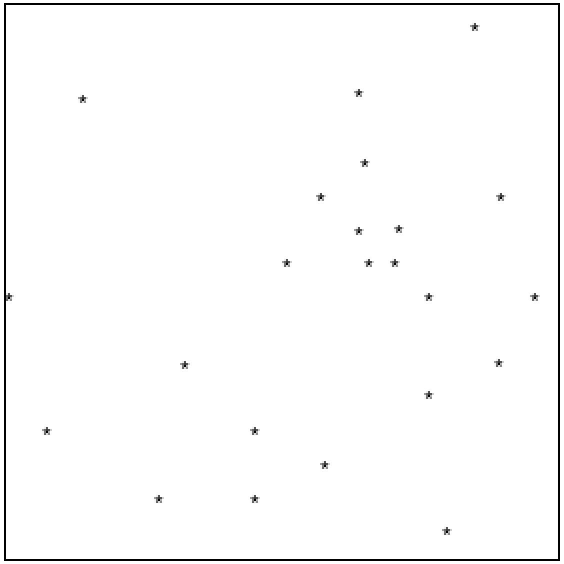


```
0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance
    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

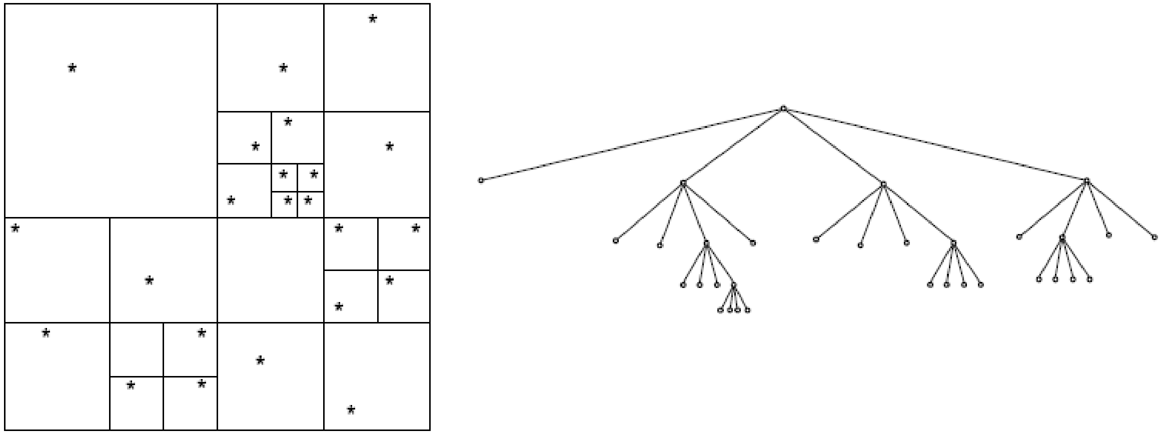
```
0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance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



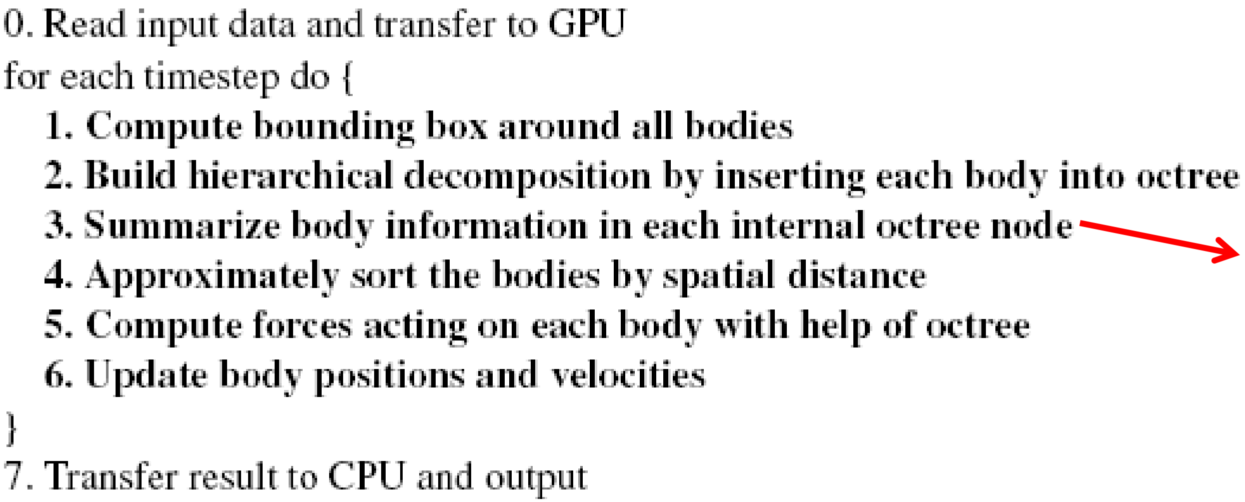
Barnes Hut n-Body Algorithm

```
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
```

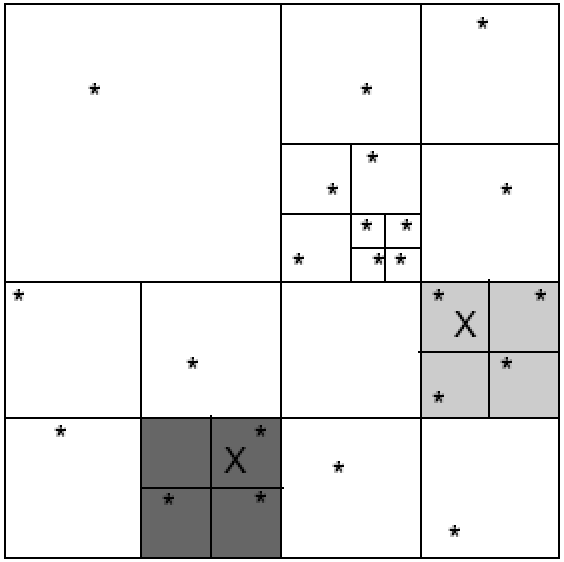
cells: Internal tree nodes
Bodies: leaves



