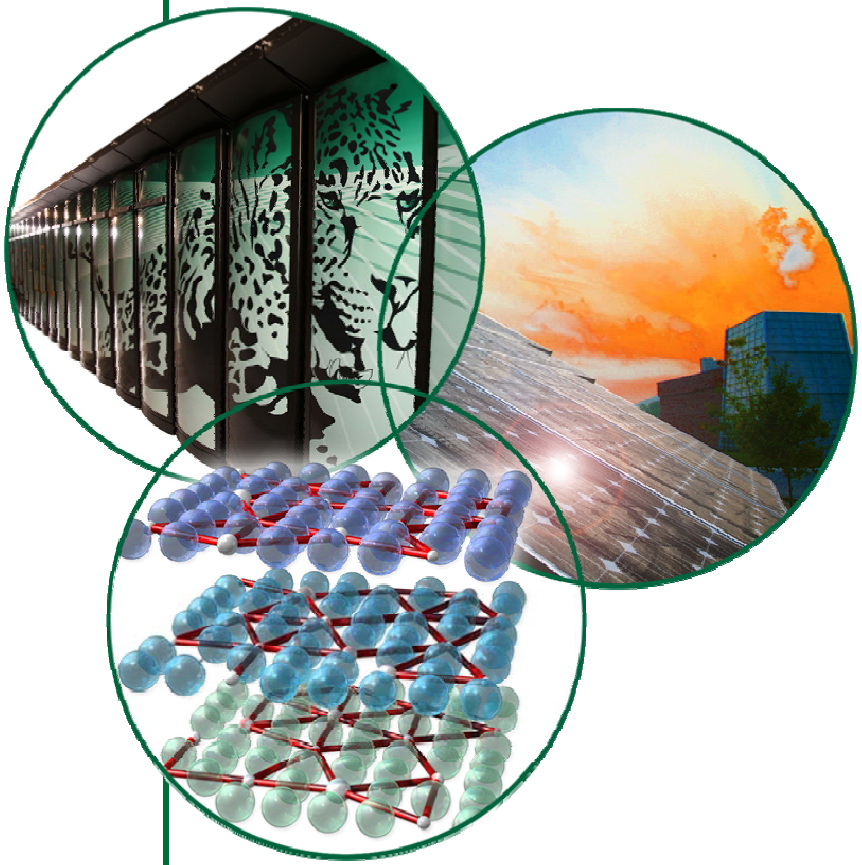
Introduction to Simulations on GPUs

Tutorial
IEEE DS-RT, Singapore

Oct 25, 2009



Kalyan S. Perumalla, Ph.D.

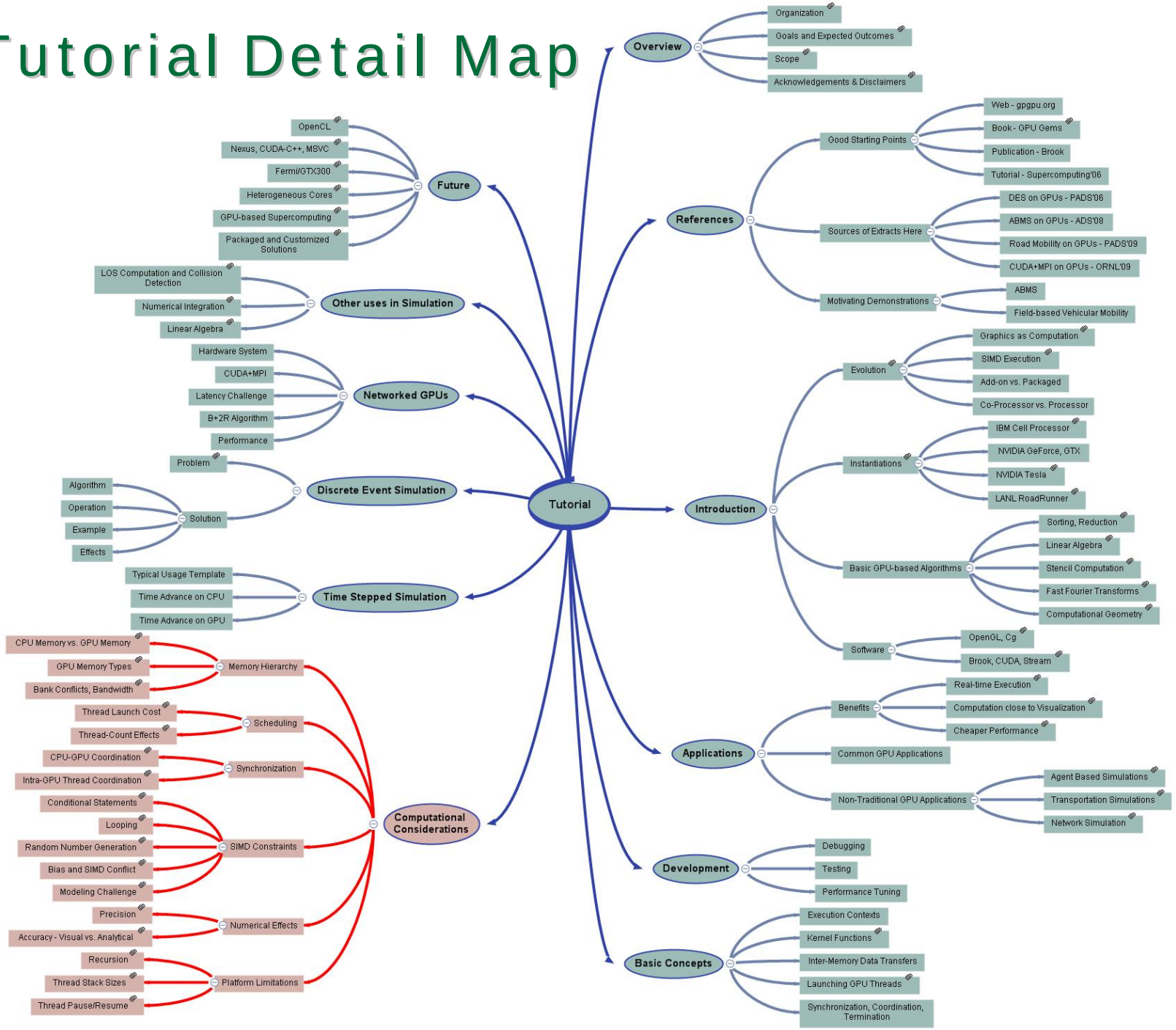
Senior Researcher
Oak Ridge National Laboratory

Adjunct Professor
Georgia Institute of Technology

Goals and Expected Outcomes

- Intended for parallel simulation researchers
- To familiarize with
  - Terminology
  - Essential concepts
  - Important considerations in GPU-based simulation
- Basic concepts presented
  - "Make it work *first*, before making it work *fast*"
- Primary focus on application-level needs and benefits
  - Secondarily on computer science/novelty
- Concepts presented mostly independent of any particular system
  - Due to the rapidly-changing nature of GPU hardware/systems horizon

Tutorial Detail Map



Tutorial Item Sequence

- Overview and References
- Part I
  - Introduction to GPUs
  - Applications
  - Development
  - Basic Concepts
- Part II
  - Computational Considerations
  - Time Stepped Simulation
  - Discrete Event Simulation
- Part III
  - Networked GPUs
  - Other uses in Simulation
  - Future

Scope

- Focus primarily on parallel simulations
- Does not cover the vast GPU literature on non-simulation applications
  - **E.g., data analysis, stream processing, and mathematical programming**
- Aimed at simulation researchers, architects, and developers
  - **E.g., As a quick primer, for research to be later pursued in greater detail**
- Introduces core concepts, terminology, and salient GPU features
  - **Additional detail obtainable on the Web, and from GPU books and publications**

Acknowledgements & Disclaimers

- Sincere thanks to
  - **Prof. Stephen Turner (NTU) and Prof. Wentong Cai (NTU) for tutorial invitation at IEEE DS-RT'09, Singapore**
  - **David Hetrick (ORNL) for institutional support**
- In fond memory of my student intern Brandon Aaby (1986-2009)
  - **the intellectual kid who excelled in many things, including GPU-related research**

References

- A Few Good Starting Points
- Sources of Extracts Used Here
- Motivating Demonstrations

Good Starting Points

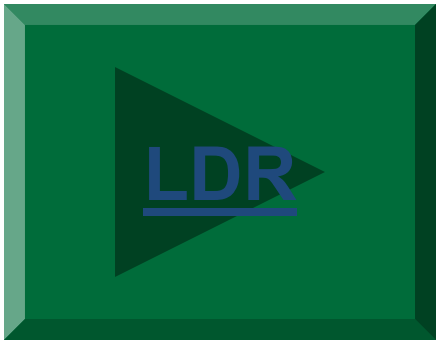
- Web – gpgpu.org
  - www.gpgpu.org is fairly active and contains several pointers
- Book – GPU Gems 2
  - M. Pharr and R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation: Addison Wesley Professional, 2005
  - http://developer.nvidia.com/object/gpu_gems_2_home.html
- Publication – Brook
  - I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, et al., "Brook for GPUs: Stream Computing on Graphics Hardware," ACM Transactions on Graphics, vol. 23, pp. 777-786, 2004
- GPU Tutorial - Supercomputing'06

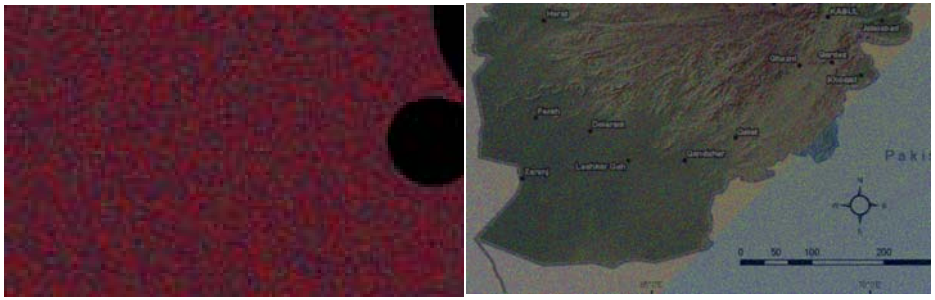
Sources of Extracts Used Here

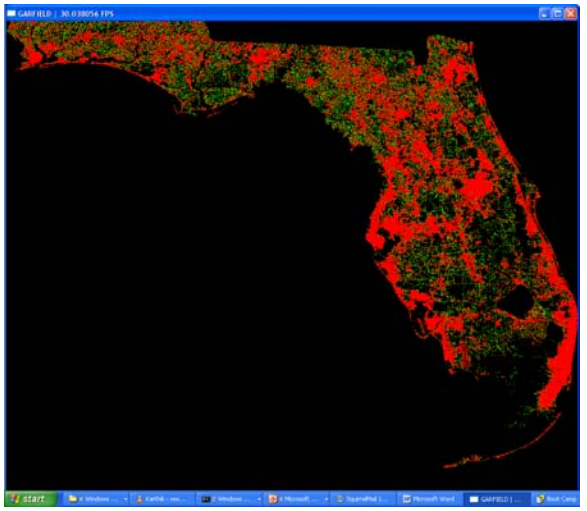
- DES on GPUs - PADS'06
  - Perumalla, K. S. (2006). “Discrete Event Execution Alternatives on GPGPUs.” Int’l Workshop on Principles of Advanced and Distributed Simulation
- ABMS on GPUs - ADS'08
  - Perumalla, K. S. and B. Aaby (2008). “Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs.” Agent-Directed Simulation Symposium (Spring Simulation Multi-Conference)
  - Best Paper Award Winner
- Road Mobility on GPUs - PADS'09
  - Perumalla, K. S. B. Aaby, S. Yoginath, and S. Seal (2009). “GPU-based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios.” Int’l Workshop on Principles of Advanced and Distributed Simulation
- CUDA+MPI on GPUs - ORNL'09
  - Aaby, B., and K. S. Perumalla (2009). “Parallel Agent-Based Simulations on Clusters of Multi-GPUs and Multi-Core Processors.” Oak Ridge National Laboratory

Motivating Demonstrations

- ABMS
  - Game Of Life
  - Afghan Leadership
- Field-based Vehicular Mobility
  - Florida State

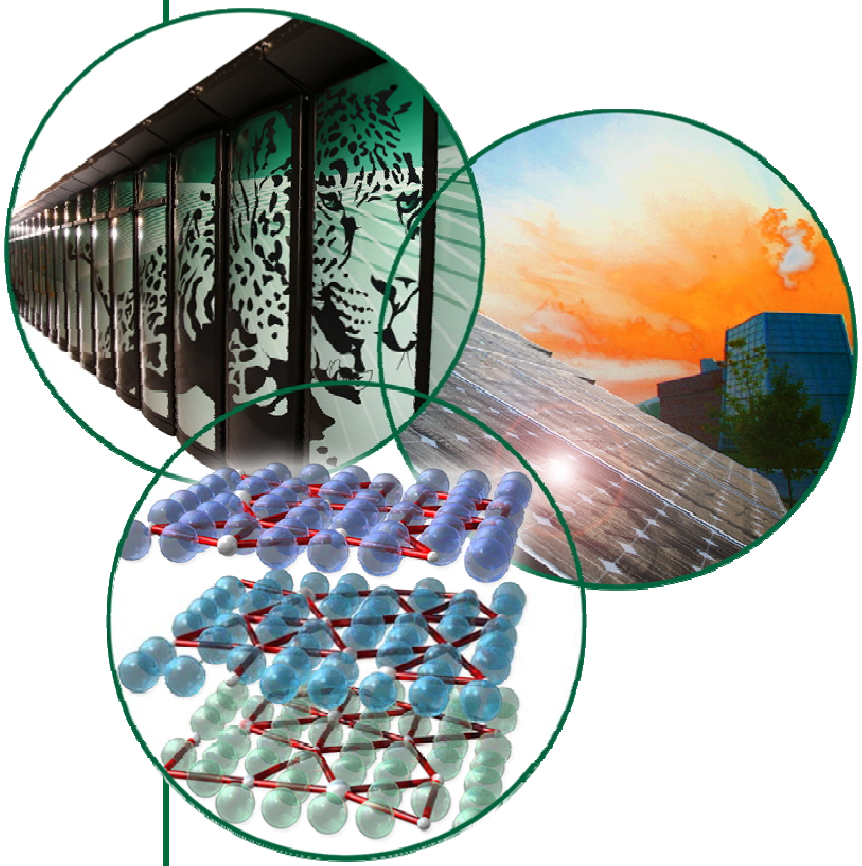






Part I

- Introduction
- Applications
- Basic Concepts

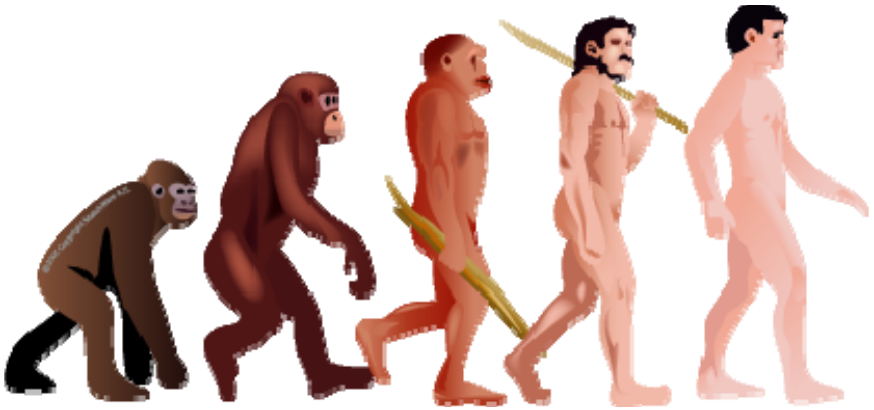


Introduction

- Evolution
- Instantiations
- Basic GPU-based Algorithms
- Software

Evolution

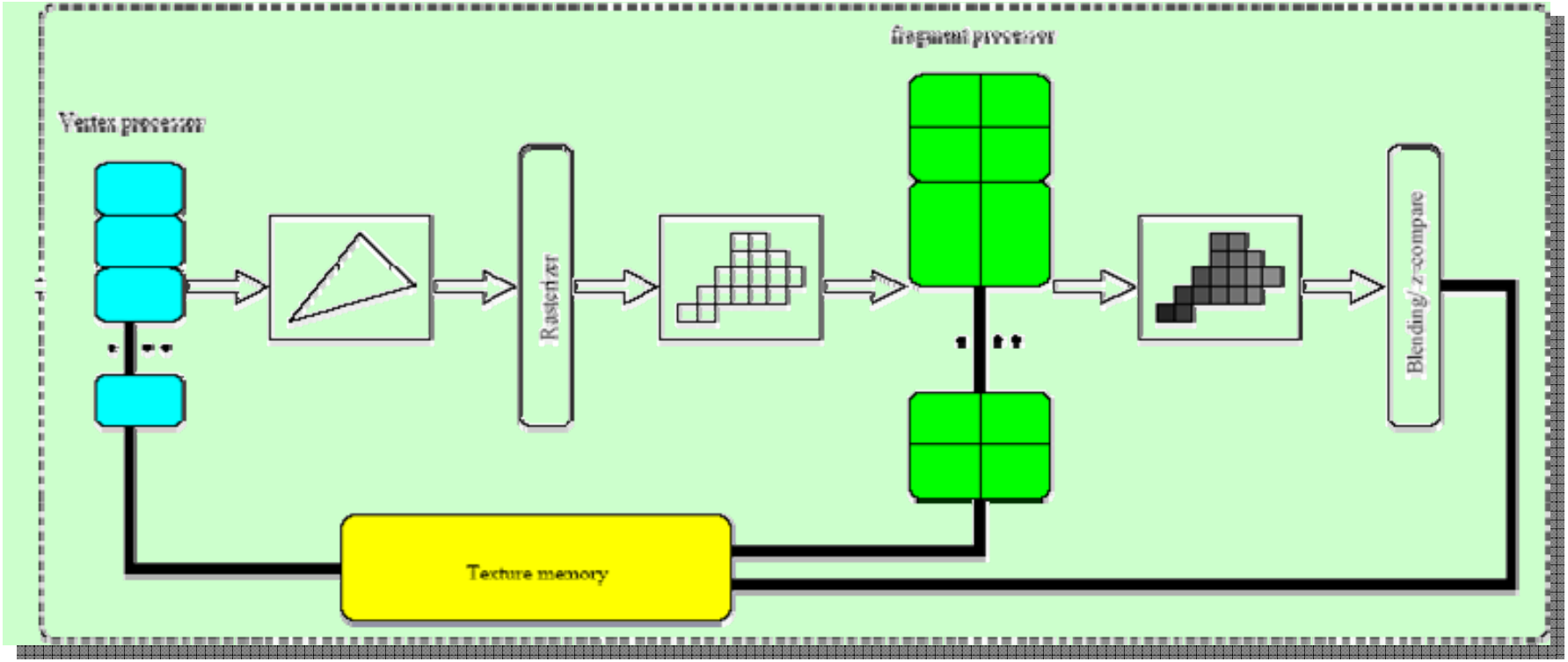
- Graphics as Computation
- SIMD Execution
- Add-on vs. Packaged
- Co-Processor vs. Processor



