Scalable Parallel Programming with CUDA on Manycore GPUs

John Nickolls

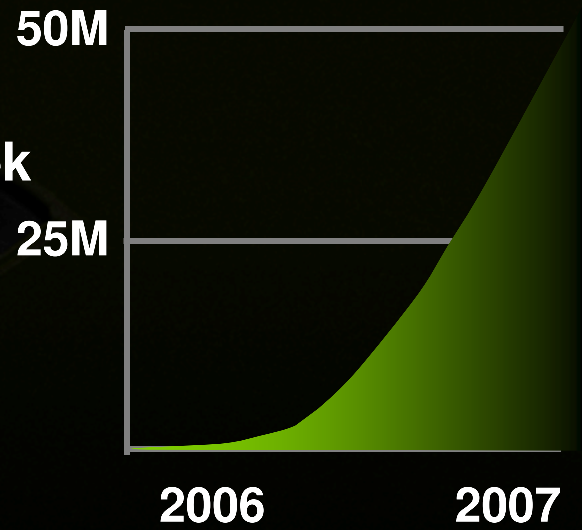
Stanford EE 380 Computer Systems Colloquium, Feb. 27, 2008

Outline

- Transition to scalable parallel computing
- CUDA applications
- CUDA programming model
- SAXPY example
- Sparse matrix vector product
- Parallel sum reduction
- N-body physics
- Tesla GPU Architecture
- Summary

The Transition to Parallel Computing

- Is well along … in unified graphics and computing processors
- The GPU is a scalable parallel computing platform
  - Thousands of parallel threads
  - Scales to hundreds of parallel processor cores
  - Ubiquitous – in laptops, desktops, workstations, servers
- CUDA parallel programming model introduced in 2007
  - Write C code for one thread
  - Instantiate parallel thread blocks
  - Tens of thousands of CUDA developers
- NVIDIA ships 1M CUDA-capable GPUs a week
  - Over 50 M CUDA-capable GPUs shipped
- Unique opportunity to innovate and develop widely-deployed parallel applications



Tesla GPU architecture

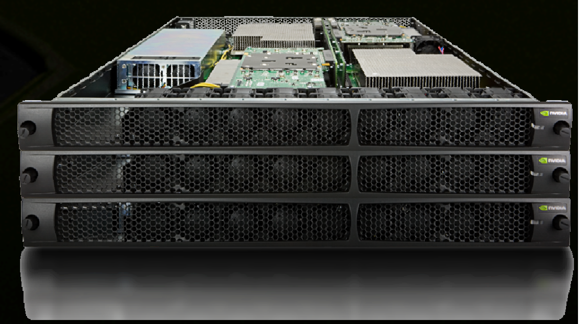
- Unifies graphics and computing
- Scalable parallel computing platform
- In laptops, desktops, workstations, servers

GeForce 8800

- 8-series GPUs deliver 50 to 200 GFLOPS on compiled parallel C applications
- GPU parallel performance pulled by the insatiable demands of PC game market

Tesla D870

- GPU parallelism doubling every 12-18 months
- Programming model scales transparently