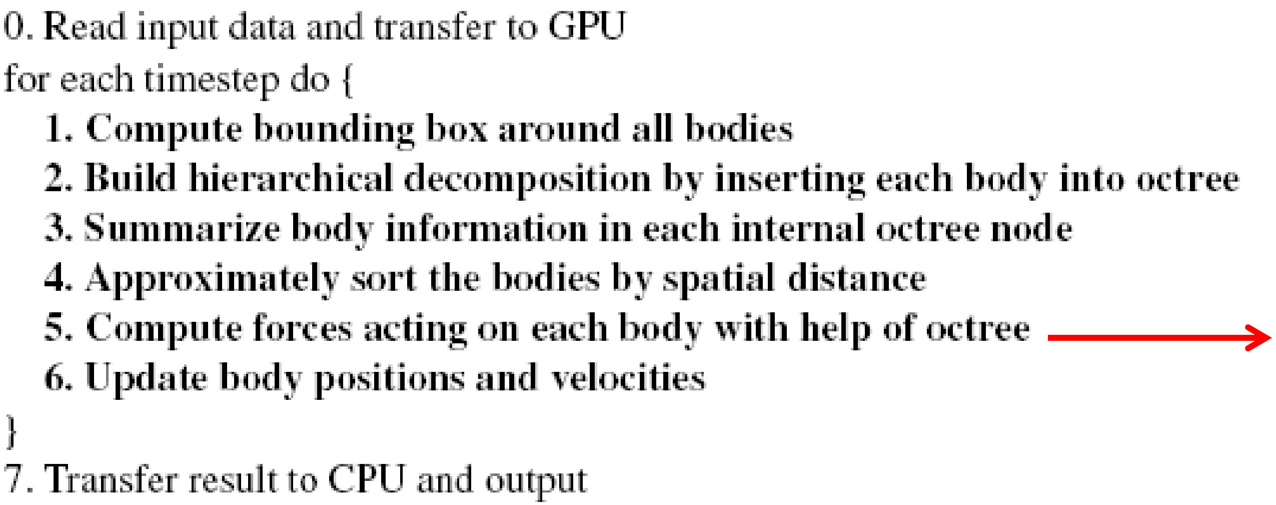
Barnes Hut n-Body Algorithm



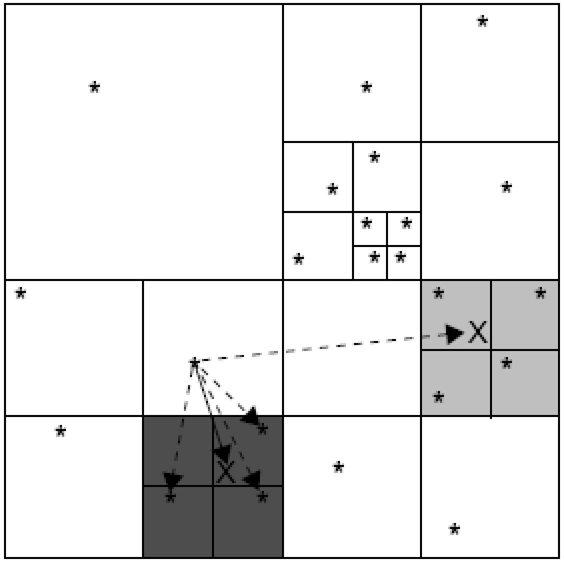
```
0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance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



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

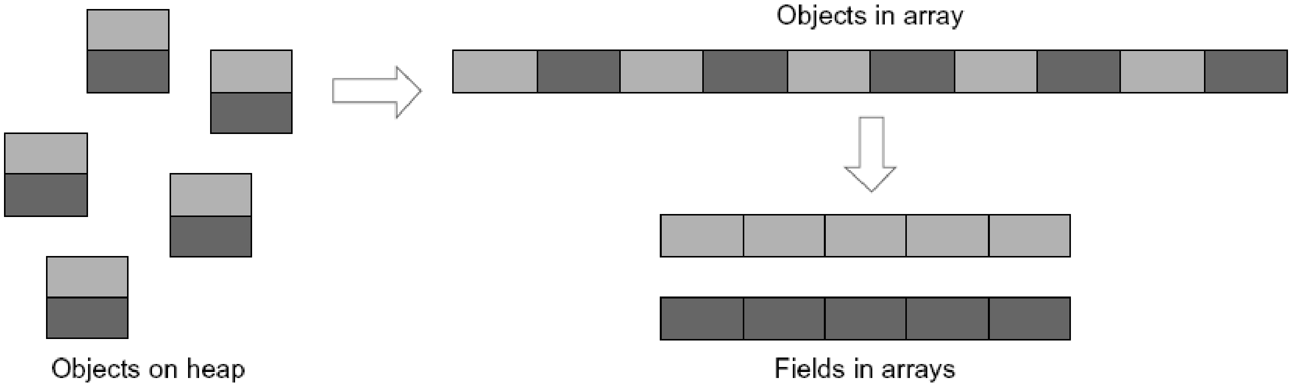
First Step: Data Structure

- 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

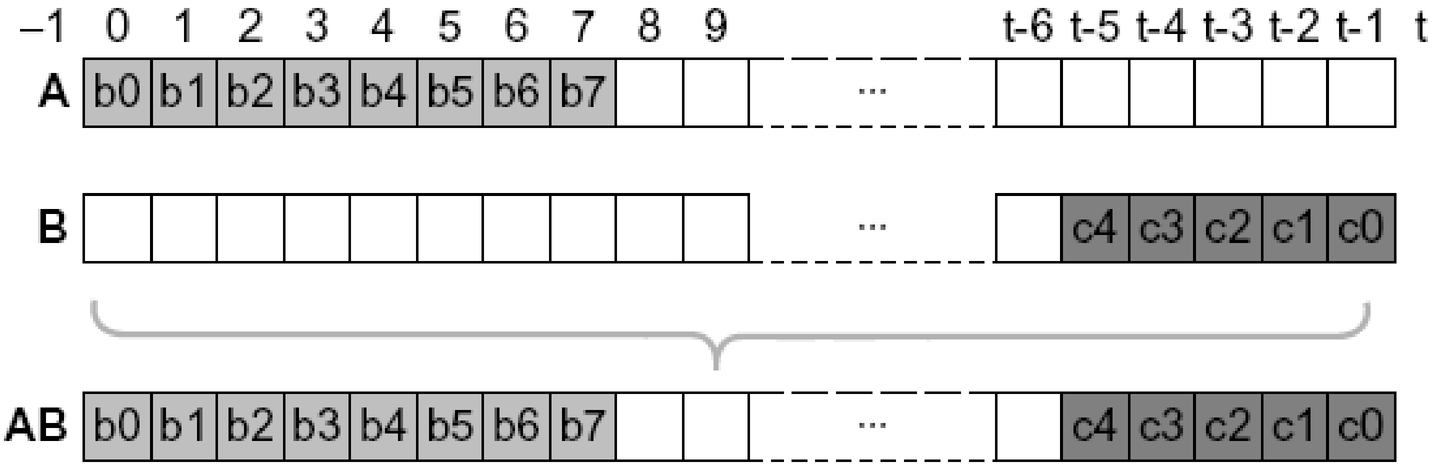
- Use an array-based data structure
- To be able to coalesce:
  - use several aligned scalar arrays, one per field
- Array indices instead of pointers makes a faster code



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.
  - We can alias arrays that hold only cell information with arrays that hold only body information to reduce the memory consumption

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 with a trip count of eight.
  - 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
  - reduction is performed in shared memory in a way that avoids bank conflicts and minimizes thread divergence

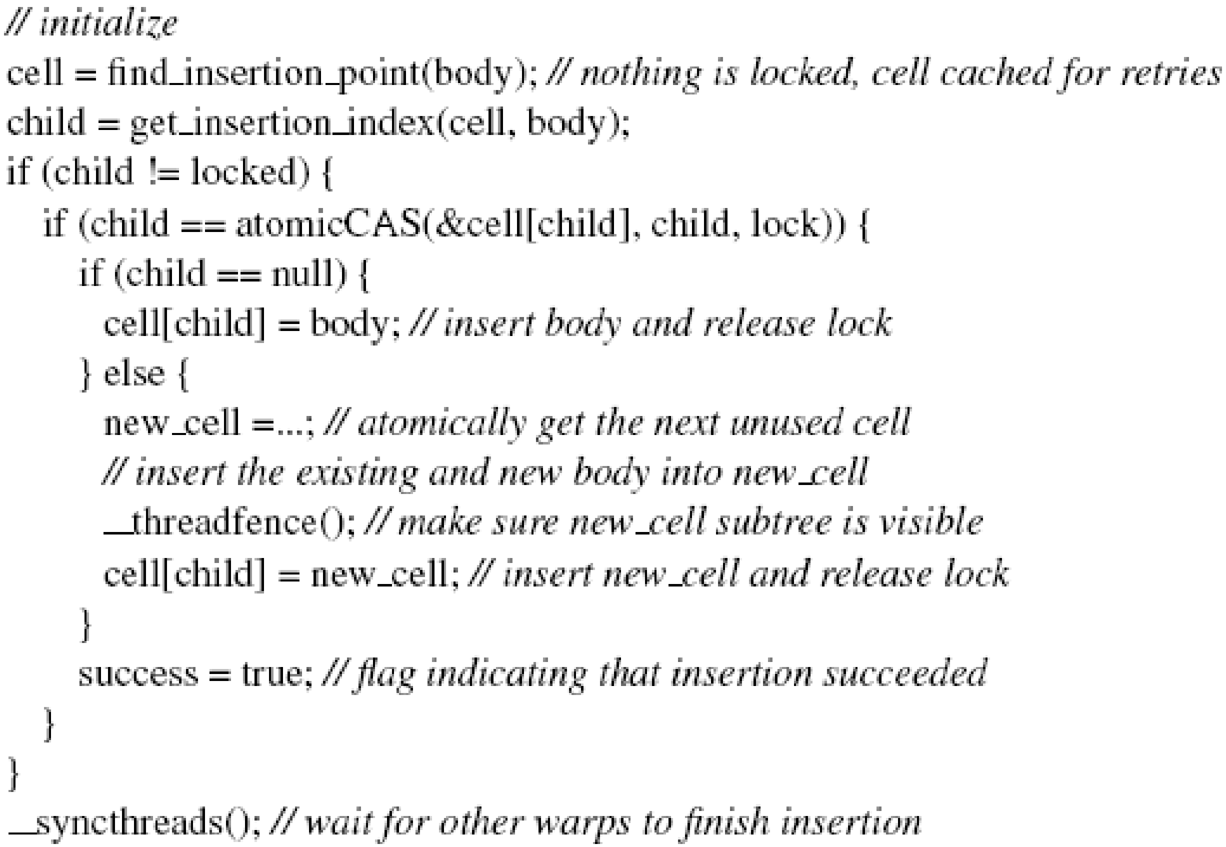
Kernel 2

- 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

- 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.

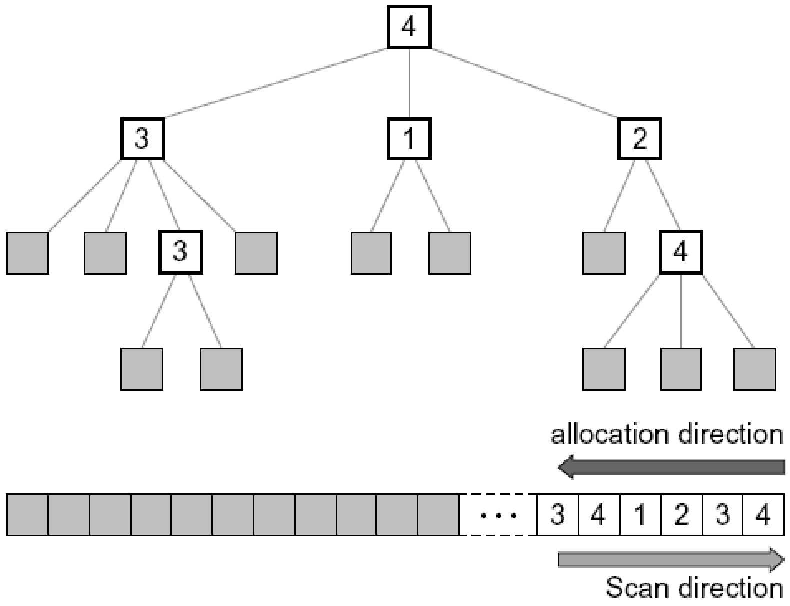
Kernel 2