Graphics as Computation

Programmable graphics primitives

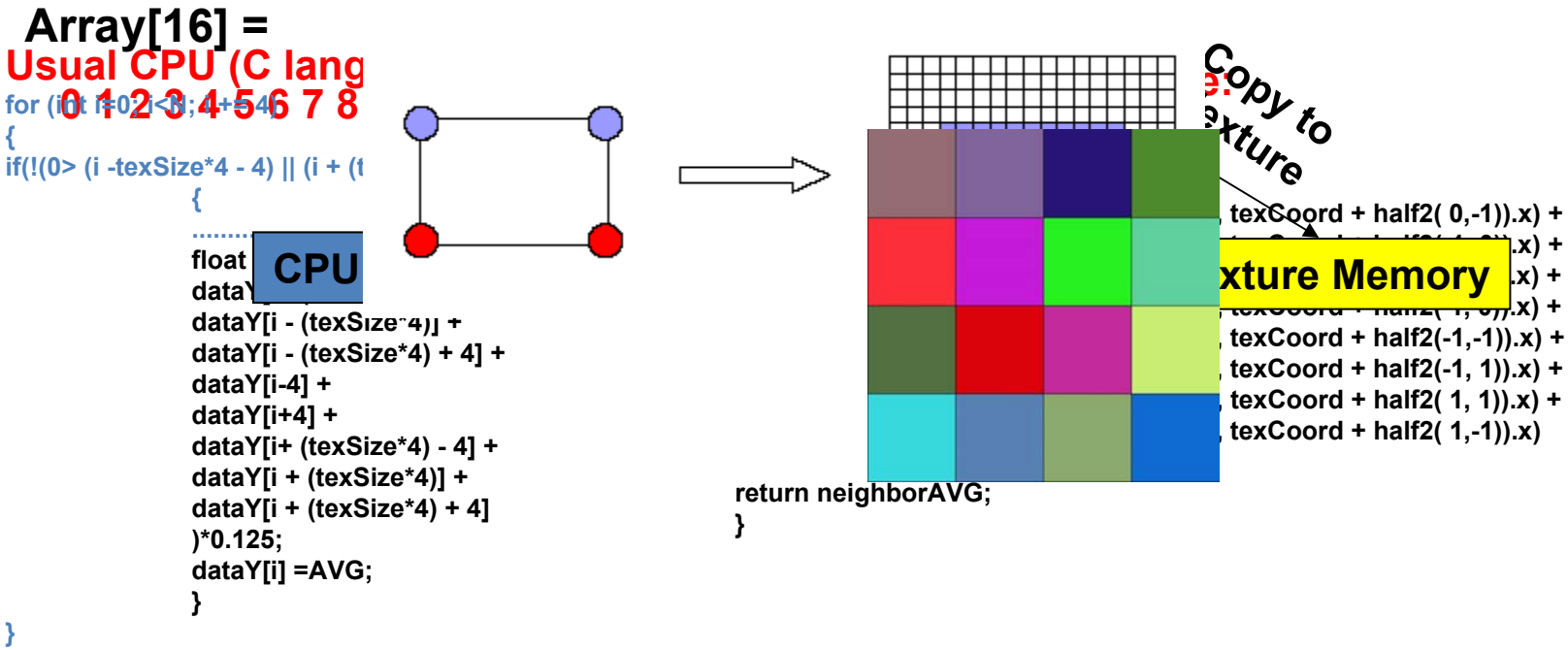
E.g., pixel shading such as bump mapping and patterned texture mapping



Computation on GPUs (Pre-CUDA)

Mapping computational concepts to graphics

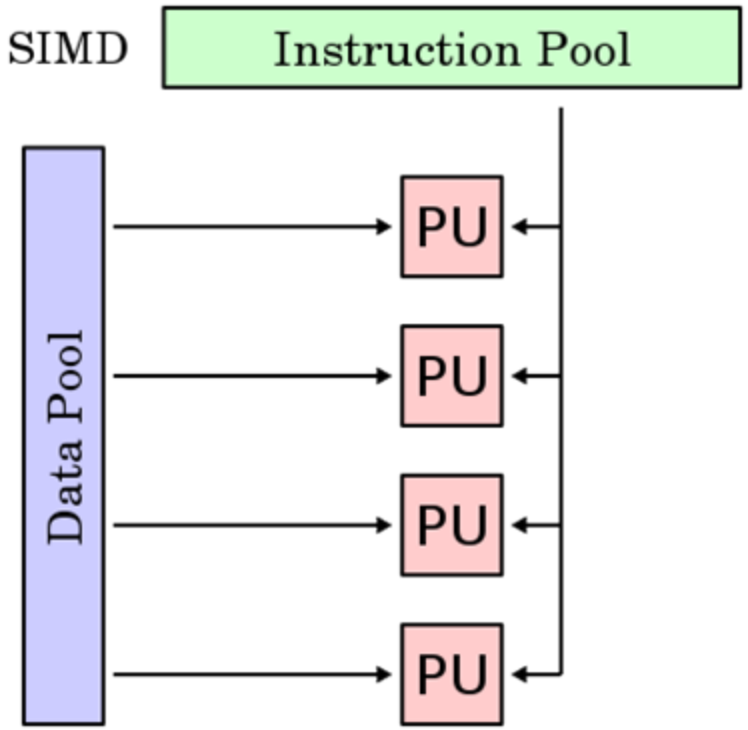
- Array => Texture
- Kernel => Fragment Program
- Feedback => Copy To Texture (Read Pixels w/ FBO)
- Data Stream => Draw Graphic



SIMD Execution

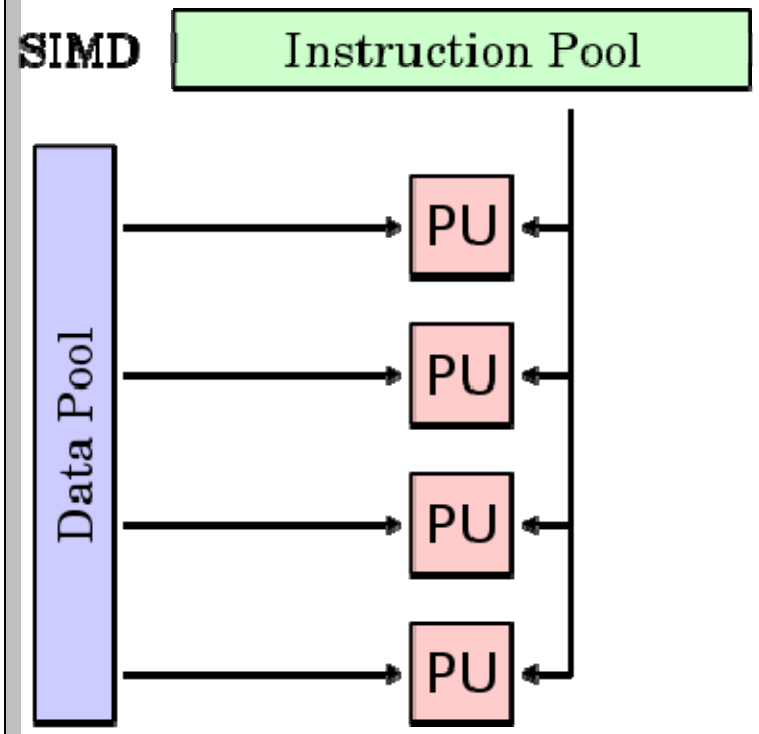
Single Instruction Multiple Data (SIMD) is a style of parallel execution

- Identical operation A is executed on multiple processors simultaneously
- Each operation *A[p]* on processor *p* operates on a distinct data set *M[p]*
- Conceptually, the operation *A* is complete only when all processors complete their operation *A[p]* on *M[p]*



SIMD Execution (continued)

- Excellent paradigm for processing large data sets
  - **For items that are mostly independent from each other yet undergo identical processing**
- However, important implications to bear in mind
  - **If *A[p]* takes different amount of time depending on *M[p]*, then the time to complete *A* is the maximum time among all *A[p]***
  - **If *M[p1]* and *M[p2]* overlap for some processors *p1* and *p2*, then (a) results may be undefined, depending on read-write hazards, and (b) *A[p1]* and *A[p2]* may be serialized, thereby decreasing performance**



Add-on vs. Packaged: To Get Started

- GPU-based systems can be built by adding a programmable GPU to a system
  - User can enhance a conventional CPU-based system
  - User needs to customize installation (software, drivers, etc.)
  - E.g., purchase NVIDIA GTX 300, install to a high-end desktop

- Alternatively, complete, customized systems available
  - Properly packaged with the best chipsets, cooling & power supplies, drivers, software, and development environments
  - Often, better value for price, if high-end configuration needed
  - E.g., NVIDIA Tesla

Co-Processor vs. Processor

- Tail wagging the dog?
  - Conventional CPU-processor and GPU co-processor relation
  - In some applications, GPU may be the main workhorse
- GPU may have to be viewed as semi-equal to CPU
  - Trends indicate they will merge (see “Future”)

- Note, GPUs are similar to other co-processors
  - Network co-processors
  - Physics co-processors
- However, GPUs have one major advantage
  - Mass consumer market: end-users, multi-media, and gaming industries

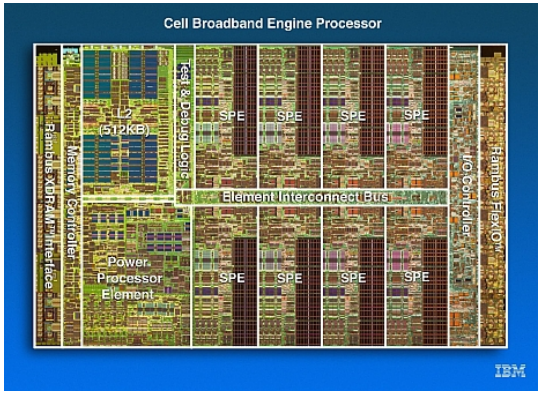
Instantiations – A Few Popular Examples

Commercial Offerings

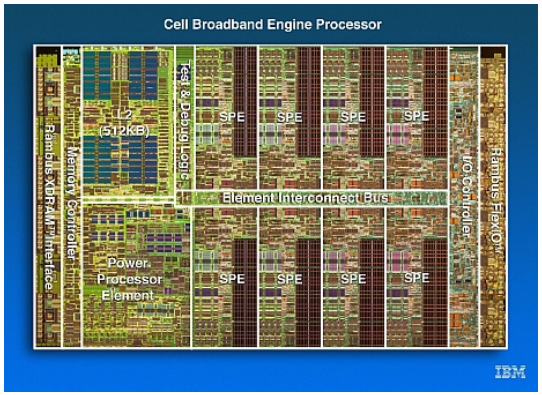
- IBM Cell Processor
- NVIDIA GeForce, GTX
- NVIDIA Tesla

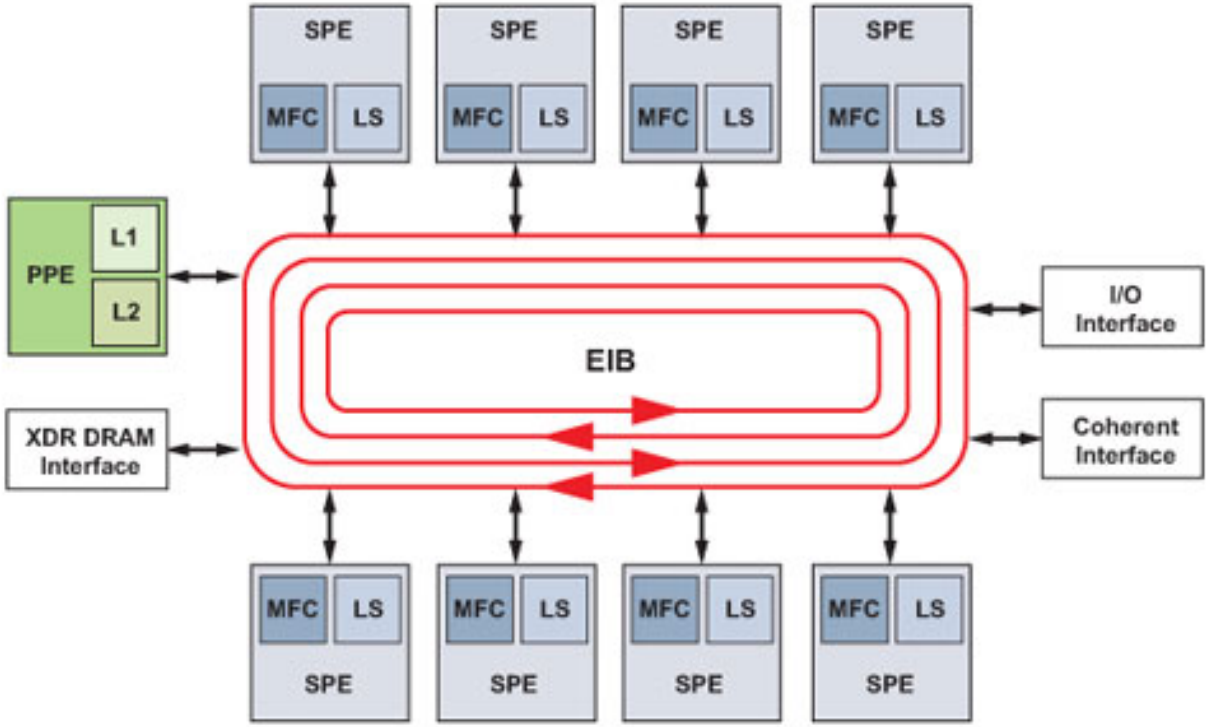
Supercomputing Scale

- LANL RoadRunner



IBM Cell Processor





- Highly successful, path-blazing chip
- Large market use
  - Entertainment systems
  - Gaming systems (Sony playstation)

NVIDIA GeForce, GTX

- Original GeForce 6000, 7000, 8000, and 9000 series
- Latest GTX 295
- Upcoming GTX 300

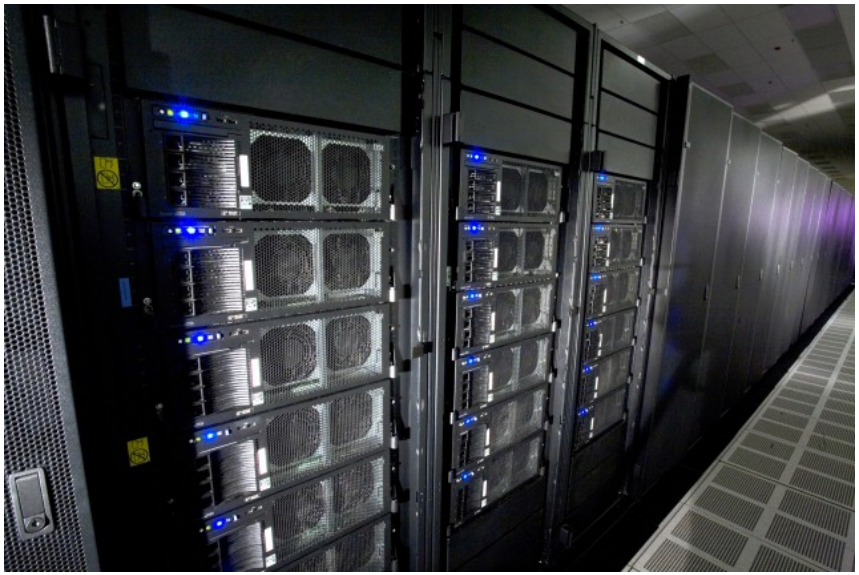
- Affordable, off-the-shelf
- Power-hungry!
  - Double PCI-e power connections
- Heat generating
  - Special cooling needs
  - E.g., liquid-cooled gaming systems

NVIDIA Tesla

- An example of a packaged solution
  - Single seamless, finely-tuned system with multiple GPUs and software environment
- "Tera-FLOP under your desk"

LANL RoadRunner

- Peta-FLOP supercomputer at the Los Alamos National Laboratory
- System based on IBM Cell processor (and AMD Opteron) architecture
- Topped the supercomputing charts in 2008
- SIMD co-processor execution realized in the extreme
- A few niche applications
  - E.g., Molecular Dynamics



Basic GPU-based Algorithms

- Sorting, Reduction
- Linear Algebra
- Stencil Computation
- Fast Fourier Transforms
- Computational Geometry
- …

Sorting, Reduction

- Much of the early GPGPU work
  - Focused on effective sorting and reduction on GPUs
- Well understood implementation
  - Sequence of cumulative optimizations for best performance
- Different realizations
  - Early implementations in Brook (e.g., reduce keyword)
  - Later architecture-aware, efficient realizations in CUDA
- Essentially based on recursive, data-parallel formulations

- Reduction operation is an important building block
- Any commutative, associative operator applied on multiple data (array)
  - E.g., Min, Max, Sum
- Quick data-parallel sweep after independent data-parallel operations
- In discrete-event simulation on GPUs, essential for min-time computation
- Support exists for very fast, highly optimized implementations
  - Language-level, or library-based

- Reductions very useful in debugging large simulations
  - E.g., Verify conservation of persons (live+dead), or flux (heat), etc.