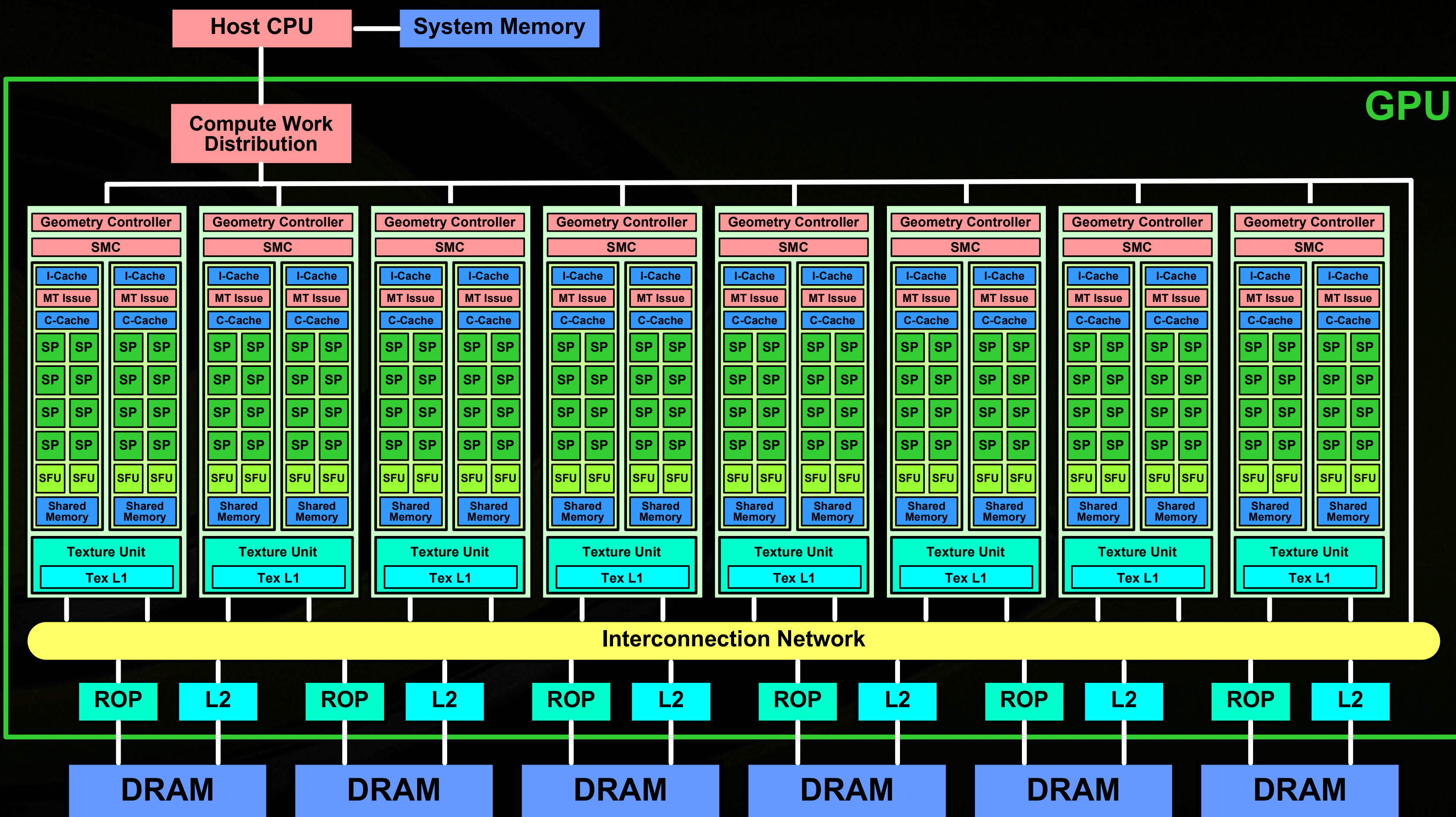


- Programmable in C with CUDA tools
- Multithreaded model uses data parallelism, task parallelism, and thread parallelism

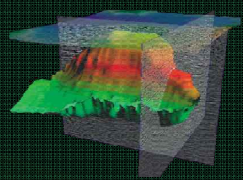
Tesla S870

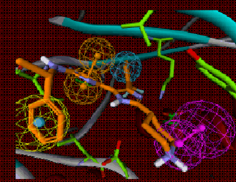
Tesla GPU Computing Architecture

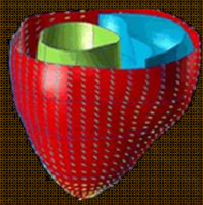
- Scalable processing and memory, massively multithreaded
- GeForce 8800: 128 SP processor cores at 1.5 GHz, 12K threads