```
// 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

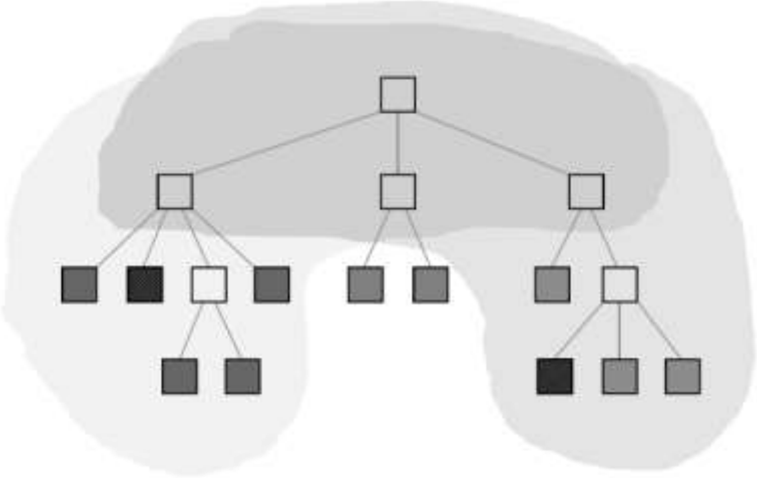


Kernel 3

- 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

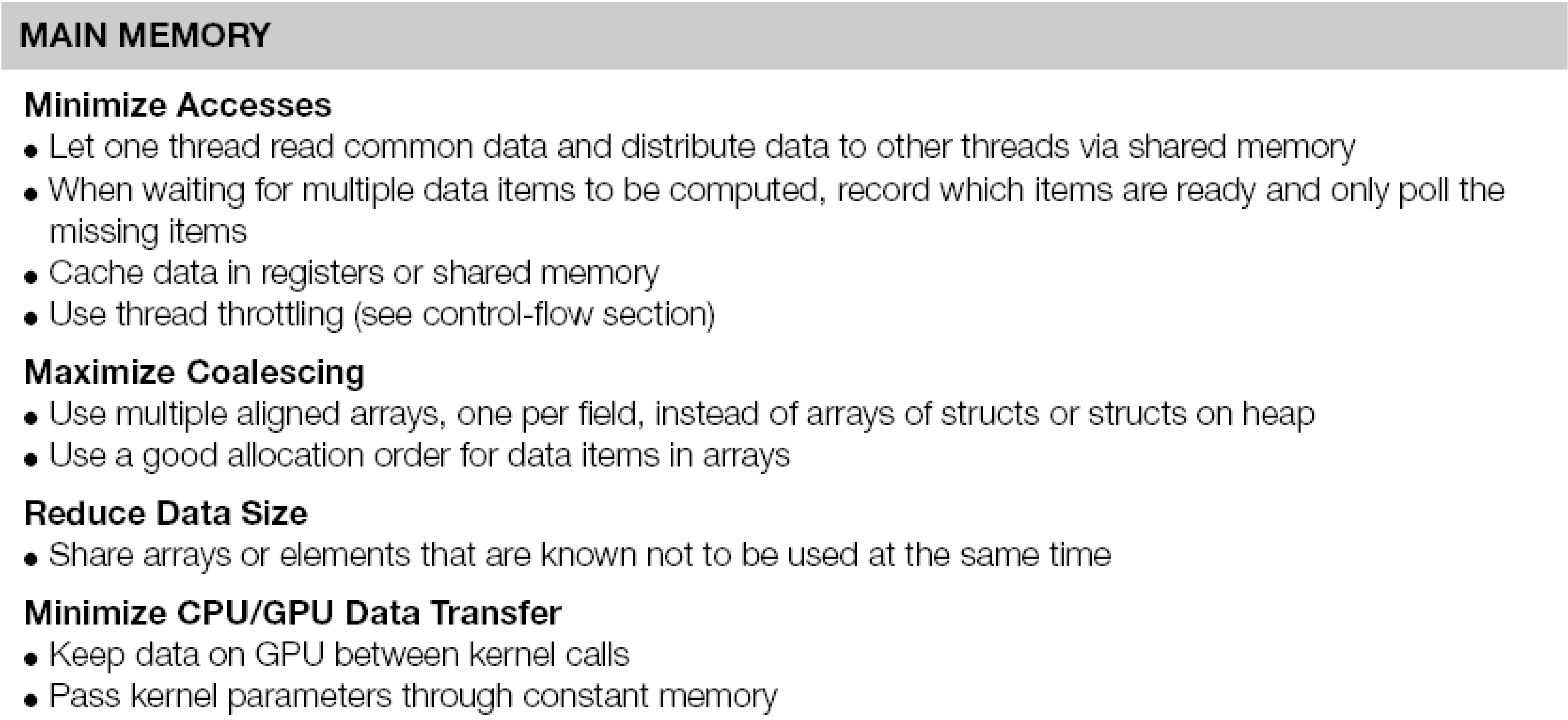
- Requires the vast majority of the runtime
- For each body, the corresponding thread traverses some prefix of the octree to compute the force acting upon this body.