Linear Algebra

Benefits

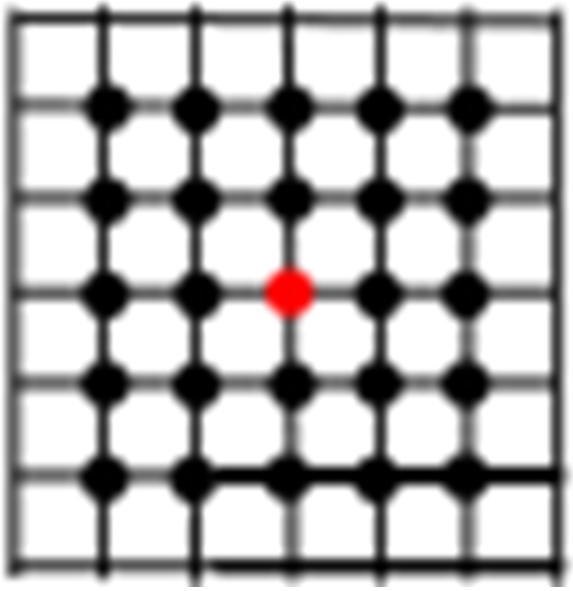
- Good use of co-processor and additional memory
- Useful when single precision (32-bit) arithmetic is sufficient
- Mixed-precision linear algebra possible by combining GPU and CPU

Examples

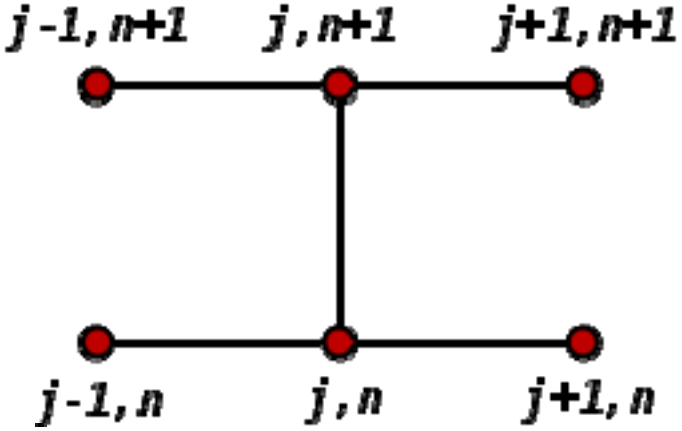
- E. S. Larsen and D. McAllister, "Fast Matrix Multiplies using Graphics Hardware," in Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, 2001
- Jens Kruger, "Linear Algebra on GPUs," in ACM SIGGRAPH 2005 Courses, 2005.

Stencil Computation

- Much of physical (mesh/cell-based) models computation built on “stencils”
  - E.g., fluid dynamics simulations, FDTD
- Highly suited for GPUs
  - Great locality, read-only neighborhood, immense data-parallelism
- Performance gains from historic design
  - Optimizations for textures and programmable surfaces



- Very useful in agent-based simulation on GPUs
- Stencil for neighborhood-based state updates
- But, additional functionality needed (for birth/death, mobility)



Fast Fourier Transforms

- 2-D and 3-D FFT Computations
- Much attention recently on highly optimized FFT using SIMD architectures
- Most action is in maximizing the use of available bandwidth

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1.$$

- Examples: Two publications in Proceedings of Supercomputing'08
- Remark: Power-of-2 vs. Non-power-of-2 performance can be quite different

Computational Geometry

- Exploiting certain hardware features of GPUs
  - E.g., depth-based culling
- For speeding up computational geometry problems
  - E.g, Vornoi diagrams, and distance fields

Software

- OpenGL, Cg
- Brook, CUDA, Stream

OpenGL, Cg

- Combination of OpenGL (Open Graphics Language) and Cg (C for graphics)
- Re-used for general-purpose computation on GPUs

Example of Cg Kernel Code

```
half4  Phase1Kernel
(
    half2 texCoord   : TEXCOORD0,
    uniform half4      globalConstants,
    uniform samplerRECT stateTex,
    uniform samplerRECT constantsTex,
    uniform samplerRECT scratchTex,
    uniform samplerRECT constantsTex2,
    uniform samplerRECT scratchTex2
) : COLOR
{
    half4 OUT = h4texRECT(stateTex, texCoord);
    half surCount = (
          ( h4texRECT(stateTex, texCoord + half2( 0,-1)).x) +
          ( h4texRECT(stateTex, texCoord + half2(-1, 0)).x) +
          ( h4texRECT(stateTex, texCoord + half2( 0, 1)).x) +
          ( h4texRECT(stateTex, texCoord + half2( 1, 0)).x) +
          ( h4texRECT(stateTex, texCoord + half2(-1,-1)).x) +
          ( h4texRECT(stateTex, texCoord + half2(-1, 1)).x) +
          ( h4texRECT(stateTex, texCoord + half2( 1, 1)).x) +
          ( h4texRECT(stateTex, texCoord + half2( 1,-1)).x)
        );

    OUT.y = surCount;
    OUT.z = OUT.x;
    OUT.x = (OUT.x) ?   (surCount <= 1 || surCount >= 4) ? 0.0 : OUT.x):((surCount == 3 ) ? 1.0 : OUT.x);

    return OUT;
}
```

Kernel arguments
-Textures and constants

- Game of Life Cg Kernel
- “C like” Language
- Kernel executed on all texture RGBA *texels*

Computation
-Query Moore neighborhood for live cells

Return value
-Ternary operator

Brook, CUDA, Stream

- Brook language from Stanford served as trailblazer
  - Automatically generated code for combinations of compiled and interpreted execution on GPU
  - Supported multiple runtime interfaces for GPU (DirectX, OpenGL, Emulated)
- CUDA generalized more, enhanced, abstracted, and standardized several of Brook's features
- Other stream processing languages and runtimes appeared (and largely disappeared!)

NVIDIA
GPU Computing
CUDA
By JeGX ;-)

NVIDIA CUDA

- NVIDIA's successor to Cg and GPGPU research
- Compute Unified Device Architecture (CUDA)
  - "C like" in syntax and structure
  - Allows asynchronous gather/scatter unlike Cg
  - Provides relatively low-level access
    - On-chip "shared" memory and registers for fast reads/writes as well as constant texture memory and off-chip global memory
  - Exposes notion of concurrent threads in "thread blocks"
  - Multi-GPU support

NVIDIA CUDA Installation Outline

- Requires NVIDIA 8 series card or newer, e.g. 8800GTX
- Download appropriate driver (includes CUDA support)
- Install CUDA Toolkit and SDK
  http://www.nvidia.com/object/cuda_get.html
  - All projects in Microsoft Visual Studio format (Windows)
  - Makefiles and examples included for *nix
- 32-bit and 64-bit architecture support
- New projects can be built off of template project included in the SDK

Applications

- Benefits for Parallel Simulations
- Common GPU Applications
- Non-Traditional GPU Applications

Benefits
(in Context of Parallel Simulations)

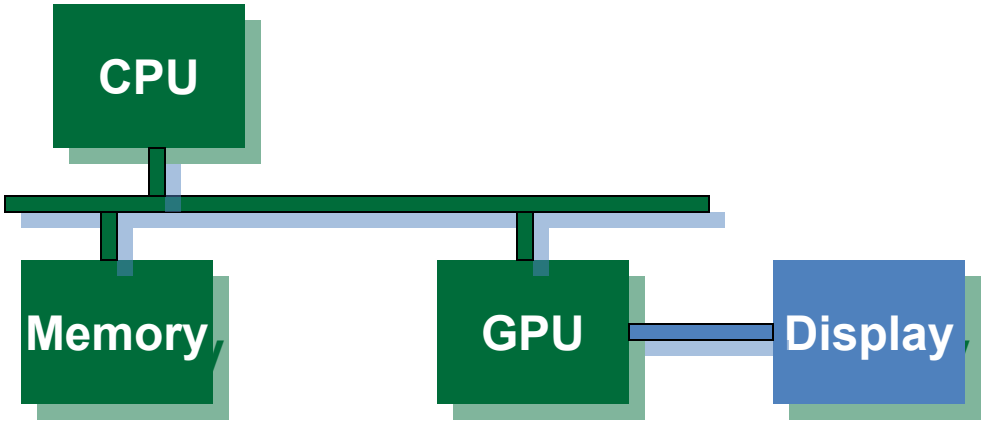
- Real-time Execution
- Computation “Close to Visualization”
- Cheaper High-Performance

Real-time Execution

- Analogous to what digital signal processing did for their applications (image/speech processing)
- Fast execution, approaching many “frames per second”
- Interactive visual simulations possible
  - Freeing CPU for user interface and customization activities
  - Example: Real-time or faster execution of vehicular transport simulations on large networks
- Applications can be run on end-user, low-end machines

Computation "Close to Visualization"

- A natural benefit: computation done closest to display
- Little data transfer overhead (from simulation memory to display frame buffer) compared to CPU-based simulation
- Post-processing for *customized* visualization and animation naturally possible



Cheaper High-Performance

- Much literature debates the performance to cost ratio
- Majority believes GPU is cheaper than CPU
  - But this may change
  - Or is already changing (E.g., multi-core processors)
- Much of the ratio difference due to market economics than fundamental technical reasons
- For niche (highly data-parallel) applications, GPU is certainly better
  - Cheaper by up to 1 order, faster by up to 2 orders, in some demonstrations

Common GPU Applications

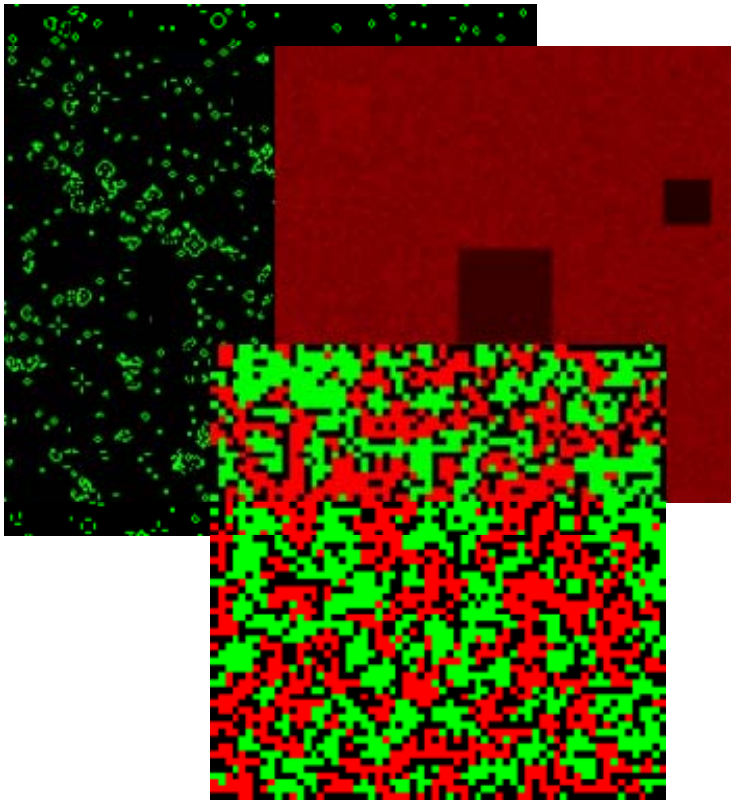
- Too many to list!
- Applications to GPUs have proliferated in past ~7 years
- You name *X*, you'll find "*X* on GPUs" in the literature

Non-Traditional GPU Applications

- Agent Based Simulations
- Transportation Simulations
- Network Simulation

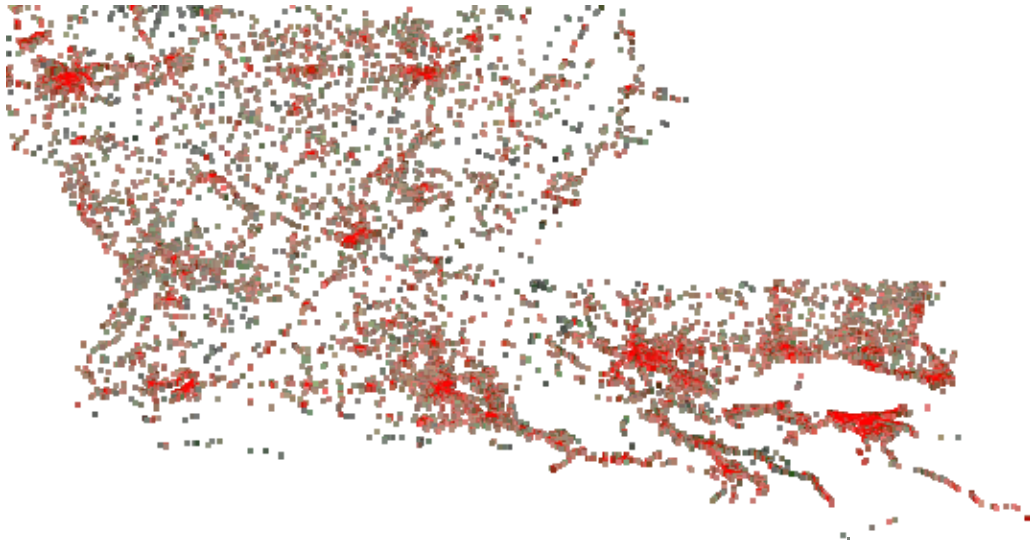
Agent Based Simulations

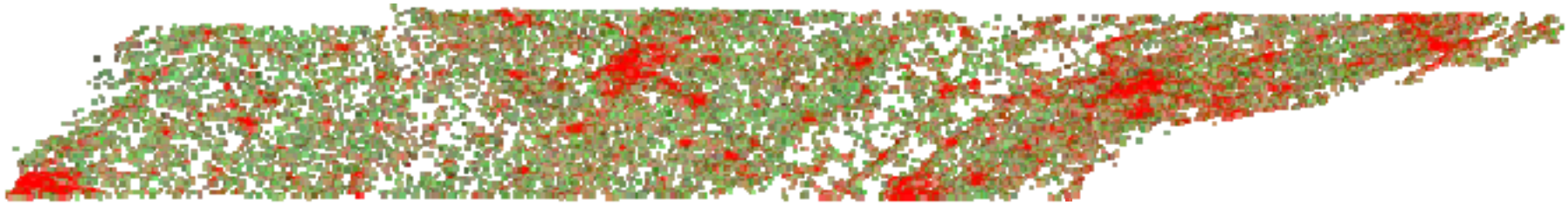
- Examples:
  - K. S. Perumalla and B. Aaby, "Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs," in Agent-Directed Simulation Symposium, 2008
  - R. D'Souza, M. Lysenko, and K. Rehmani, "SugarScape on Steroids: Simulating Over a Million Agents at Interactive Rates," in AGENT Conference on Complex Interaction and Social Emergence, 2007



Transportation Simulations

- Example:
  - K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal, "GPU-based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios," in Principles of Advanced and Distributed Simulation, 2009





Network Simulation

- Example:
  - Z. Xu and R. Bagrodia, "GPU-Accelerated Evaluation Platform for High Fidelity Network Modeling," in Principles of Advanced and Distributed Simulation, 2007

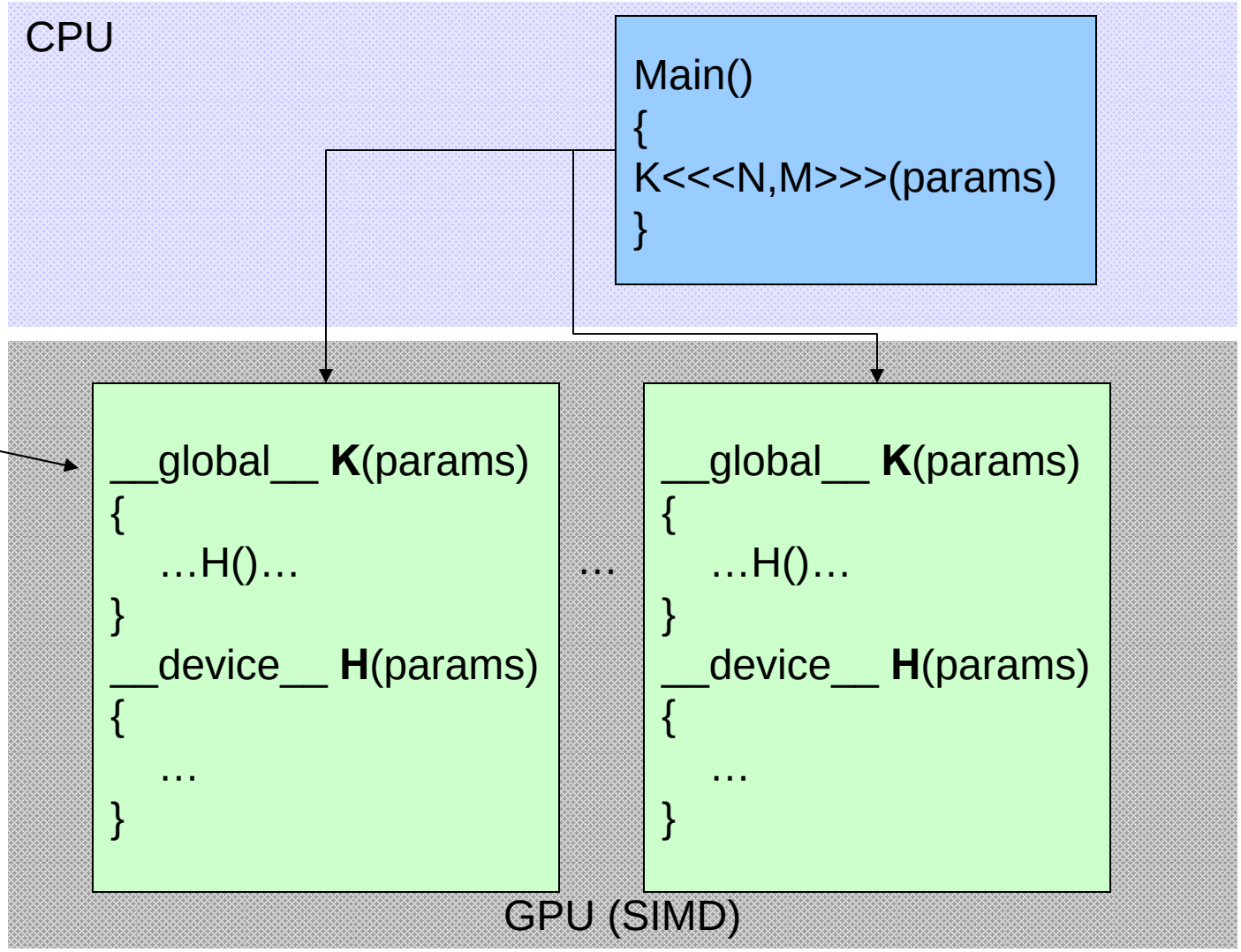
Development

- Debugging
- Testing
- Performance Tuning

Basic Concepts

- Execution Contexts and Kernel Functions
- Inter-Memory Data Transfers
- Launching GPU Threads
- Synchronization, Coordination, Termination