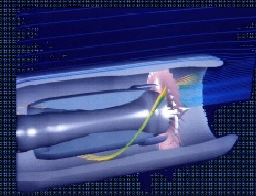


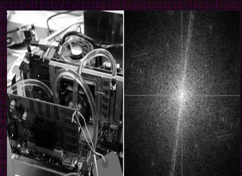
GPU Computing Application Areas

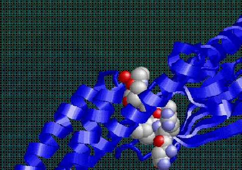

Computational Geoscience

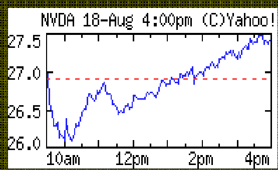

Computational Chemistry


Computational Medicine


Computational Modeling


Computational Engineering


Computational Biology



Computational Finance

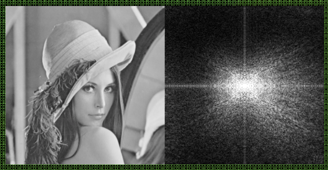
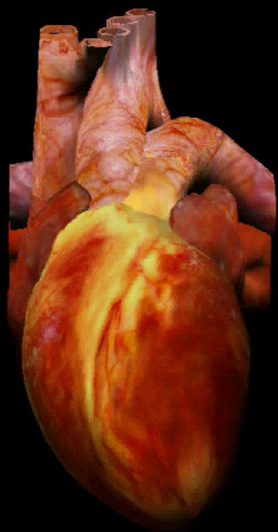
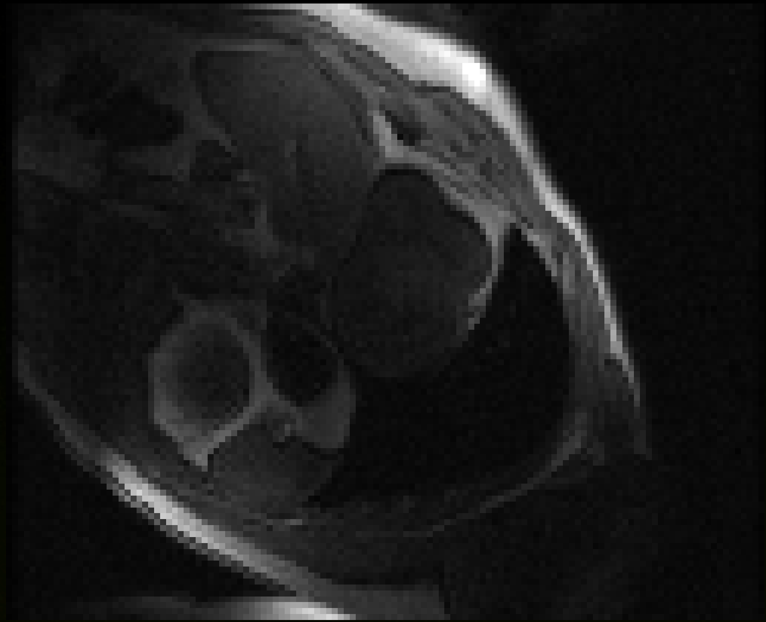


Image Processing

Dynamic Real-Time MRI





Bioengineering Institute, University of Auckland,
IUPS Physiome Project

http://www.bioeng.auckland.ac.nz/movies/database/cardiovascular_system/textured-heart-beat.mpg

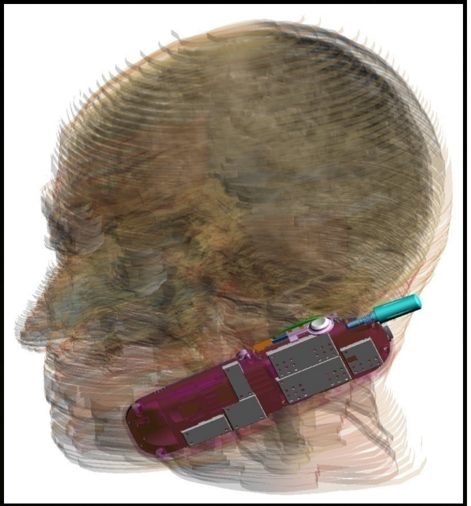
Zhi-Pei Liang's Research Group, Beckman Institute, UIUC
Used with permission of Justin Haldar