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
- 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 (see control-flow section)

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

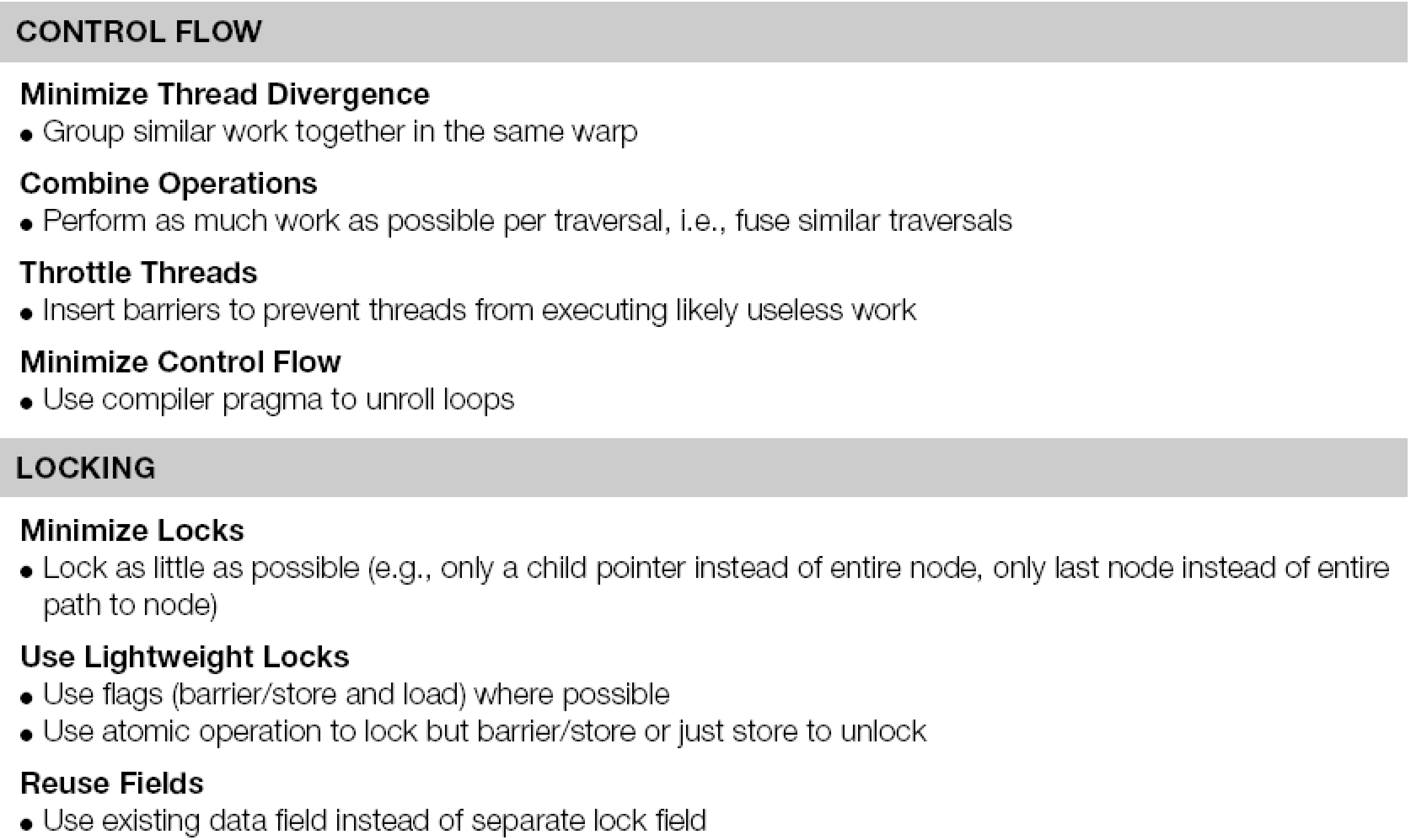
Reduce Data Size

- 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)