Execution Contexts and Kernel Functions

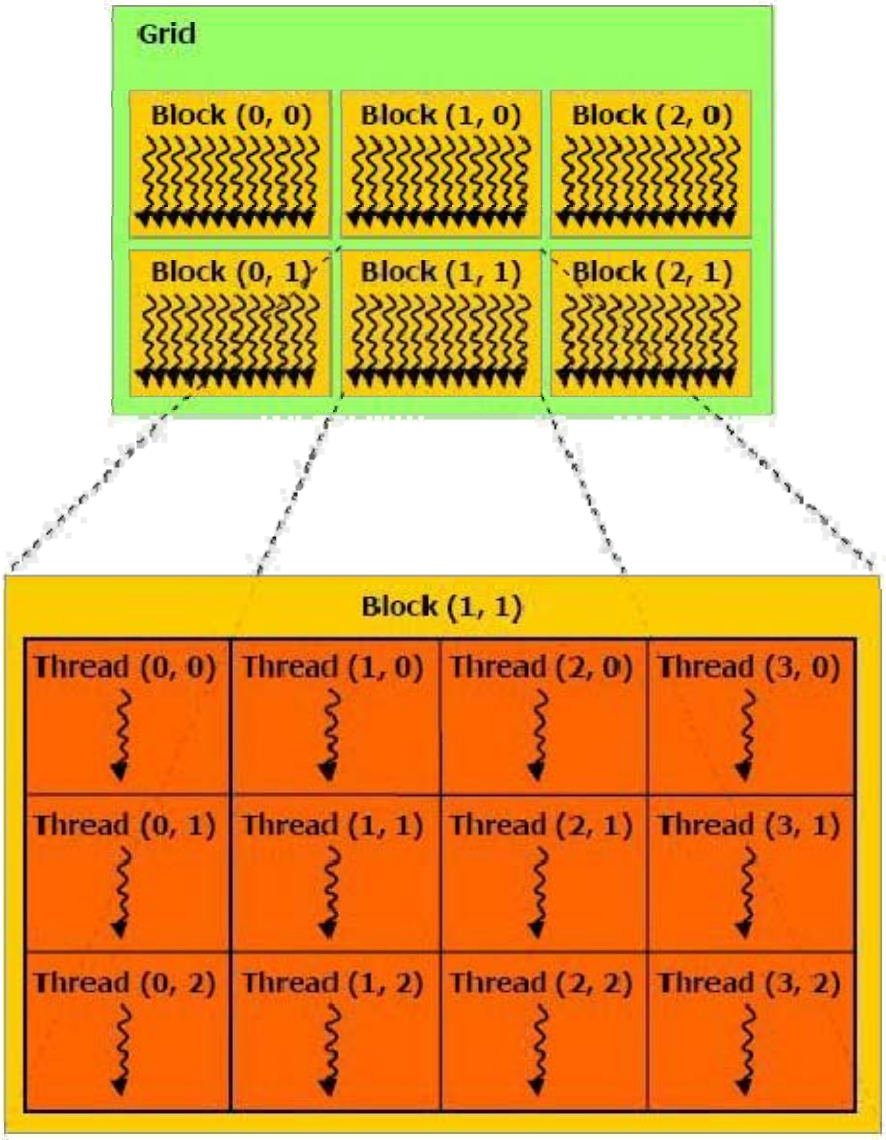


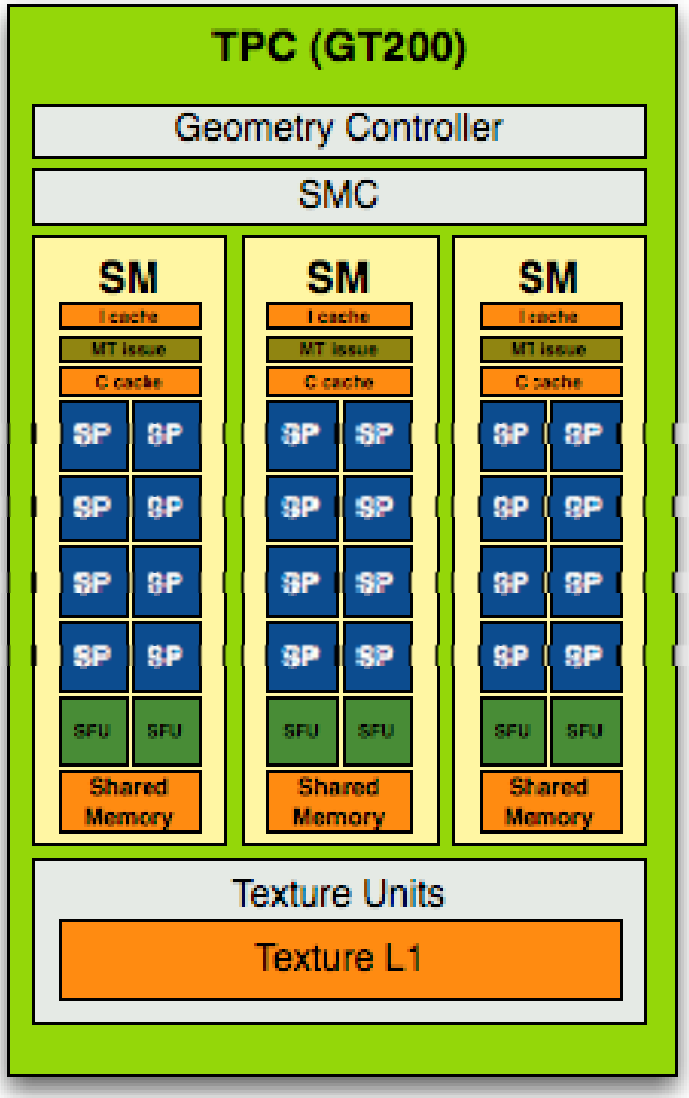
Each invocation of the kernel function (in this example, **K**()) starts a "thread" on each SIMD unit

Threads may be organized as "blocks"; each thread thus has a block identifier and thread identifier

- Kernel functions typically tagged with qualifier keywords by the user (e.g., __global__ or __kernel__)
- Kernel functions execute within the GPU context

Software vs. Hardware Views





Software vs. Hardware Views

GPU Languages and APIs

E.g., CUDA

- Blocks and Threads
- Memory Types
- Kernels and variable types and qualifiers

Contrast to CPU

E.g., C/C++

- Processes, threads
- Process-memory, shared memory
- Stacks, heaps, function frames

GPU Chips

E.g., GTX 200 & 300 series

- Thread Processing Clusters
- Streaming Multi-processors
- Streaming Processors
- Register files, DPUs, SFUs, Memory

Contrast to CPU

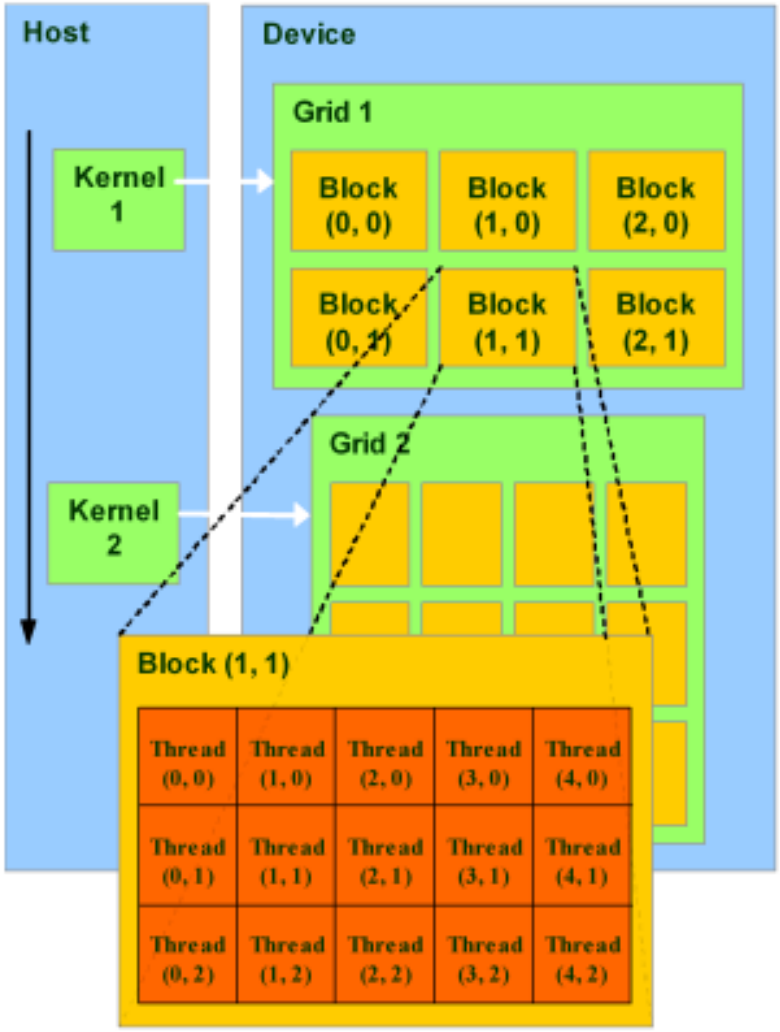
E.g., Intel & AMD CPUs

- Symmetric SM Multiprocessor sockets
- Mult-core processors
- Registers, ALUs, L1/2/3 Caches

Inter-Memory Data Transfers

- A fact of life in almost all current GPU systems
  - Many different notions and types of memory used
- Data transfer is one of the most taxing issues
  - More later, on this issue

Launching GPU Threads



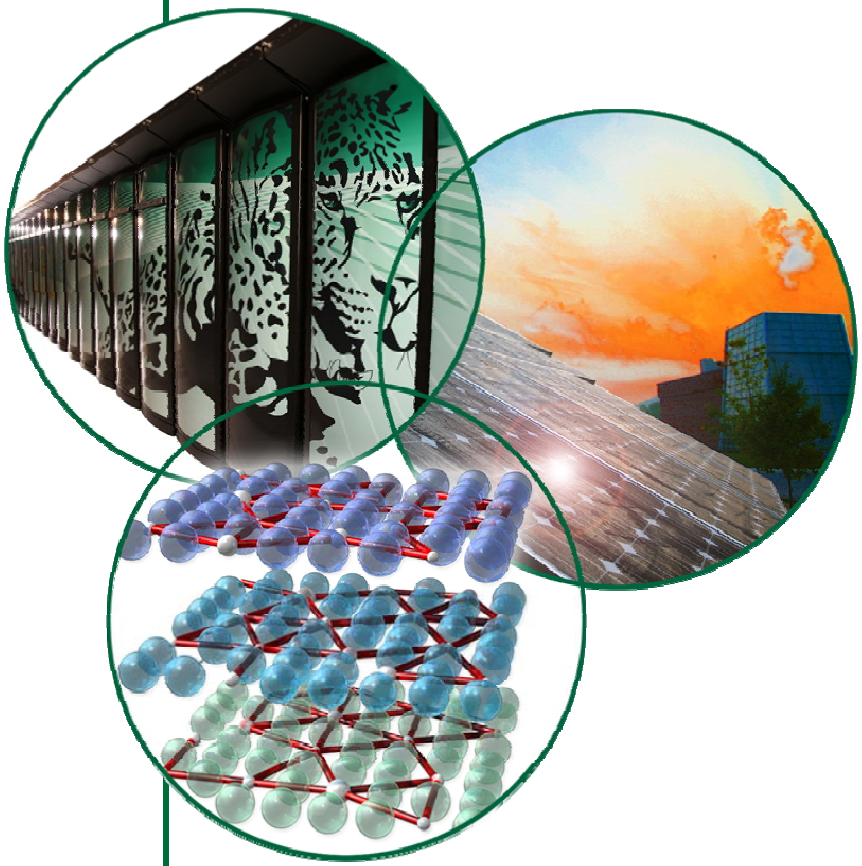
- Host initiates "launch" of many SIMD threads
- Threads get "scheduled" in batches on GPU hardware
- CUDA claims extremely efficient thread-launch implementation
  - Insignificant cost even for launching millions of CUDA (SIMD) threads at once

Synchronization, Coordination, Termination

- CPU-GPU synchronization
- Multi-GPU coordination
- Intra-GPU, inter-block synchronization
- Intra-block synchronization
- CPU-side termination

Part II

- Important Computational Considerations
- Time-stepped, Discrete-Event, and Hybrid Simulations



Computational Considerations

- Memory Hierarchy
- Scheduling
- Synchronization
- SIMD Constraints
- Numerical Effects
- Platform Limitations

Memory Hierarchy

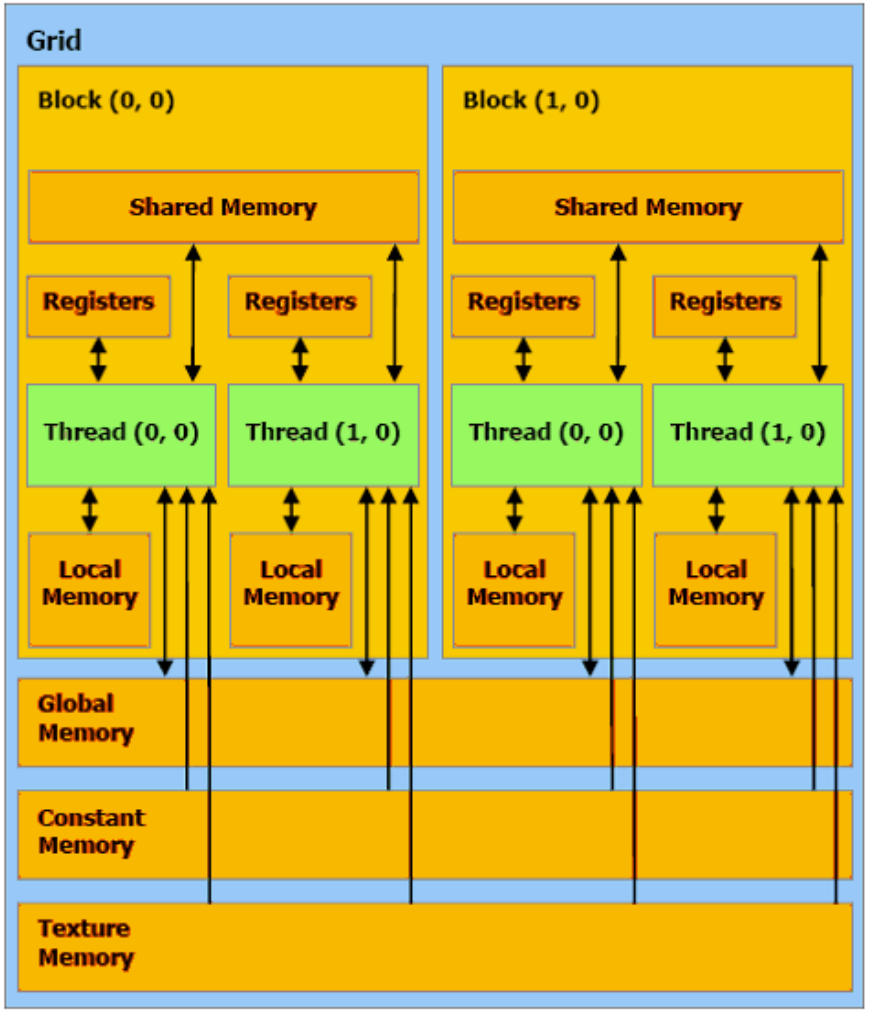
- CPU Memory vs. GPU Memory
- GPU Memory Types
- Bank Conflicts, Bandwidth

CPU Memory vs. GPU Memory

- Almost all GPU systems so far are co-processor-style architectures
  - A traditional CPU augmented by one or more GPUs
- In the fastest GPU systems, CPU memory is distinct from GPU memory
  - For least CPU-GPU synchronization, and
  - For the best VLSI layout of memory + processors on GPU
- For the next 3-4 years (in my opinion), main-memory *vs*. GPU-memory distinction is a necessary evil
  - Until future architectures change this, with “heterogeneous multi-cores”

GPU Memory Types

- GPU memory may come in several flavors
  - Registers
  - Local Memory
  - Shared Memory
  - Constant Memory
  - Global Memory
  - Texture Memory
- An important challenge is organizing the application to make most effective use of hierarchy

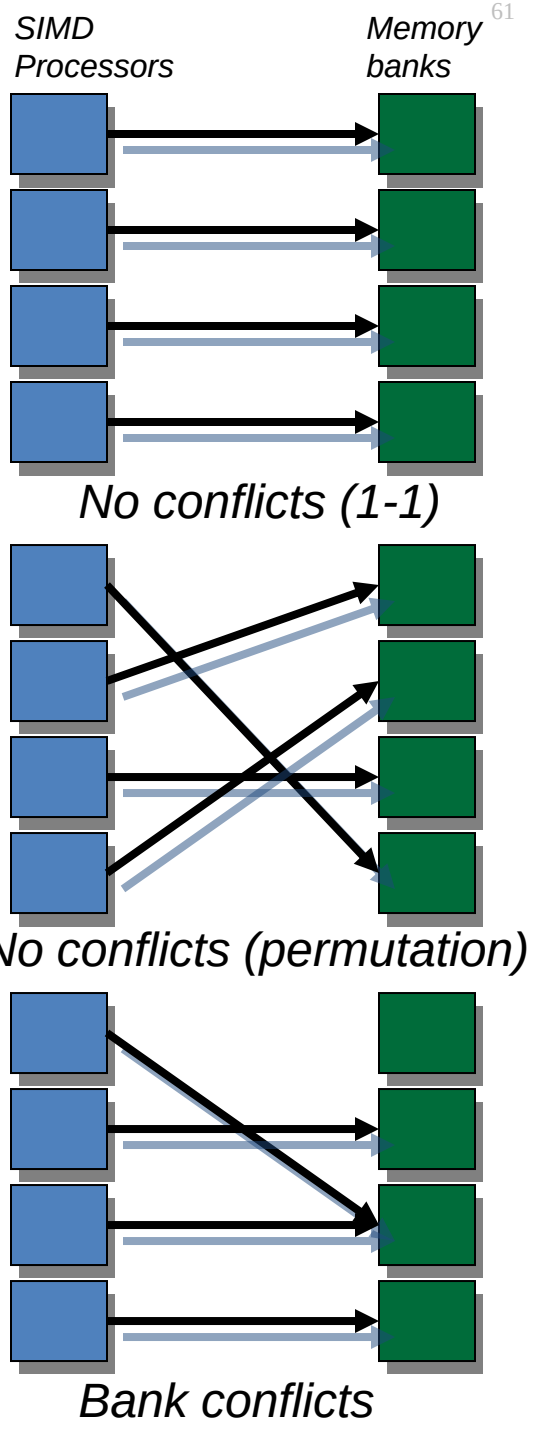


GPU Memory Types (NVIDIA)

Memory Type	Speed	Scope	Lifetime	Size
Registers	Fastest (4 cycles)	Thread	Kernel	
Shared Memory	Very fast (4 -? cycles)	Block	Thread	
Global Memory	100x slower (400- cycles)	Device	Process	
Local Memory	150x slower (600 cycles)	Block	Thread	
Texture Memory	Fast (10s of cycles)	Device	Process	
Constant Memory	Fairly fast (read-only)	Device	Process	