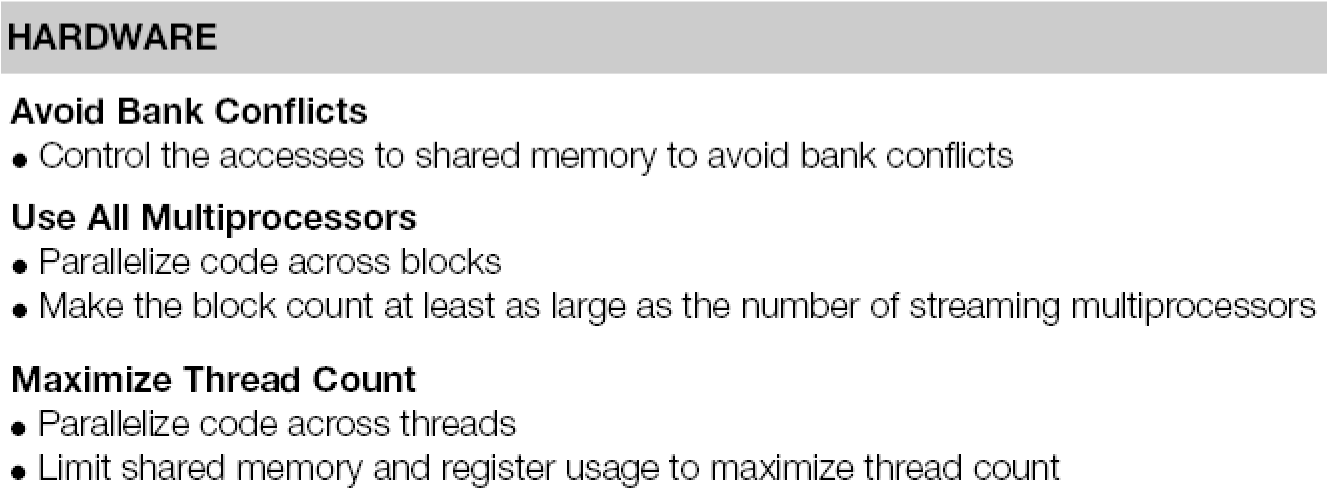
Use Lightweight Locks

- 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

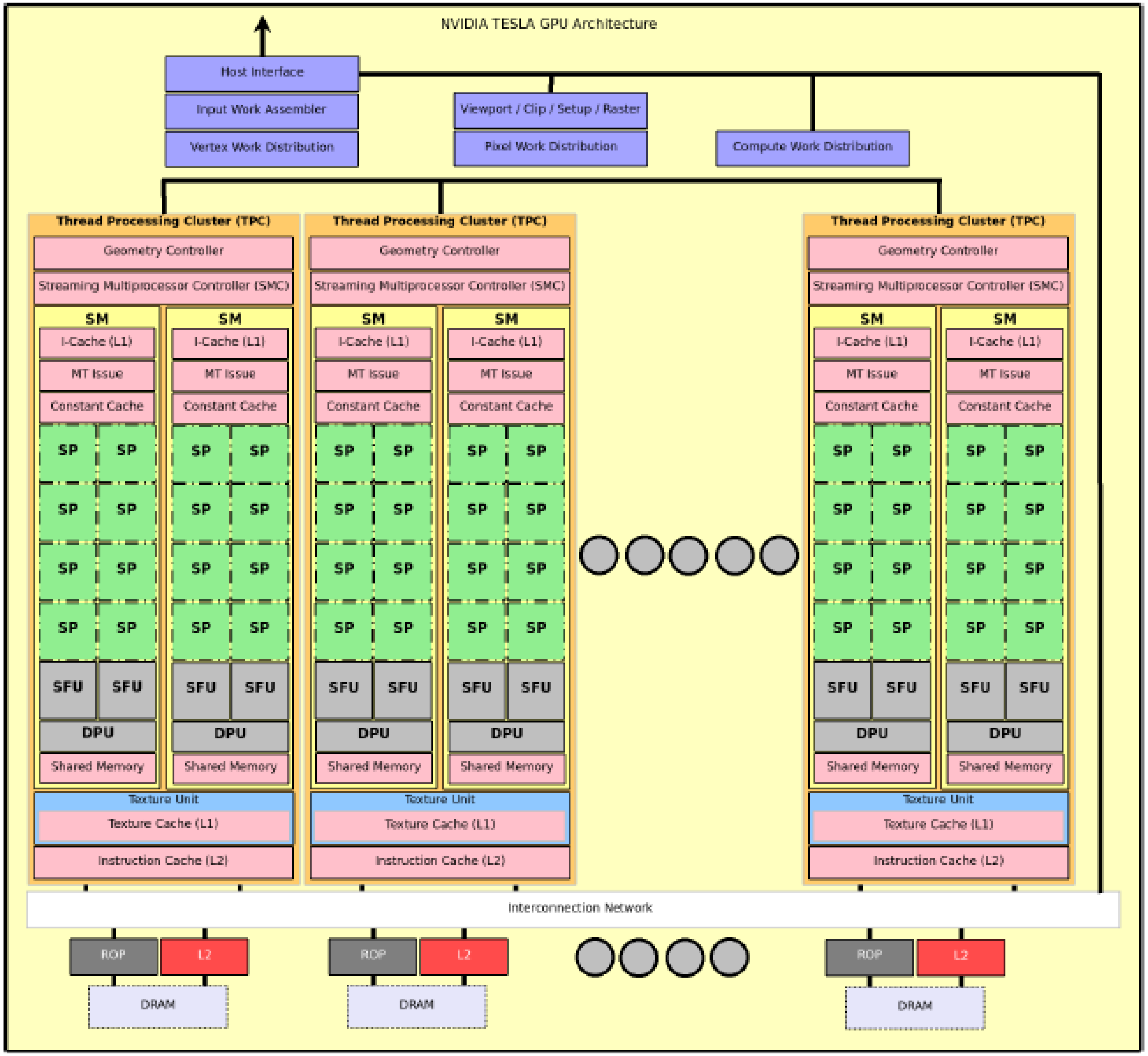
- Parallelize code across threads
- Limit shared memory and register usage to maximize thread count

The Road Ahead

Reliability Issues

- Reliability in GPUs is not as addressed as other aspects
- Graphics applications may be fault tolerant, but other applications running on GPUs are not.
- Large scale GPU failure after shipment/deployment is not uncommon

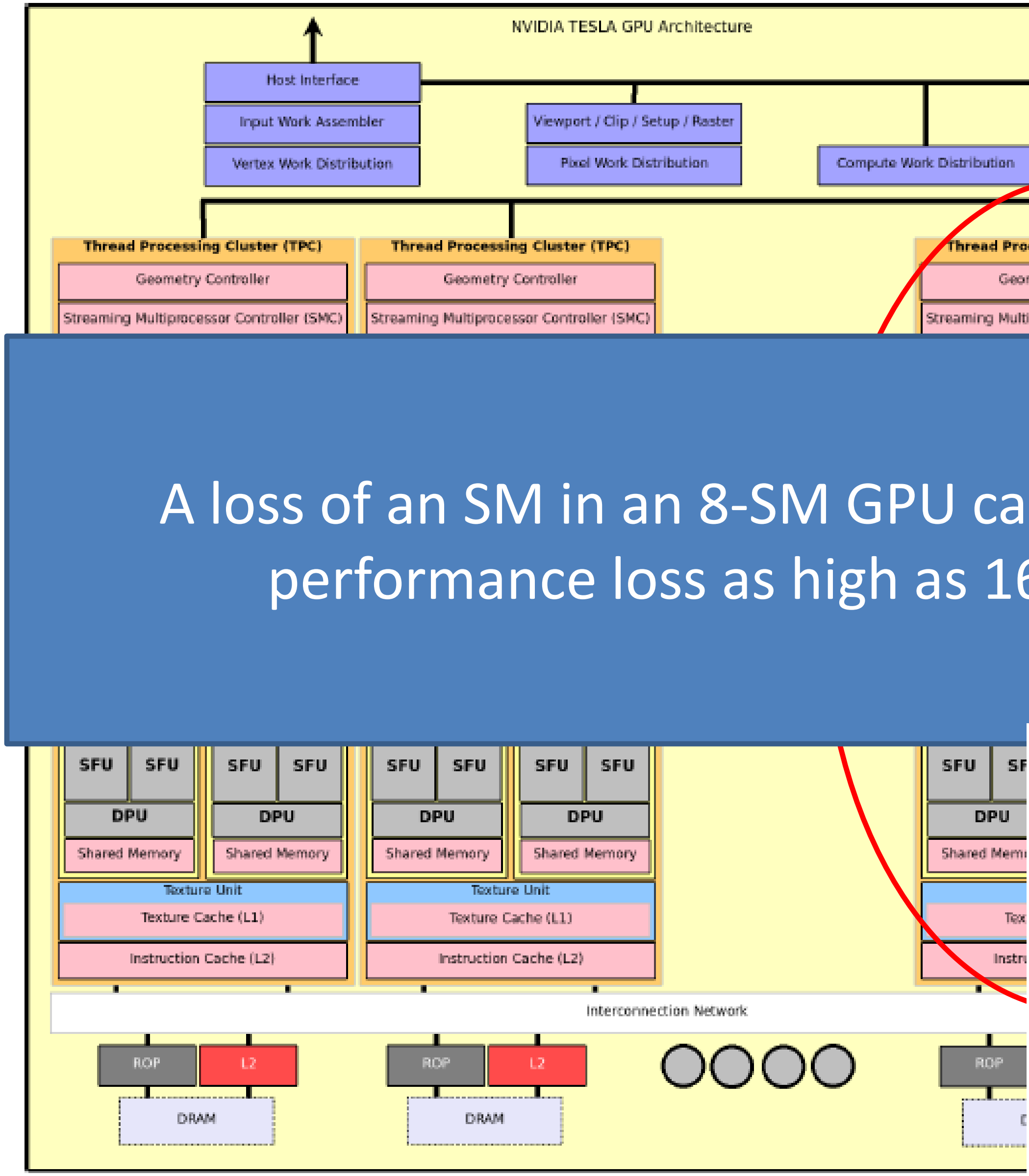
Typical GPU: Massive Parallelism



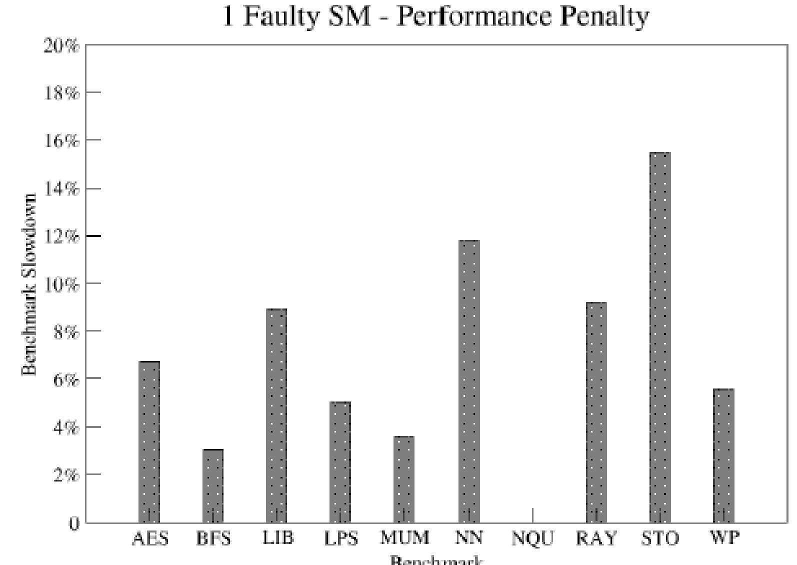
As the number of processing elements increases

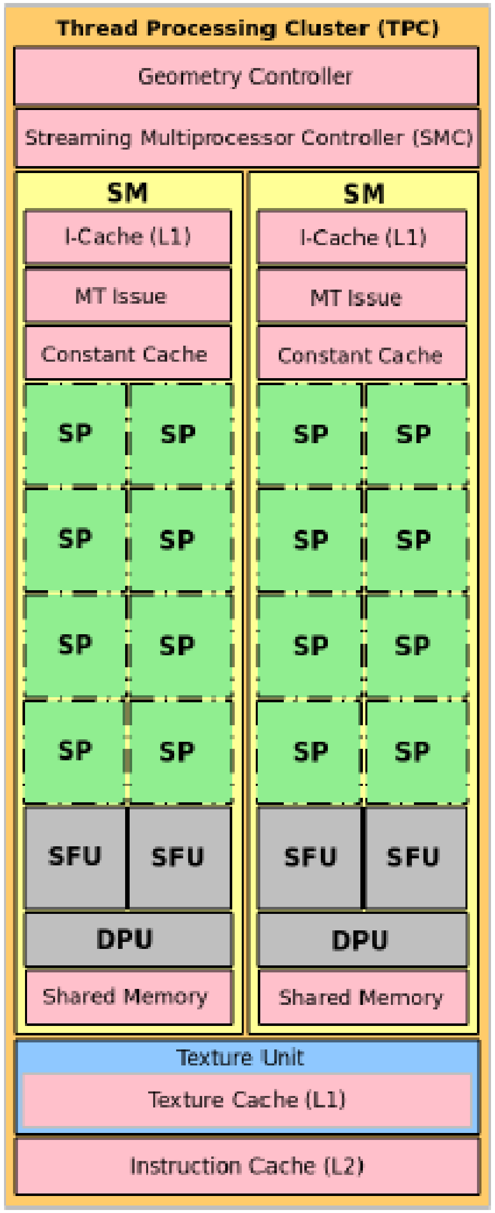
The probability that one or more of them fail increases.

Typical GPU: Massive Parallelism


NVIDIA TESLA GPU Architecture
Host Interface
Input Work Assembler
Vertex Work Distribution
Viewport / Clip / Setup / Raster
Pixel Work Distribution
Compute Work Distribution
Thread Processing Cluster (TPC)
Geometry Controller
Streaming Multiprocessor Controller (SMC)
Thread Processing Cluster (TPC)
Geometry Controller
Streaming Multiprocessor Controller (SMC)
Thread Processing Cluster (TPC)
Geometry Controller
Streaming Multiprocessor Controller (SMC)
SFU
SFU
DPU
Shared Memory
Texture Unit
Texture Cache (L1)
Instruction Cache (L2)
Interconnection Network
ROP
L2
DRAM
ROP
L2
DRAM
ROP

A loss of an SM in an 8-SM GPU can cause performance loss as high as 16%!

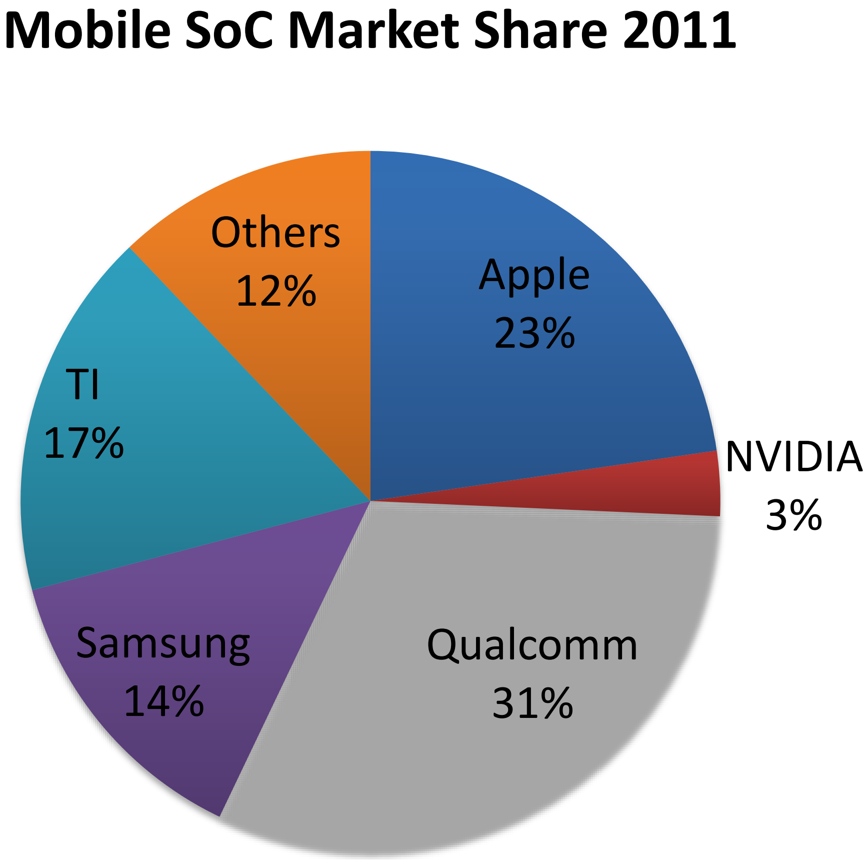




What if one of the SMs fails??

- Turn off the faulty part.
- Giving hints
- Reliability-aware software??