Bank Conflicts, Bandwidth

- "Bank conflicts" a direct implication of SIMD execution accessing multiple memory banks
  - Suppose an operation *A* is executed in parallel on multiple SIMD processors
  - Let the instance of *A* on processor *p* be denoted by *A[p]*, accessing a memory location *M[A[p]]* hosted on memory bank *B[A[p]]*
  - If all *B[A[p]]* are distinct from each other, then, no bank conflict occurs
  - If any two or more *B[A[p]]* are same, those memory accesses get serialized on that bank
- Performance may degrade by a factor of *W*
  - If worst conflict has W processors accessing the same bank
  - Because all other processors have to stall for *W* units of memory access time (compared to 1 unit without conflicts)

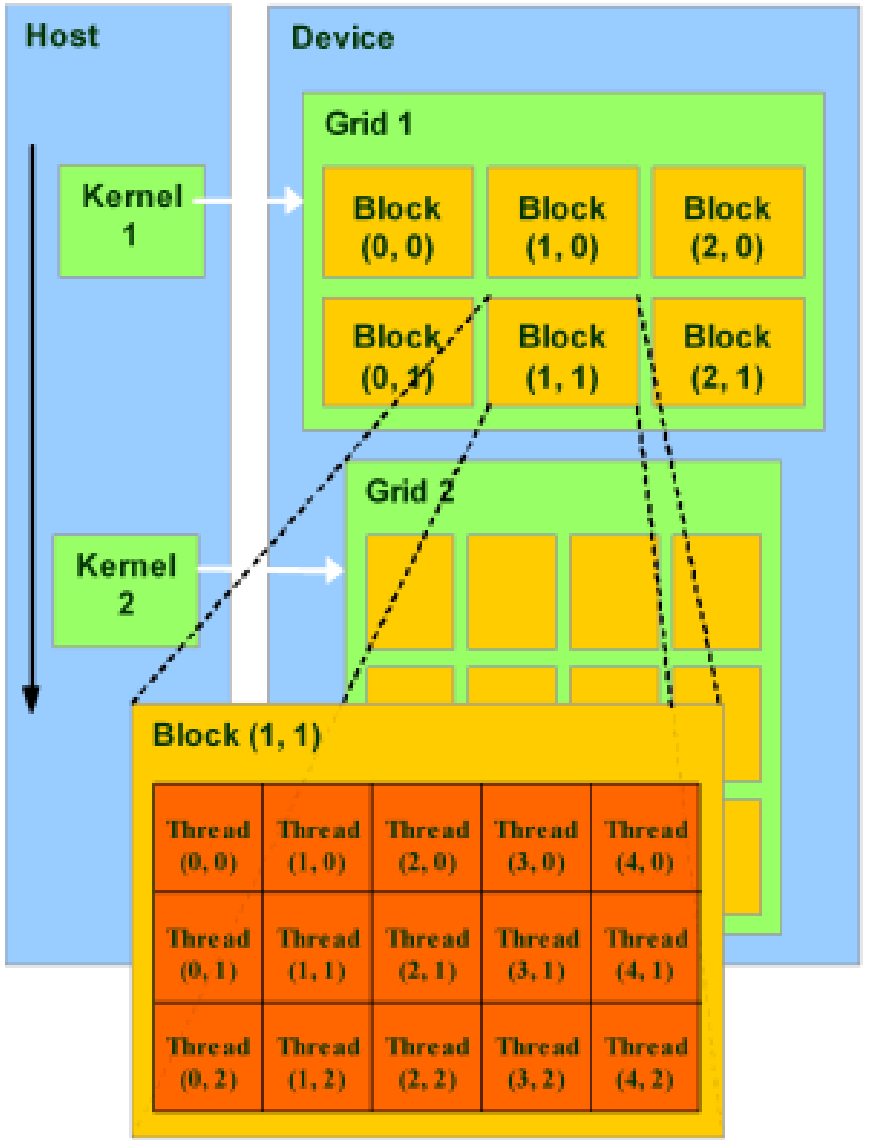


Scheduling

- Thread Launch Cost
- Thread-Count Effects

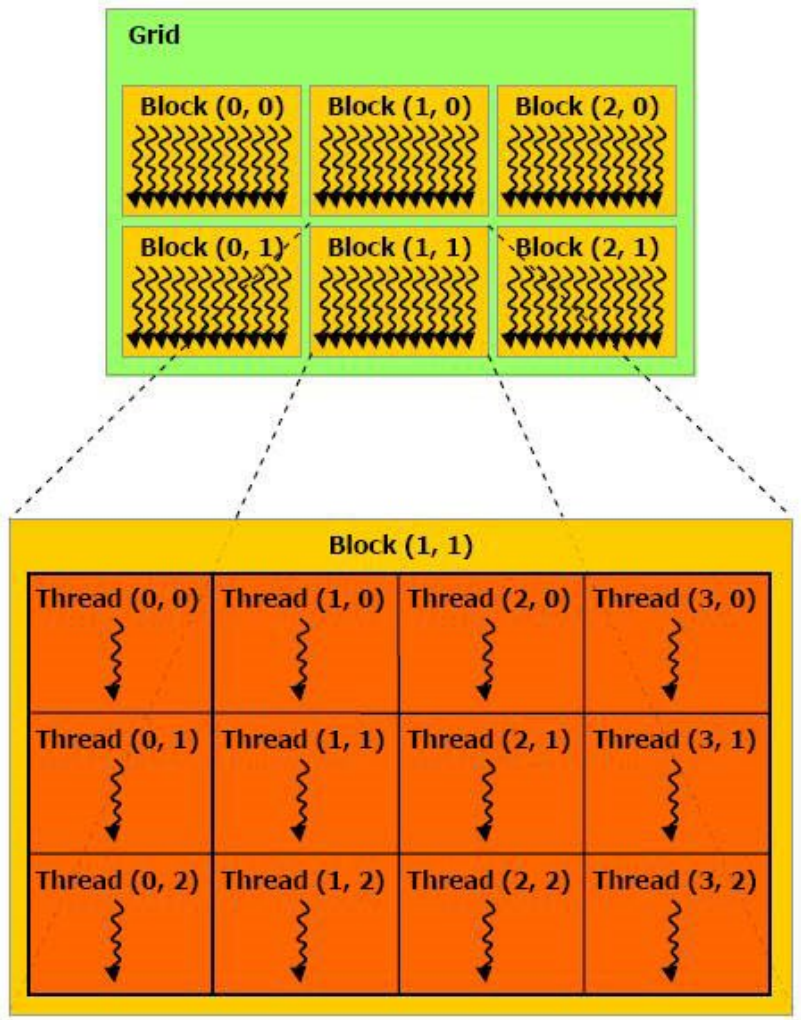
Thread Launch Cost

- Amortize thread launch cost
  - Although launch may be low-cost
- Perform more operations within one kernel invocation (thread)
  - Aggregate functionality that does not require global synchronization
  - Use local (block-level) synchronization if/as necessary



Thread-Count Effects

- Threads are launched in batches to the stream processors
  - Fragment Processors and/or Vertex Processors of yesteryear GPUs
  - Batches are called Warps in CUDA
- Each batch should contain sufficient threads to fill the number SIMD processing units on the GPU
  - High efficiency is achieved by asynchronous memory servicing, large Warp counts
  - Large number of blocks needed to overlap memory-fill latencies



Synchronization

- CPU-GPU Coordination
- Intra-GPU Thread Coordination

CPU-GPU Coordination

- Flush GPU pipelines
- Need to stall until GPU threads done
  - Otherwise, memory is in inconsistent state

Intra-GPU Thread Coordination

- Threads need to coordinate on GPU across steps
- CUDA only provides intra-block synchronization, but no inter-block synchronization
  - Often intra-block synchronization is useful in simulations, but difficult to implement efficiently

SIMD Constraints

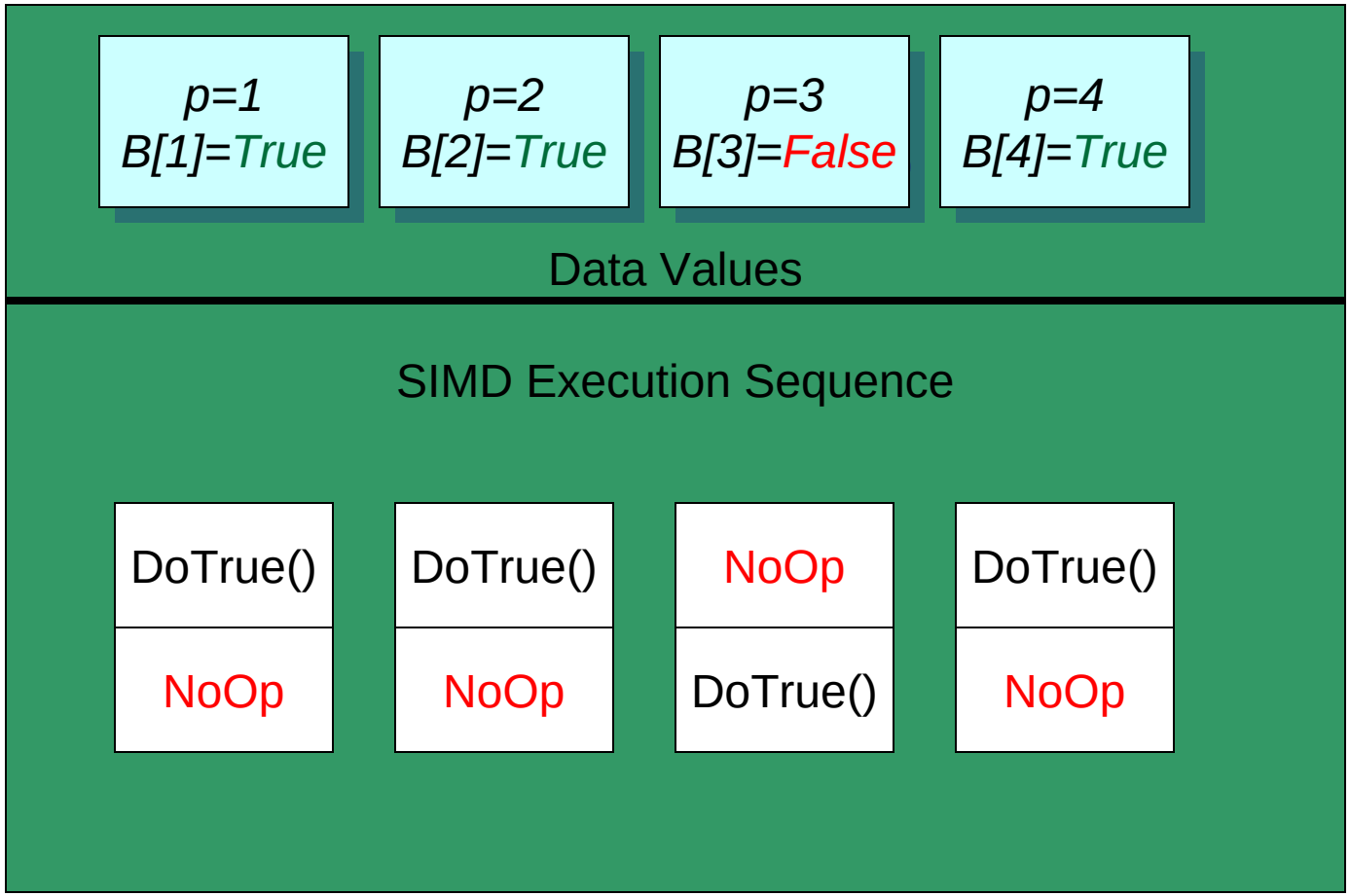
- Conditional Statements
- Looping
- Random Number Generation
- Bias and SIMD Conflict
- Modeling Challenge

Conditional Statements

- SIMD brings a natural challenge with conditional statements
- The way in which the true and false branches of a conditional statement are executed by all GPU processors
  - Since data is different across processors, some processors P(true) may evaluate the condition to be true and the others P(false) find it false
- SIMD needs all processors to execute same block of instructions
  - Hence all processors must execute the true branch first, during which only P(true) will execute the true branch, and P(false) will execute a no-op
  - Next, all P(false) execute the false branch, while P(true) execute a no-op
- If most of the time P(true) or P(false) are empty, then performance is unaffected
  - E.g., when all processors evaluate the same truth value
  - Total time equals time for the chosen branch
  - Otherwise, total time taken is the sum of times for both branches together!

Conditional Statements (continued)

```
p=processor ID
If( B[p] )
{
    DoTrue();
}
Else
{
    DoFalse();
}
```



- In general, best to minimize conditional statements in kernels
  - This can be done by invoking different kernels from the CPU itself, by carefully partitioning the data sets a priori

Looping

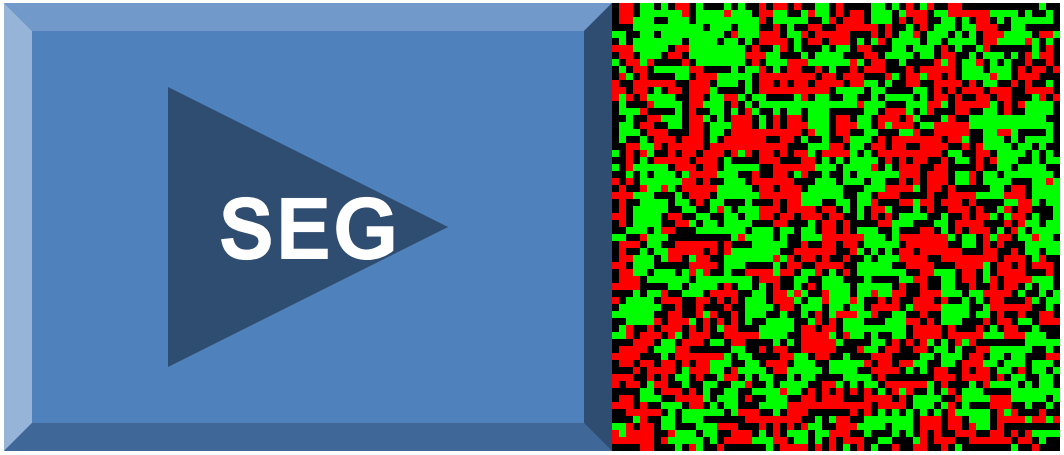
- Looping may be realized on CPU or GPU:
  - The loop in CPU calls a GPU kernel multiple times, once per CPU iteration
  - The CPU makes a single kernel call; a loop within the kernel performs the iterations on the GPU
- In the first (CPU-loop), kernel invocation cost is incurred for every iteration (thread launch cost)
  - **But all kernels are naturally synchronized after every iteration**
- In the second (GPU-loop), only one kernel invocation is involved
  - **Saves thread launch cost for every iteration**
  - **But kernels need synchronization operation after every iteration**
  - **Not always possible (e.g, *_syncthreads()* is block-specific in CUDA)**

Random Number Generation

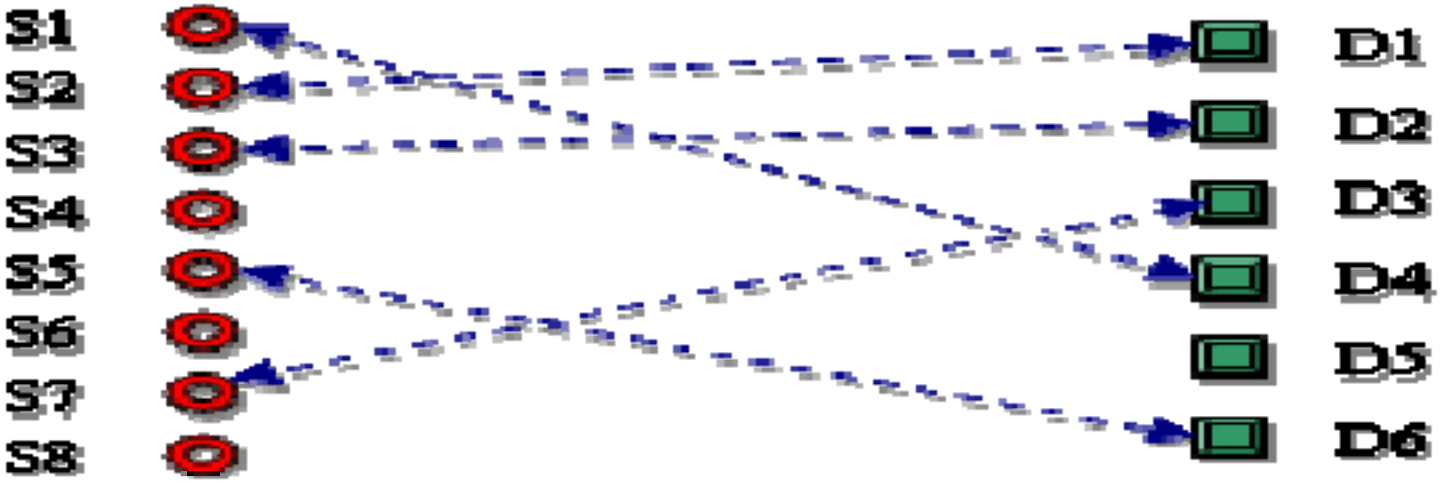
- Data parallelism implies one random number generator (RNG) per data/entity element
- Small memory sizes implies small working set for RNG
- Simple linear congruential generators may be sufficient for some (agent-based models)
- Mersienne Twister available in CUDA, but performance implications unclear on large agent models

Bias and SIMD Conflict

- Data parallelism presents some fundamental conflicts
  - Data parallelism vs. Model specification
  - E.g., Segregation model
- Certain ad-hoc schemes are possible, but can interfere with model needs
  - RUN DEMO



Bi-Partite Mapping Challenge (Data Parallel)



Appears in ABM for modeling

- Exclusion
- Information propagation

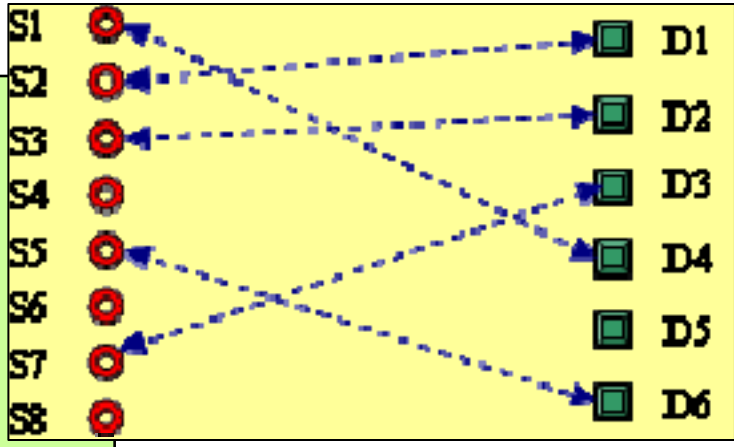
Examples

- Move
- Infect
- Spawn

Problem: Naïve approaches are inadequate

- Priority-based schemes result in artificial bias
- Semaphore-based schemes incur runtime overheads