GPUs in Embedded Devices

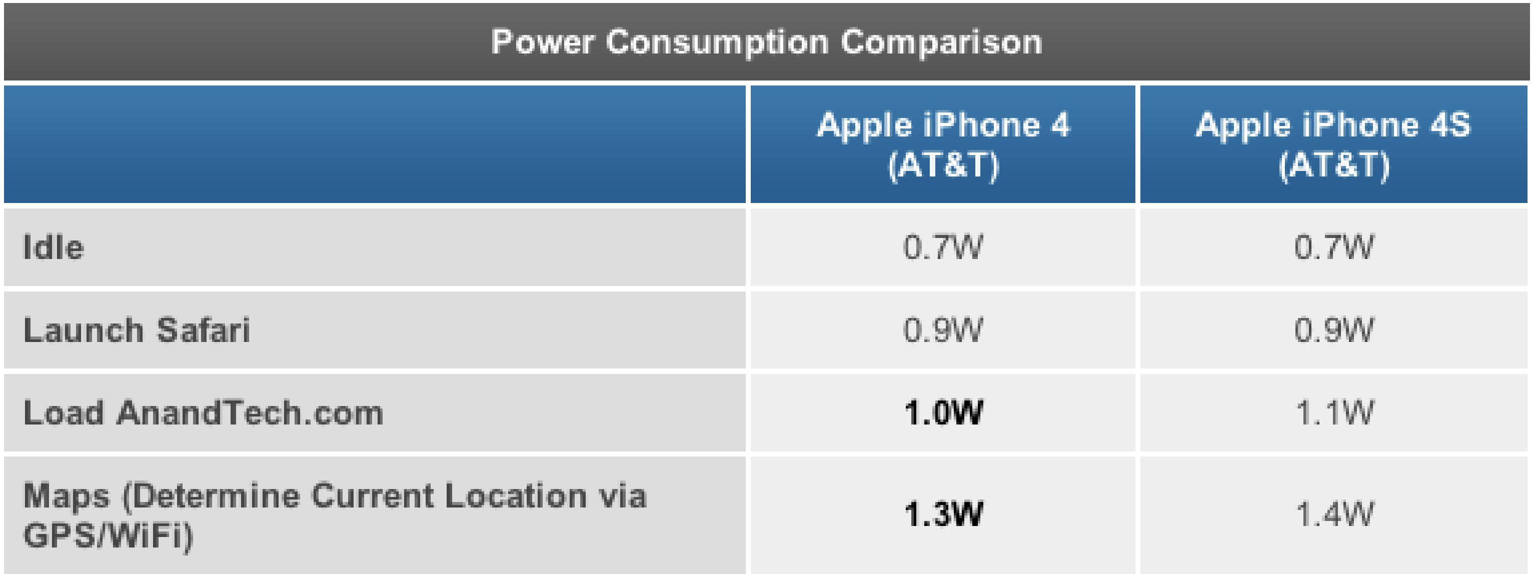


Market Share Data from PC Perspective

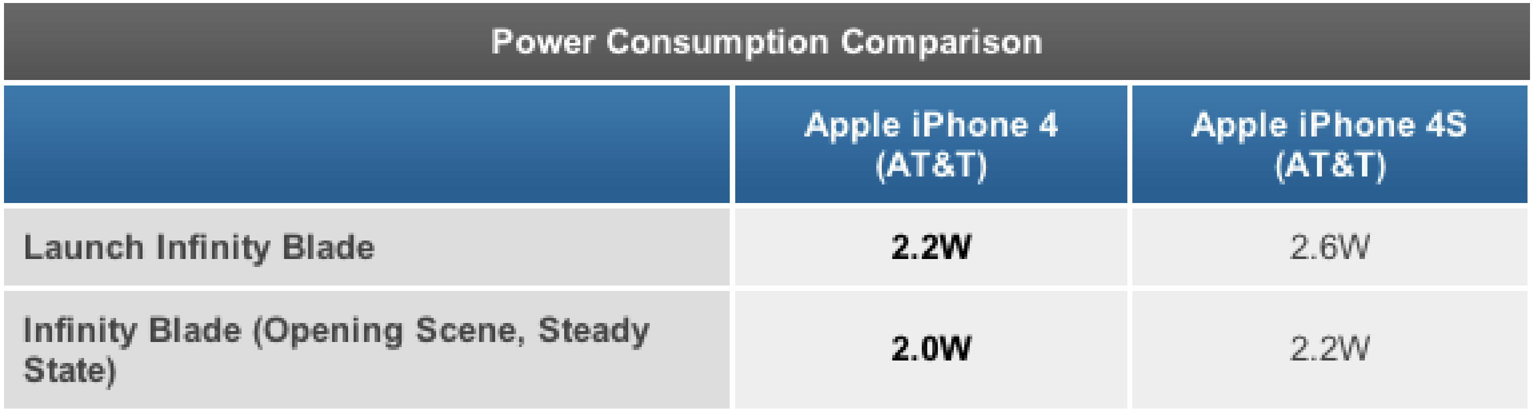
The Constraints of Mobile

- Energy
  - Cell phone battery capacity of 5-7 Watt-hour (tablets 20-40 Wh)
- Area
  - PCB size constraints
  - Cooling constraints

Some Energy Numbers



Power Consumption Comparison		
	Apple iPhone 4 (AT&T)	Apple iPhone 4S (AT&T)
Idle	0.7W	0.7W
Launch Safari	0.9W	0.9W
Load AnandTech.com	**1.0W**	1.1W
Maps (Determine Current Location via GPS/WiFi)	**1.3W**	1.4W



Power Consumption Comparison		
	Apple iPhone 4 (AT&T)	Apple iPhone 4S (AT&T)
Launch Infinity Blade	**2.2W**	2.6W
Infinity Blade (Opening Scene, Steady State)	**2.0W**	2.2W

Data from AnandTech

Some Theoretical Performance Numbers

	Apple iPad 2	ASUS Transformer Prime	Some Nice Desktop
CPU	A5 @ 1GHz	Tegra 3 @ 1.4GHz	Sandy Bridge @ 3.4GHz
GPU	POWERVR SGX543MP2 @ 250MHz	Mobile GeForce @ 500MHz	GTX680 @ 1GHz
Memory Interface	64-bit @ (maybe) 800MHz = 6.4GB/s	32-bit	256-bit @ 6GHz = 192GB/s
GPU GFLOPS	16 GFLOPS	12 GFLOPS	3 TFLOPS

Mobile Data from AnandTech
GTX680 Specs from Newegg

Tegra 2 Mobile GeForce

- Separate vertex and pixel shaders
  - 4 of each, each capable of 1 multiply-add /clock
- Pixel, texture, vertex, and attribute caches
  - Reduce memory transactions
  - Pixel cache useful for UI components
- Memory controller optimizations
  - Arbitrate between CPU & GPU requests
  - Reorder requests to limit bank switching

Case Study: the new iPad

- Screen resolution of 2048x1536
  - Higher than nearly all desktop and laptop displays
- Battery life approximately equal to previous version
- Resolution, power, performance: pick any two.

iPad Gaming Performance

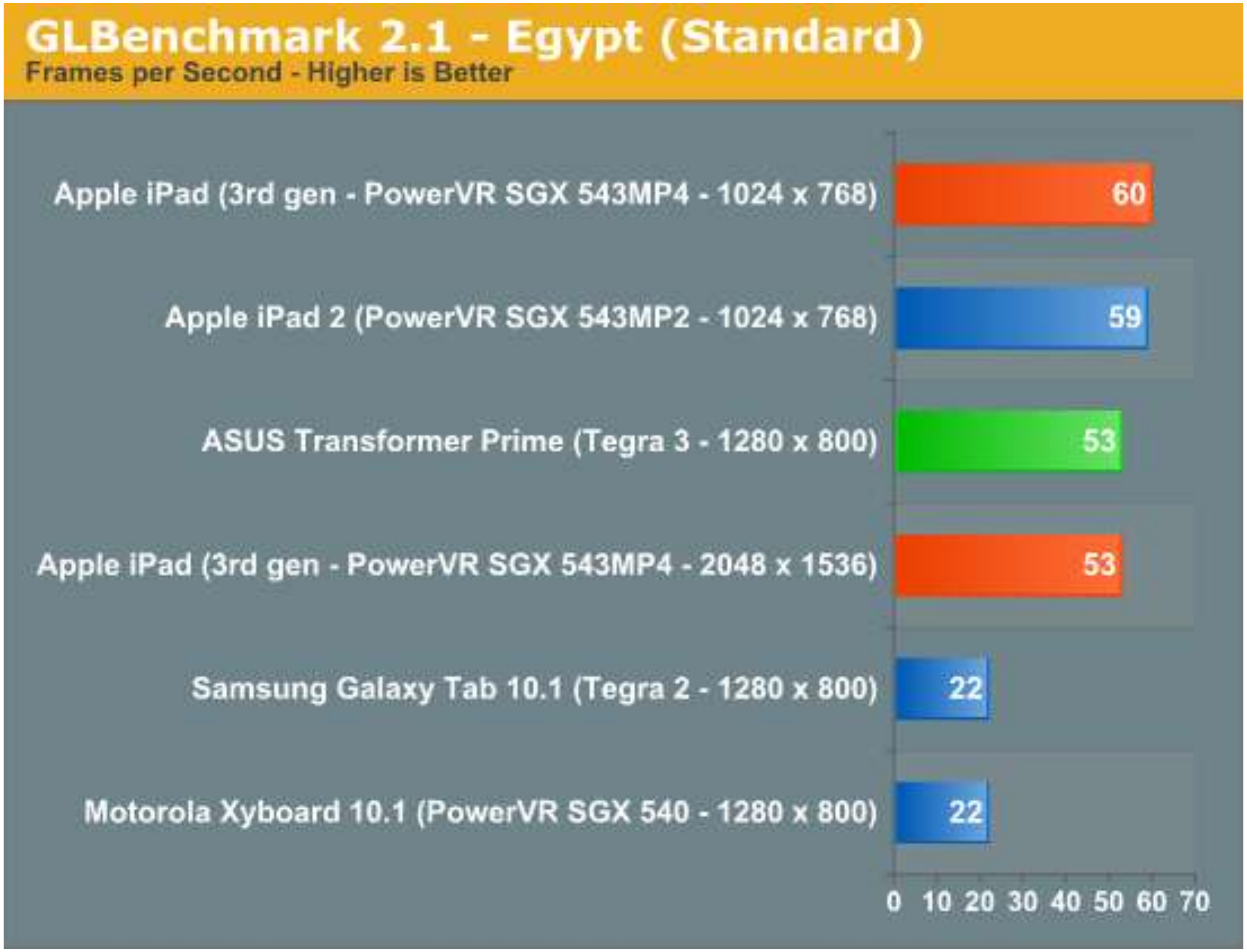
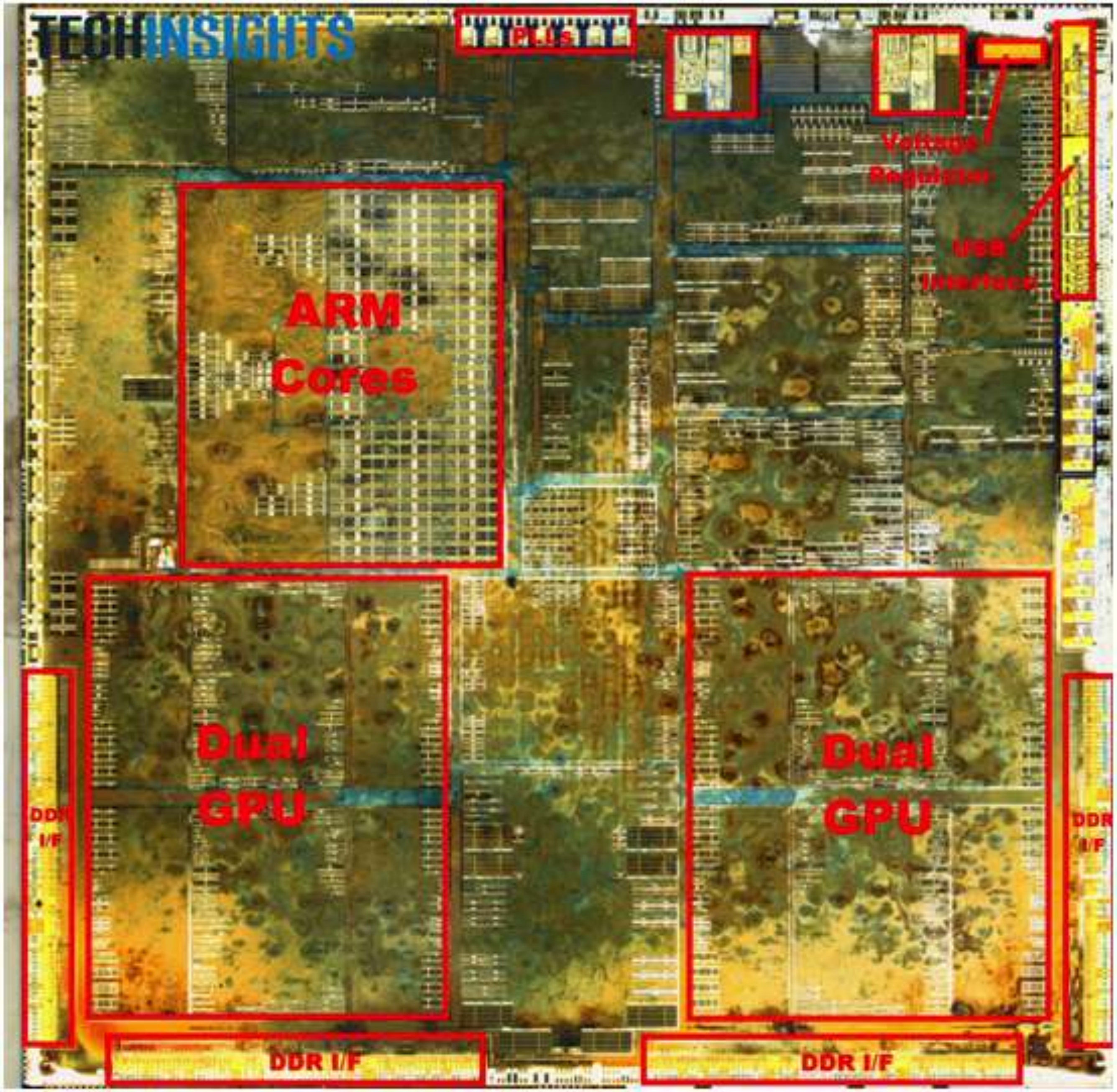


Image from AnandTech



Apple A5X Die Shot

Image from UBMTechInsights

Interesting Question

Do we need GPU for non-graphics application in mobile gadgets?

Conclusions

- When considering your problem:
  - Pick your algorithm
  - Choose the data structure
  - Try to make as many threads and blocks as possible busy
  - Know your hardware
  - Tweaks are inevitable
  - Correctness, performance, and power.