Randomized Bi-Partite: Algorithm



Pass 1: For each agent A_{ij}
    If A_{ij} is a source
        Do nothing
    Else (A_{ij} is a destination)
        Within the neighborhood of vision v,
         Randomly select a source
             S (tentative)

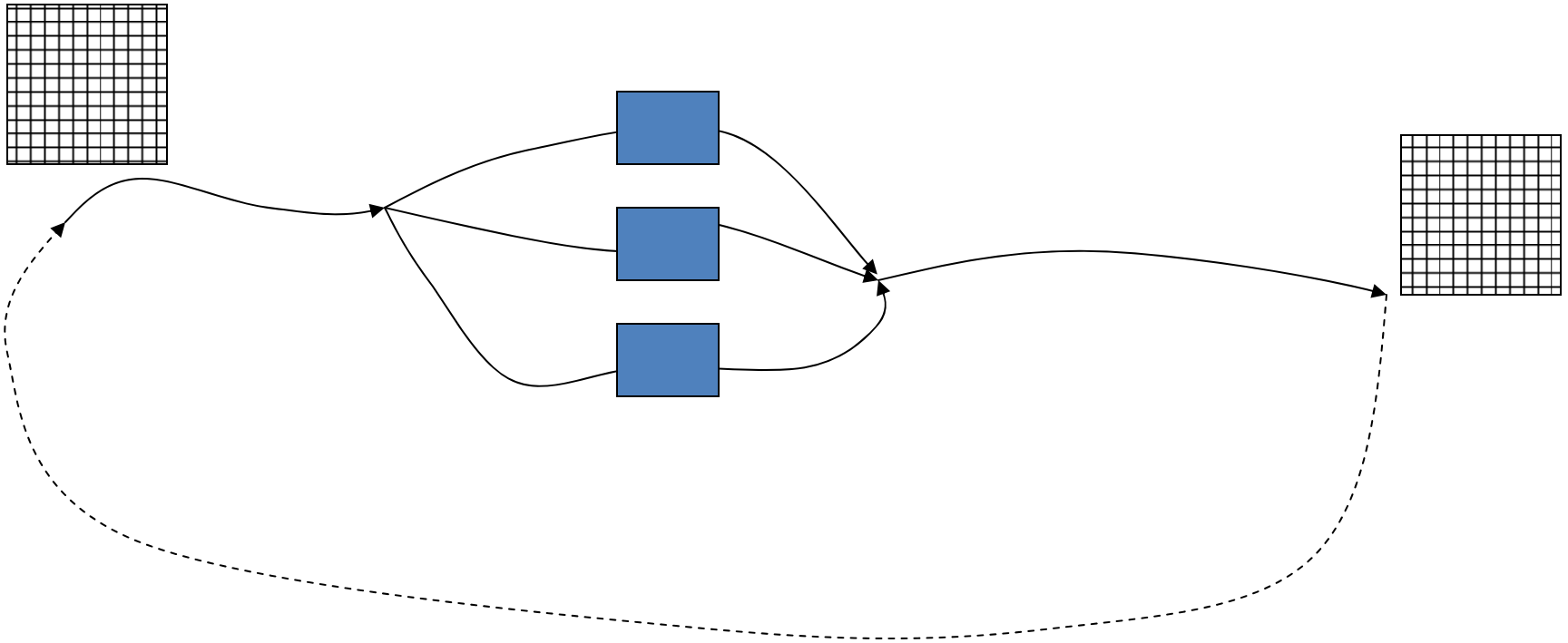
Pass 2: For each agent A_{ij}
    If A_{ij} is a source
        Within the neighborhood of vision v,
         Find the number of destination agents
          who have picked A_{ij} as their source
        If the number is exactly equal to 1
            Mark self as mapped to that
            Unique destination
    Else (A_{ij} is a destination)
        If this agent has a source selected
            S(tentative)
           Examine the neighborhood of S to
             verify that A_{ij} is the only
             destination that selected S
           If A_{ij} is unique in selection of S
             Mark self as mapped to that S

Modeling Challenge

- Major challenge in moving from current CPU models to GPU
- Often reformulation of model is required
  - E.g., field-based model for transportation
- For maximum efficiency, irregular topologies must be converted to rectangular structures "somehow"

Major Hurdle: Computation Paradigms

- CPU = Random memory access
- GPGPU = Streams-based paradigm
- GPU = Highly data-parallel (SIMD) paradigm



Fallout for DES

- In general, unclear how to map LPs and events to GPGPU framework
- Can't update LP in isolation
- Can't execute "single event" individually
- Stream LP states?
- Stream events? Only simultaneous ones? Lookahead-based window? Multiple events/LP?

Numerical Effects

- Numerical Precision
- Accuracy - Visual vs. Analytical

Numerical Precision

- In GPU evolution, numerical precision has been low, but has been improving
- Newest NVIDIA processors have begun supporting double precision floating point
  - One double precision unit (DPU) per SM
  - 64-bit arithmetic, from previous limitation of 32-bit arithmetic
- Some limitations still exist, to be carefully considered
- Examples
  - Double precision units are fewer than single precision units
  - In kernel invocations, silent conversion may happen (double precision actual arguments from CPU to single precision formal arguments on GPU)

Accuracy - Visual vs. Analytical

- GPU hardware is historically geared towards visual consumption
  - Floating point effects are not same as found on CPUs
- Imprecision is less pronounced on latest generation of hardware
  - But lower precision data types usable in GPU languages
- Special function units (SFU) have different algorithms
  - **Different precision than CPU-equivalents for transcendental functions**

For precision effects comparisons often need to account for imprecision, esp. in our simulations. For example,

```
if( x == 1 ) {}
```

should be realized as

```
const float eps=1e-5;
if( 1-eps < x && x < 1+eps ) {}
```

Platform Limitations

- Recursion
- Thread Stack Sizes
- Thread Pause/Resume

Recursion

- Recursion occurs when a function (kernel) F invokes itself in a chain of function (kernel) invocations
  - Direct recursion example: F() { ... if() { ... F(); ... } ... }
  - Indirect recursion example: F() { ... G(); ...} G() { ... F(); ... }
- Few GPU systems support recursion in GPU threads (kernels)
- Most in fact "flatten" or "inline" the function call graph
  - E.g., via static analysis of function call chain
  - Perform memory allocation to fit the chain, all in the compiler
  - Hence, currently, only call chains of DAGs (directed acyclic graphs) or trees

Thread Stack Sizes

- Thread stack size are statically determined
- Maximum stack size may be severely limited by the memory available on GPU
  - Typically to a few dozen kilo bytes per thread
- For best efficiency and safety, may be best to avoid very long function (kernel) call-chains
  - Unlike in current day CPU-based software

Thread Pause/Resume

- GPU threads are fully in-lined function call graphs
  - In almost all current GPU systems
  - Little support for pause/resume semantics of CPU-based threads
- Major distinction to bear in mind while moving from CPU-based simulations to GPU-based simulation systems
  - Process-oriented simulations difficult to realize on GPUs

Time Stepped & Discrete Event Simulation

Time Stepped Simulation

- Typical Usage Template
- Time Advance on CPU
- Time Advance on GPU

Discrete Event Simulation

- Data parallel execution (SIMD) implies traditional event loop does not make sense
- Cannot implement discrete event time leaps with conventional algorithm(s)
- Need to revisit the application program interface (API), and refine it for GPUs

Solutions

- Algorithms
- Operation
- Example and Performance Study

TS and DES: Example Application

$$\frac{\partial Q}{\partial t} = \alpha_x \frac{\partial^2 Q}{\partial x^2} + \alpha_y \frac{\partial^2 Q}{\partial y^2} + \beta$$

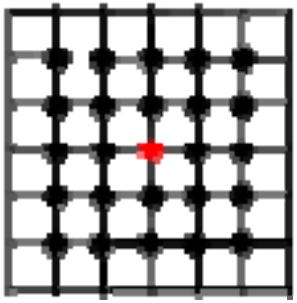
- Simulation of Diffusion Processes
  - E.g., heat, dye, mood, disease, ...
- 2-D scenario
  - Spatial grid along x, y axes
- Study performance of time-stepped and discrete-event execution on CPU & GPGPU

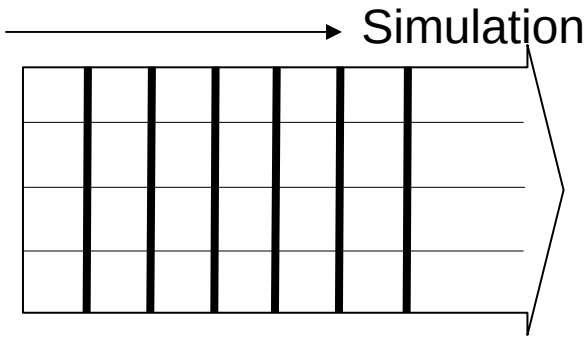
Time-stepped Approach

- Advance simulation time in fixed increments, **Δt**
- Update entire grid state every **Δt**

$$\frac{\partial Q}{\partial t} = \alpha_x \frac{\partial^2 Q}{\partial x^2} + \alpha_y \frac{\partial^2 Q}{\partial y^2} + \beta$$

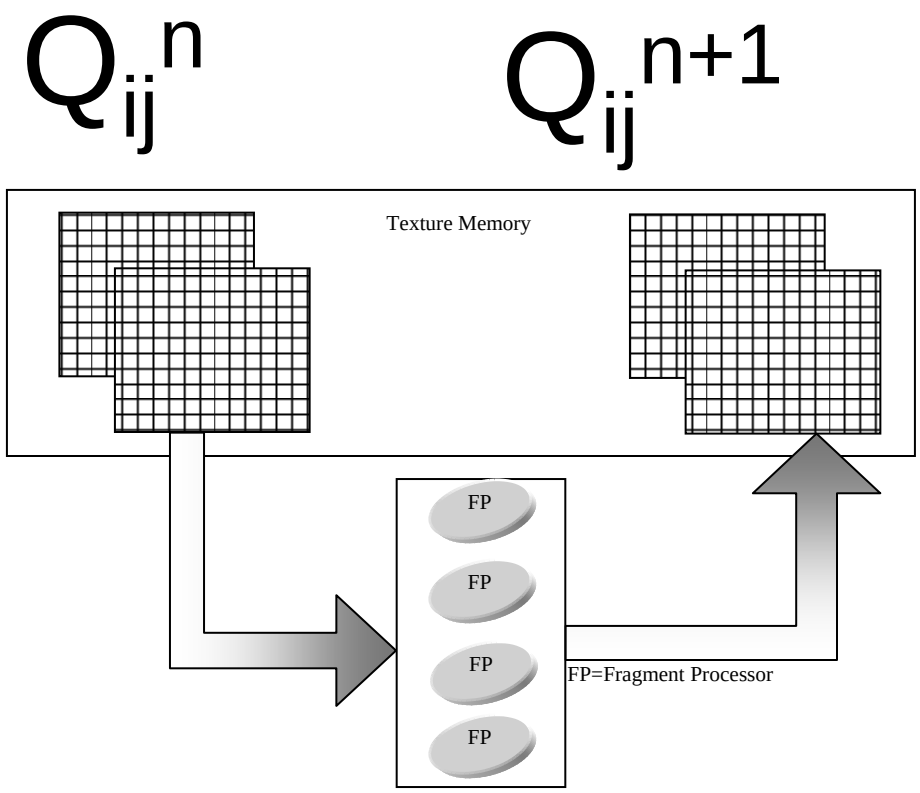
- Maps very easily to stream processing of GPGPUs





$$\frac{q_{i,j}^{n+1} - q_{i,j}^{n}}{\Delta t} = \alpha_x \frac{q_{i,j-1}^{n} - 2q_{i,j}^{n} + q_{i,j+1}^{n}}{\Delta x^2} + \alpha_y \frac{q_{i-1,j}^{n} - 2q_{i,j}^{n} + q_{i+1,j}^{n}}{\Delta y^2} + \beta$$

Mapping Time Stepped Update to GPGPU



Q[i][j] = f(Q[i][j],
Q[i-1][j], Q[i+1][j],
Q[i][j+1], Q[i][j-1])

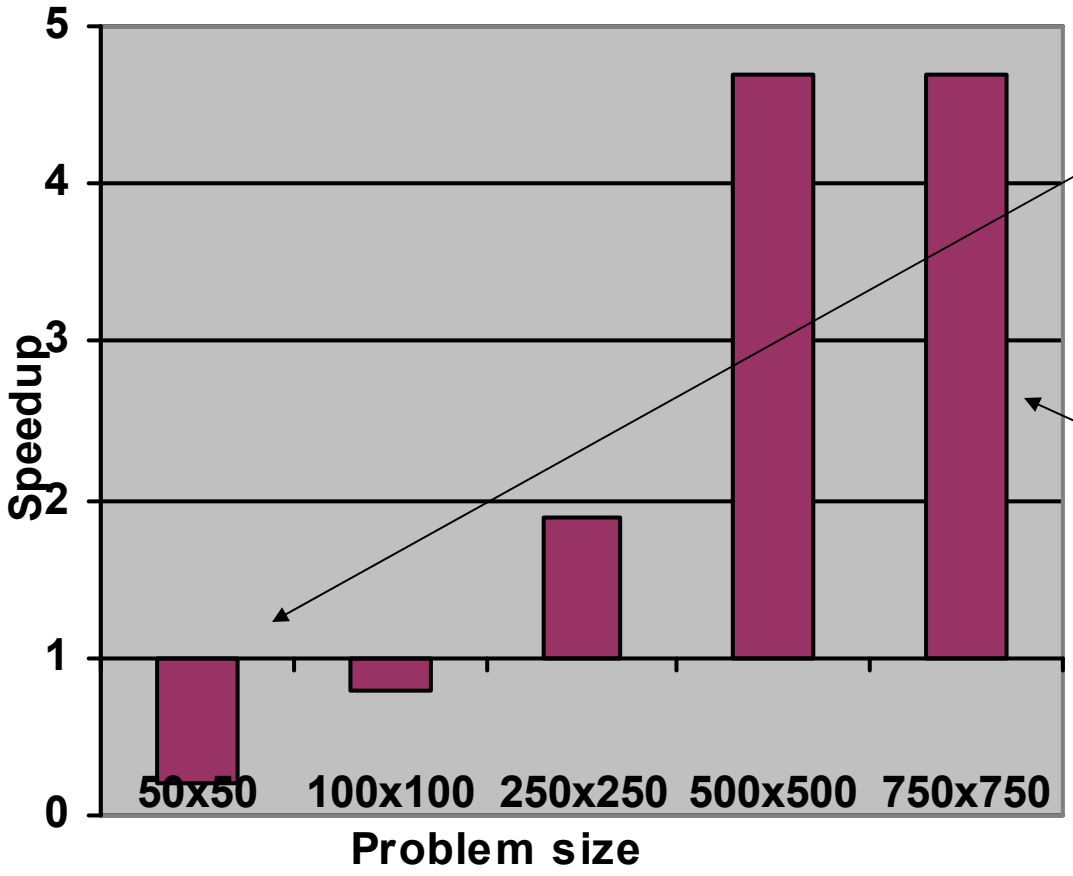
Note: No read/write hazards

- Most existing GPGPU simulations are time-stepped!
- Shown to be much faster on GPGPU than on CPU

Experiment Platforms

- CPU: Centrino 2.1GHz, 2GB
- GPU: NVIDIA GeForce 6800 Go, 256MB,
  16 Fragment processors
- CPU: Microsoft VC++ v7
- GPU: Brook stream compiler,
  DirectX 9 runtime

Time-stepped Performance on GPGPU



Large caches of CPU help on small grid sizes

Streaming, 2-D caching and asynchronous memory operations of GPGPU help on large grid sizes

- Performance relative to time-stepped code on CPU
- 2x implies TS on GPGPU is twice as fast as on CPU

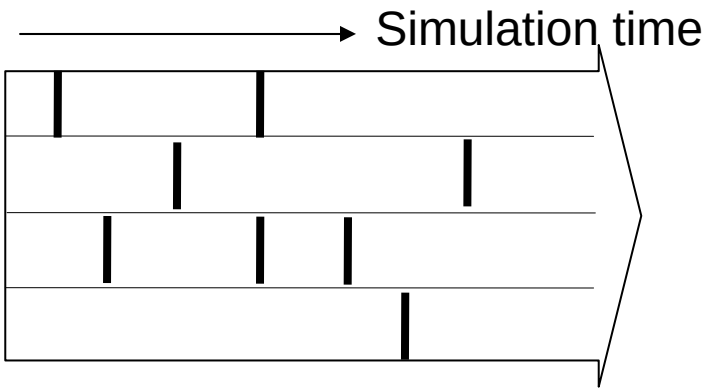
But is it a fair comparison?

- Time-stepped simulation on CPU is not the fastest method
  - Other methods exist
  - E.g., discrete event formulation
- Time-stepped may favor GPGPU
  - Asynchronous memory operations, etc.
- Let us compare with DES on CPU…

Discrete Event Approach on CPU

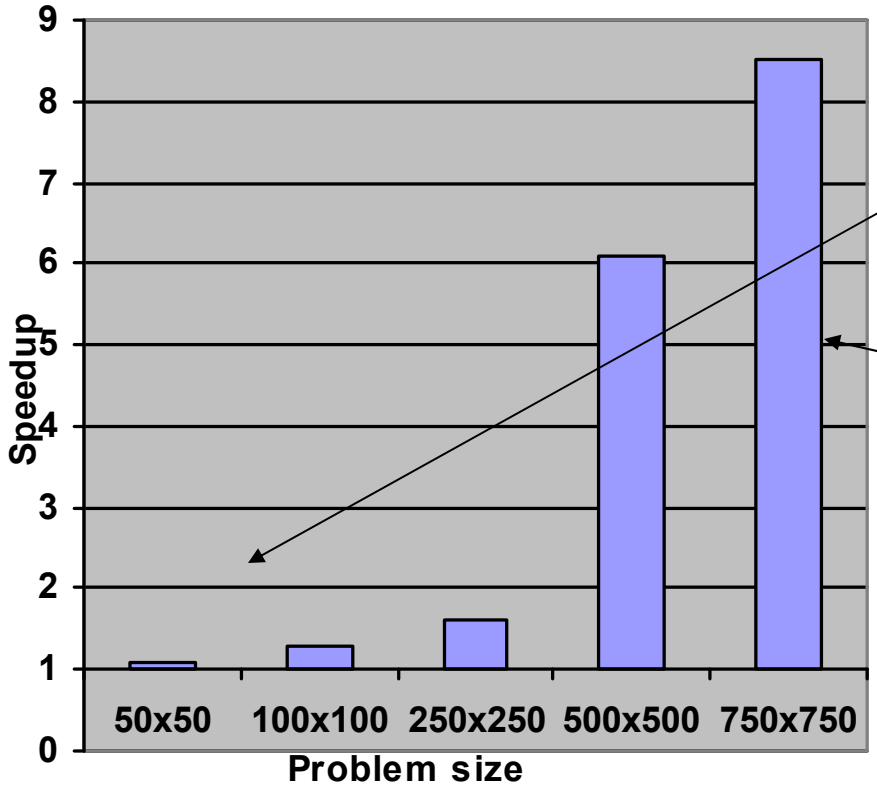
- Advance simulation time in variable increments
- Each grid element is advanced independently
  - Solve for **Δ**t, for a given resolution of Q (state space)
- Maps to event list based simulation

$$\frac{\partial Q}{\partial t} = \alpha_x \frac{\partial^2 Q}{\partial x^2} + \alpha_y \frac{\partial^2 Q}{\partial y^2} + \beta$$



$$\frac{q_{i,j}^{n+1} - q_{i,j}^{n}}{\Delta t} = \alpha_x \frac{q_{i,j-1}^{n} - 2q_{i,j}^{n} + q_{i,j+1}^{n}}{\Delta x^2} + \alpha_y \frac{q_{i-1,j}^{n} - 2q_{i,j}^{n} + q_{i+1,j}^{n}}{\Delta y^2} + \beta$$

Discrete-Event Performance on CPU



Little gain over TS

- Tight coupling across grid

High gain over TS

- Needless computation eliminated on larger grids

- Performance relative to time-stepped code on CPU
- Clearly DES is much faster than TS

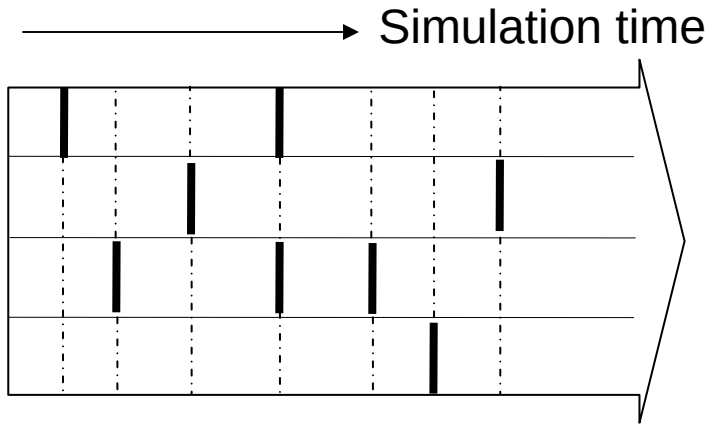
How can we apply similar approach to GPGPU?

- DES-style event list infeasible on GPGPU
  - Data parallel execution
- But global reductions are very fast
  - Fast logarithmic reduction algorithms
- Can reap the benefits of larger time advances
  - Without event lists, but with global advances

Hybrid Approach

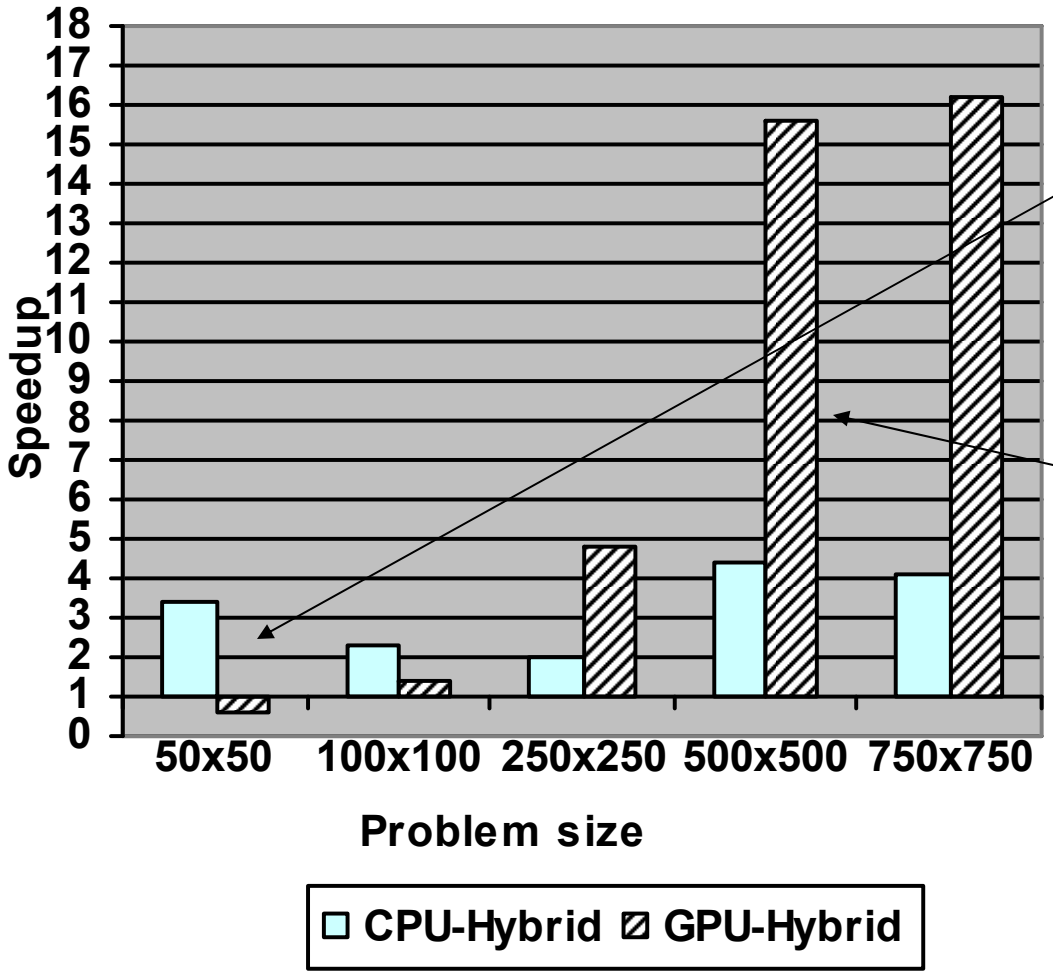
- Compute upper bound on **Δ**t for each element
  - Solve for **Δ**t, for a given resolution of Q (state space)
- Advance time by minimum **Δ**t, update *all* elements
- Maps to GPGPUs very well!

$$\frac{\partial Q}{\partial t} = \alpha_x \frac{\partial^2 Q}{\partial x^2} + \alpha_y \frac{\partial^2 Q}{\partial y^2} + \beta$$



$$\frac{q_{i,j}^{n+1} - q_{i,j}^{n}}{\Delta t} = \alpha_x \frac{q_{i,j-1}^{n} - 2q_{i,j}^{n} + q_{i,j+1}^{n}}{\Delta x^2} + \alpha_y \frac{q_{i-1,j}^{n} - 2q_{i,j}^{n} + q_{i+1,j}^{n}}{\Delta y^2} + \beta$$

Hybrid Performance on GPGPU & CPU



CPU much faster due to very large cache

- Small grid fits in cache!

High gain on larger grids

- Faster time advances enabled by hybrid execution

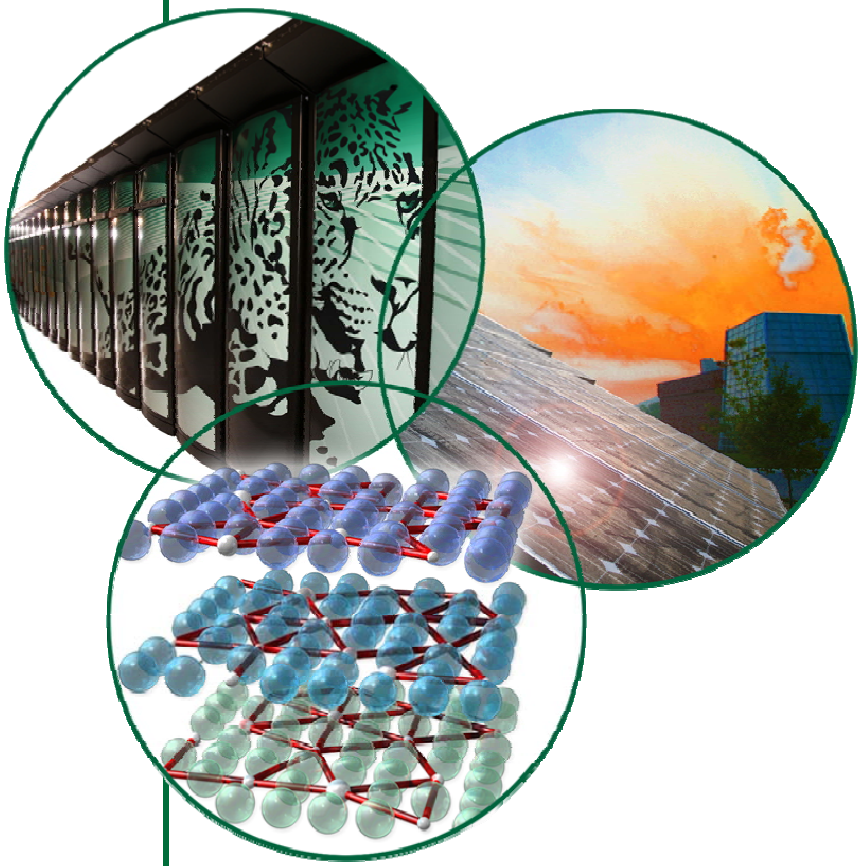
- Performance relative to time-stepped code on CPU

Performance Study Result

- Performance gain from DES-style execution can be reaped on GPGPU as well
  - Using proper adaptation of DES to hybrid
- GPGPU can give several fold improvement over CPU performance on plain TS as well as DES (hybrid)
  - GPU-Hybrid is 17× relative to TS-CPU!

Part III

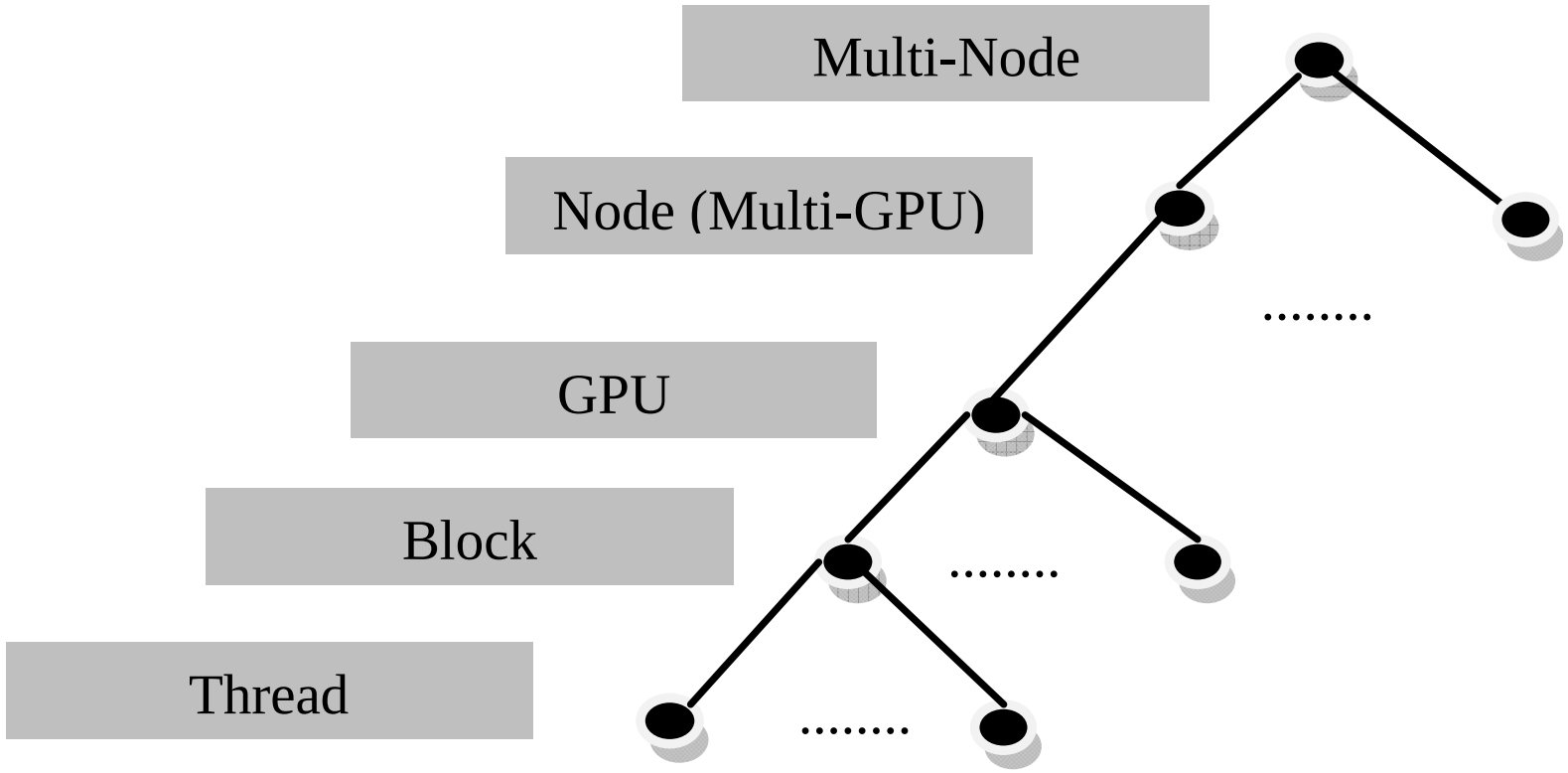
- Networked GPUs
- Other Types of GPU Usage in Simulations
- Future Developments and Outlook



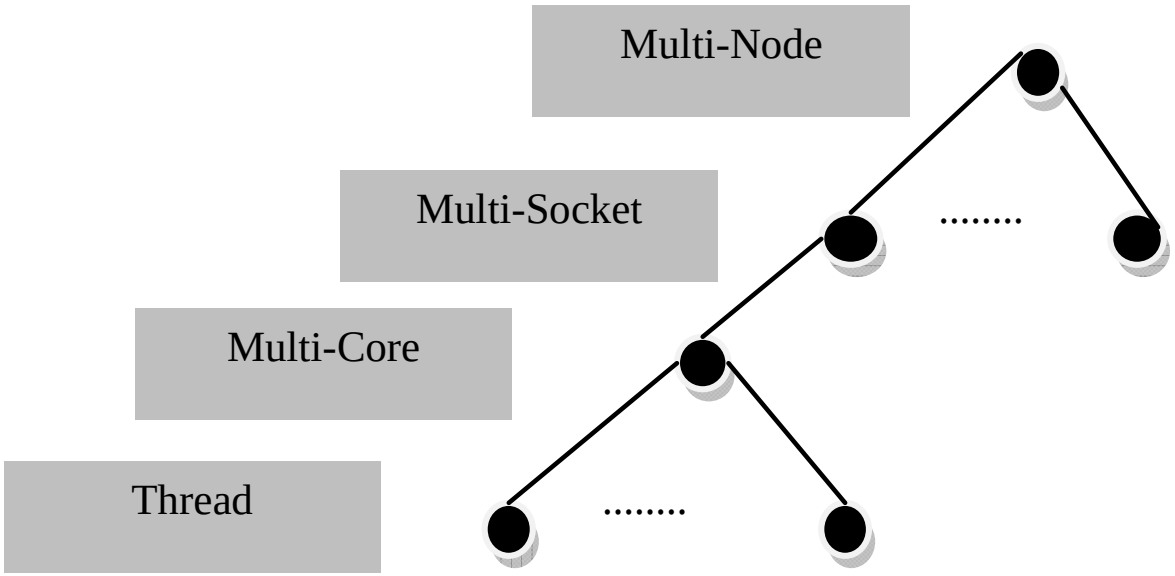
Networked GPUs

- Hardware System
- CUDA+MPI
- Latency Challenge
- B+2R Algorithm
- Performance

Hierarchical GPU System Hardware



Analogous Networked Multi-core Hardware System



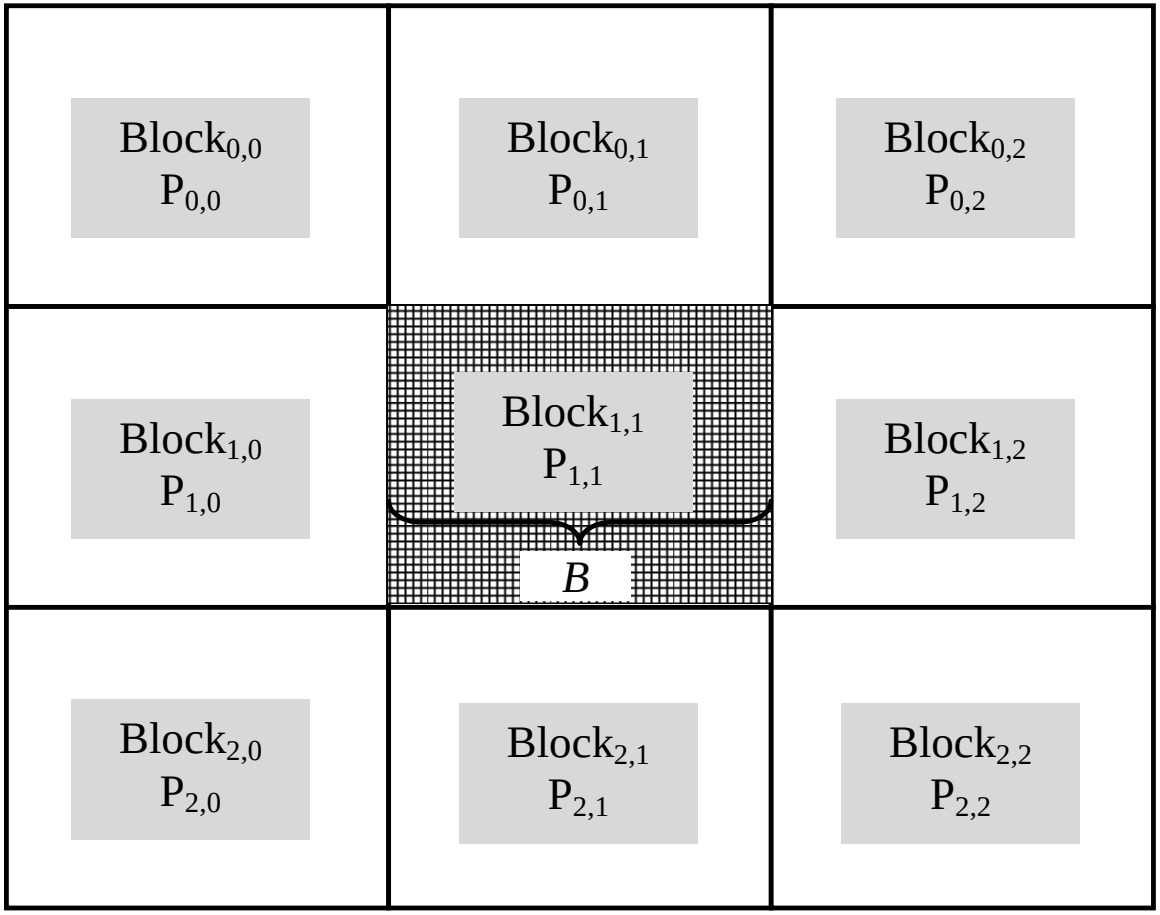
CUDA+MPI

- An economical cluster solution
  - Affordable GPUs, each providing one-node CUDA
  - MPI on giga-bit Ethernet provides inter-node communication
- Memory speed-constrained system
  - Inter-memory transfers can dominate runtime
  - Runtime overhead can be severe
- Need a way to tie CUDA and MPI
  - Algorithmic solution needed to overcome latency challenge

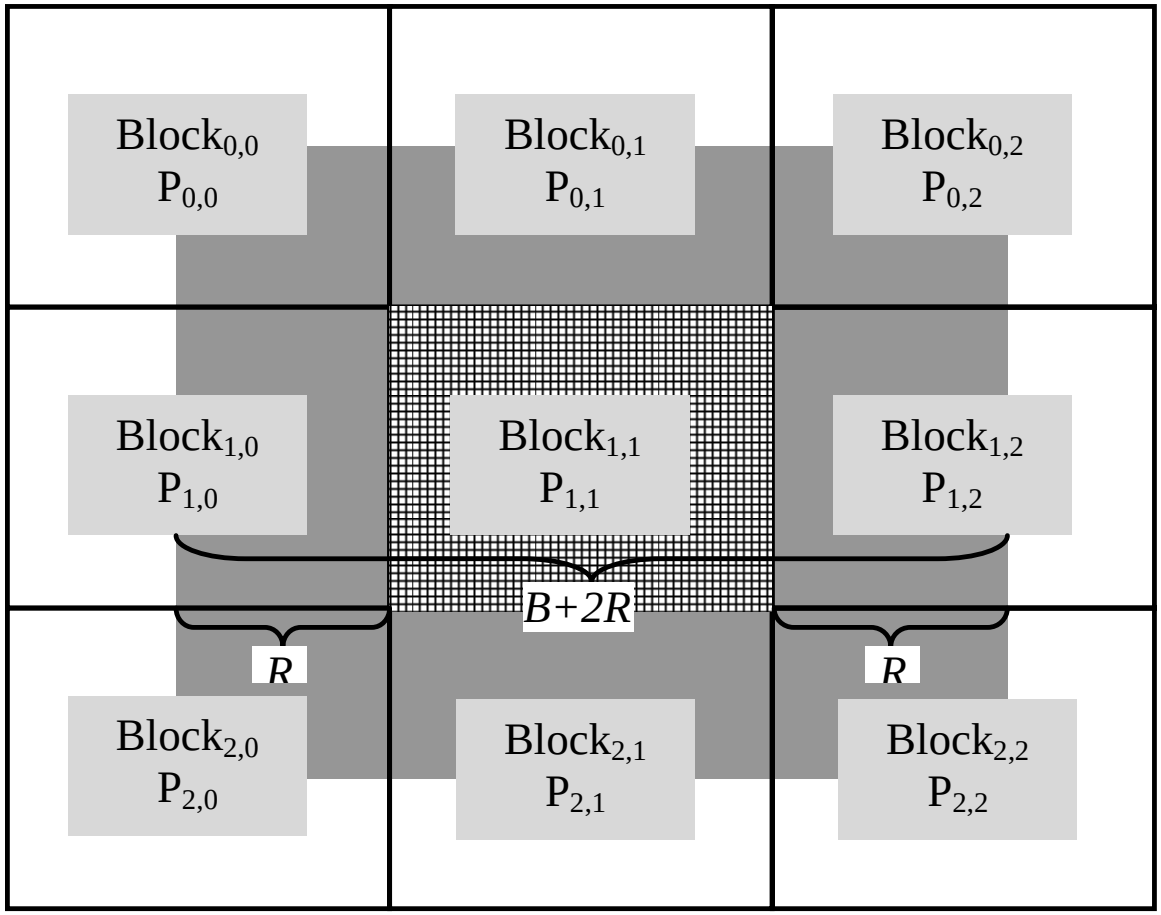
Latency Challenge

- High latency between GPU and CPU memories
  - CUDA inter-memory data transfer primitives
- Very high latency across CPU memories
  - MPI communication for data transfers
- Naïve method gives very poor computation to communication ratio
  - Slow-downs instead of speedups across distributed GPUs

Example: Conventional Method



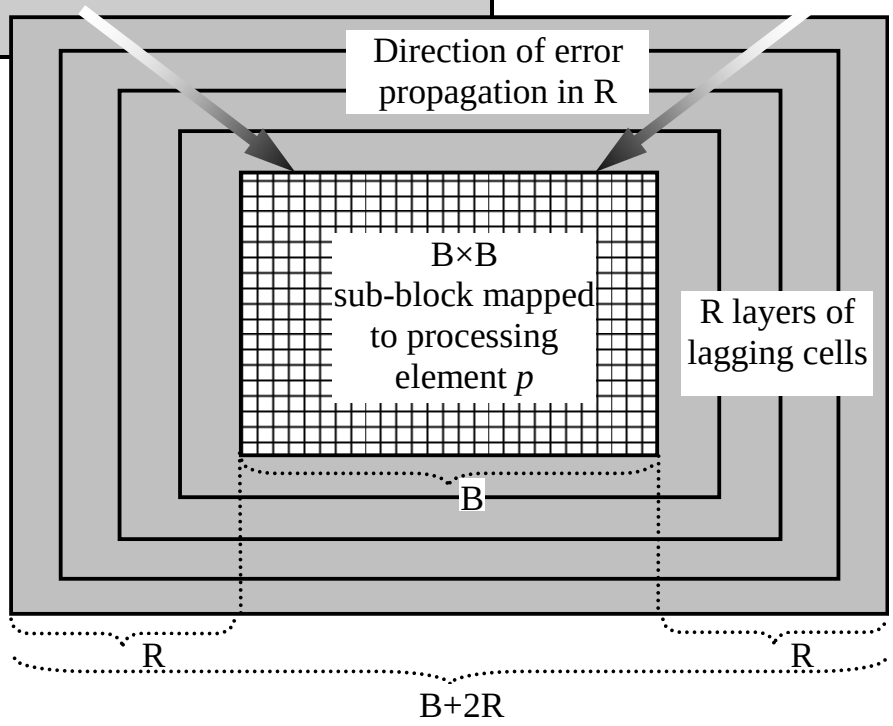
Our B+2R Method



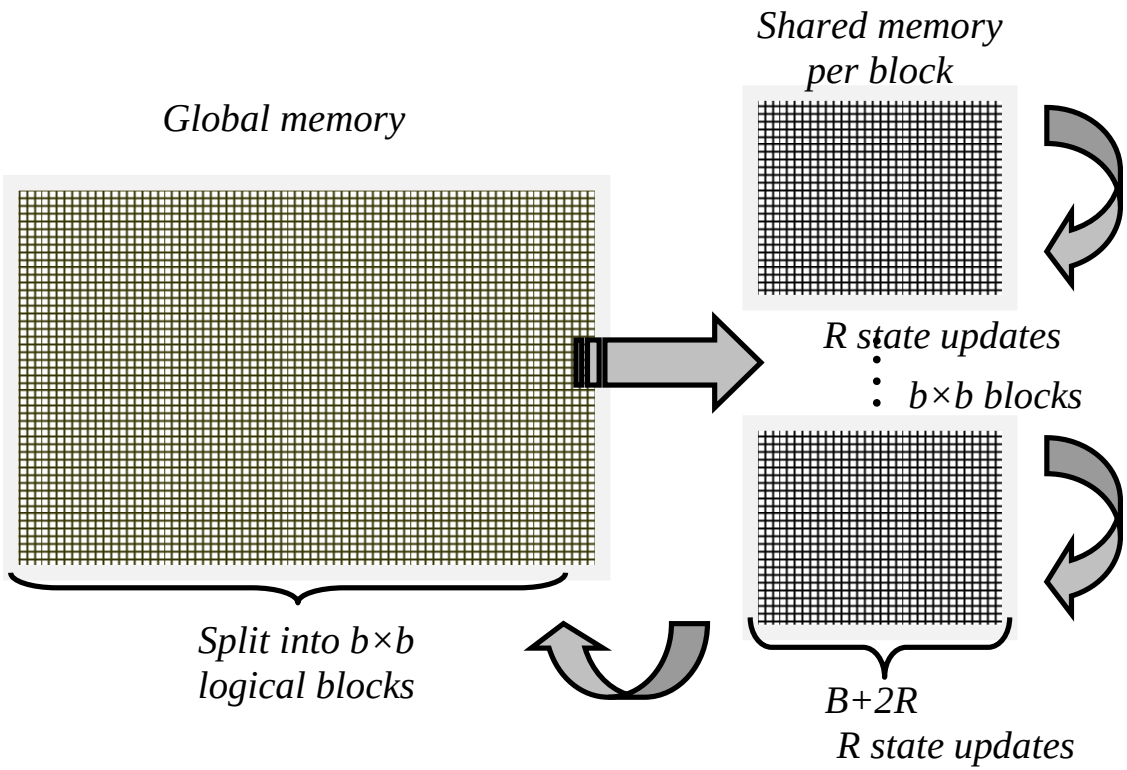
B+2R Algorithm

Let T_e be total number of iterations in the simulation

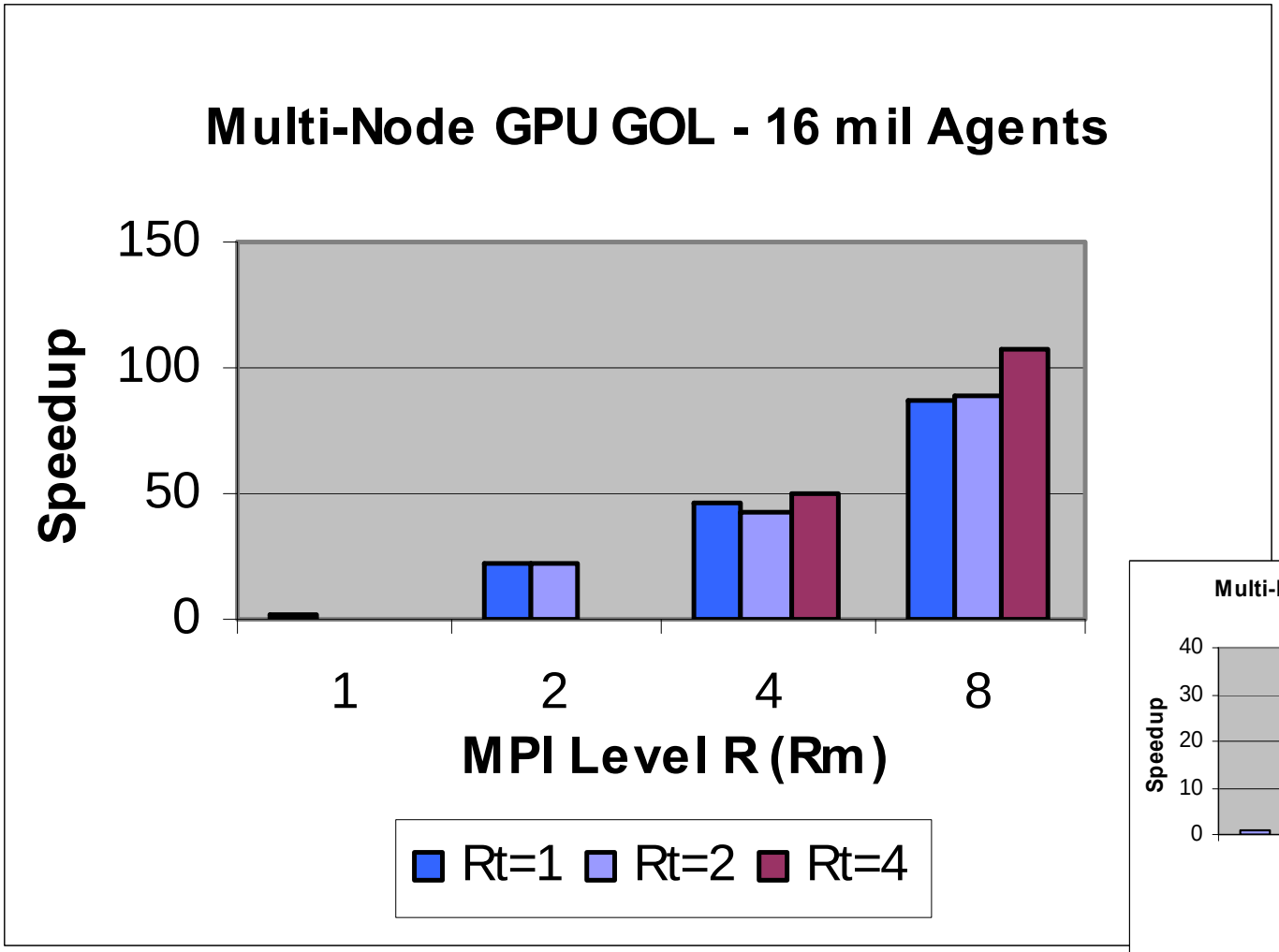
1 For all blocks $Block_{ij}$ in the given agent grid G
1.1 Let (t_{li}, t_{lj}) be the top left index of $Block_{ij}$
1.2 Let (b_{ri}, b_{rj}) be the bottom right index of $Block_{ij}$
1.3 For t=0 to T_e/R
1.4 For r=R-1 down to 0
1.5 Update(t_{li}-r, t_{lj}-r, b_{ri}+r, b_{rj}+r)
1.6 Communicate(t_{li}, t_{lj}, b_{ri}, b_{rj}, r)
1.7 Barrier()

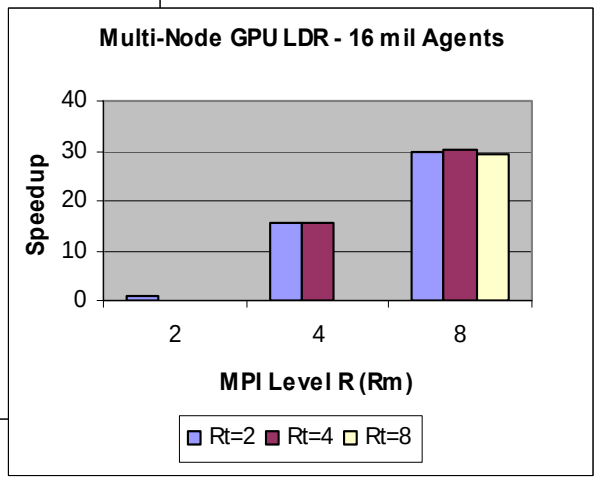


B+2R Implementation within CUDA



Performance





Over 100× speedup with MPI+CUDA

Speedup relative to naïve method with no latency-hiding

Other Types of GPU Usage in Simulations

- LOS Computation and Collision Detection
- Numerical Integration
- Linear Algebra

LOS Computation and Collision Detection

- Expensive computation such as line-of-sight (LOS) determination, and collision detection delegated to GPU
- Highly data-parallel computation well-suited for SIMD architecture of GPUs
- Examples
  - M. Verdesca, J. Munro, M. Hoffman, M. Bauer, and D. Manocha, "Using Graphics Processor Units to Accelerate OneSAF: A Case Study in Technology Transition," in *Interservice/Industry Training, Simulation and Education Conference (IITSEC)*, 2005
  - Gress A., Guthe M., Klein R., "GPU-based Collision Detection for Deformable Parameterized Surfaces," in *Computer Graphics Forum,* 2006

Numerical Integration

- Small memory foot-print, small time-step integrators are best suited for GPU platform

Example:

- J. Gao, E. Ford, and J. Peters, "Parallel Integration of Planetary Systems on GPUs," in *Proceedings of the 46th Annual Southeast Regional Conference on XX*, 2008

- Computation of forces in N-body problems may be viewed as direct numerical integration, performed on the GPUs
- Integration can be off-loaded to the GPU as co-processor.
- Dead-reckoning of entities (by integration schemes) in semi-automated forces, for example, can be off-loaded to the GPUs (no citation yet!)

Linear Algebra

- Matrix operations (multiplication or inversion) occur commonly inside a simulation, as part of simulation state updates.
- Examples
  - Matrix operations within a state update of an entity (within a multi-scale, multi-resolution simulation method)
  - Matrix operations across the entire simulation state (entire domain of a simulation using implicit methods)
  - These can be delegated to GPU as co-processors.

- The speed of lower precision may be exploited.
- Lower-precision arithmetic may be sufficient in some applications.
- Single-precision or mixed-precision linear algebra is another key motivation for using GPUs for linear algebra

Future Developments and Outlook

- OpenCL
- Nexus, CUDA-C++, MSVC
- Fermi/GTX300
- Heterogeneous Cores
- GPU-based Supercomputing
- Packaged and Customized Solutions

OpenCL

- Open Computing Language
- Device-independent programming (ideally!)
- Apple-led effort
- Gaining industry support
- We may expect NVIDIA, Microsoft, IBM, Intel, AMD, and others to support it

Nexus, CUDA-C++, MSVC

- New, “developer-friendly” environment from NVIDIA
- Integration with Microsoft Visual Studio
- Moving from C to C++ (CUDA already had some C++)

Fermi/GTX300

- Very recent offerings in the market
- Among the most powerful commodity, off-the-shelf GPU-based systems

Heterogeneous Cores

- Multi-cores all on die, but cores differ in functionality and capabilities
- Many types of cores into one system
- This is probably the medium- to long-term trend
- Little distinction between current processor and co-processor
- Customizable and/or packaged multi-cores

GPU-based Supercomputing

- Roadrunner-trend may continue
  - May not necessarily be based on IBM Cell processor
- Already NVIDIA (Fermi)-based high-performance configurations being installed
  - One at ORNL/Georgia Tech led by Jeffrey Vetter

Packaged and Customized Solutions

- E.g., Mobile platforms, financial markets, data mining
- E.g., Solutions marketed by Mercury Systems

Thank you!

Questions?

Slides will be made available online at www.ornl.gov/~2ip