G80 GPU is 245x CPU

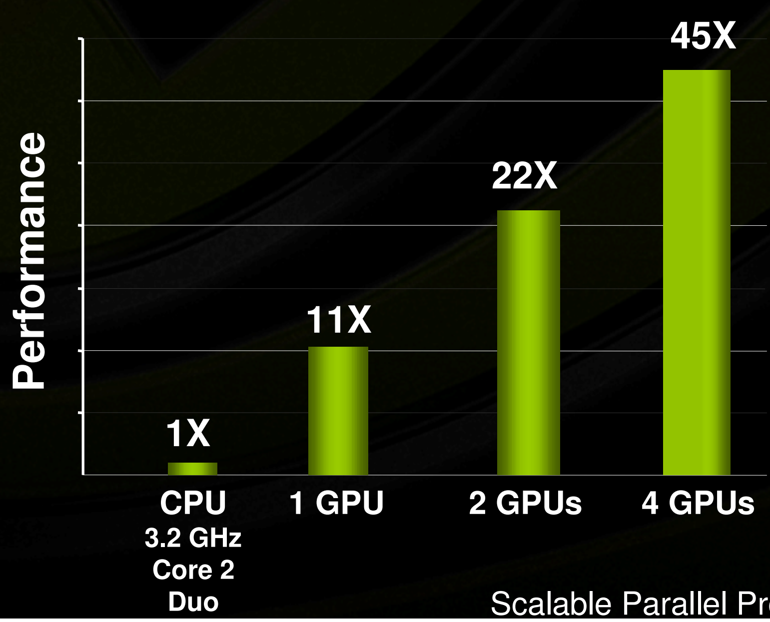
Acceleware

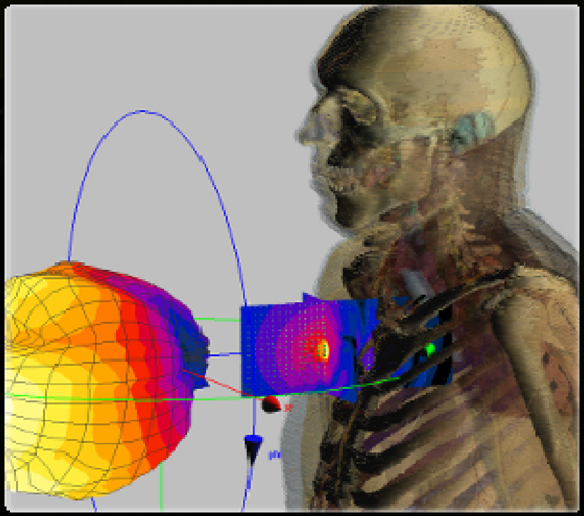
GPU Electromagnetic Field simulation

- 3D Finite-Difference and Finite-Element (FDTD)
  - Cell phone irradiation
  - MRI Design / Modeling
  - Printed Circuit Boards
  - Radar Cross Section (Military)



Cell phone EM Field





Pacemaker with Transmit Antenna

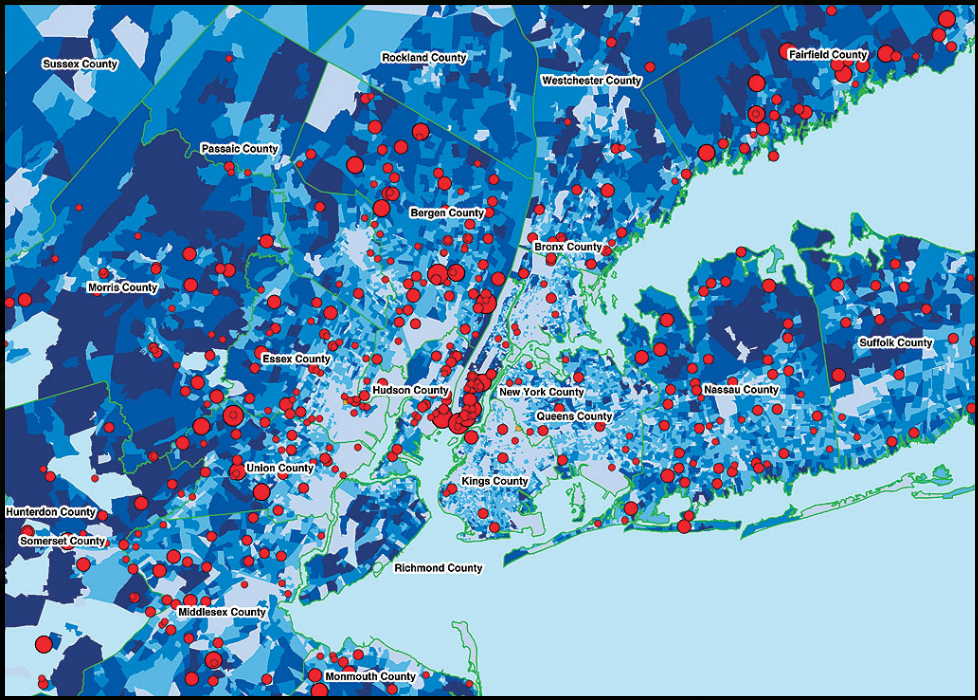
Manifold 8 GIS Application

From the Manifold 8 feature list:

… applications fitting CUDA capabilities that might have taken tens of seconds or even minutes can be accomplished in hundredths of seconds. … CUDA will clearly emerge to be the future of almost all GIS computing

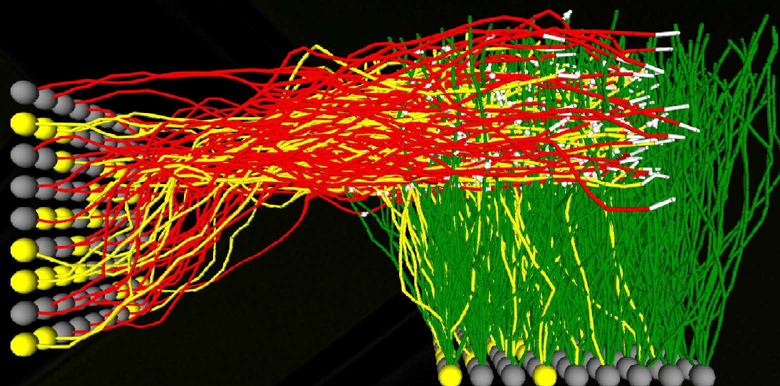
From the user manual:

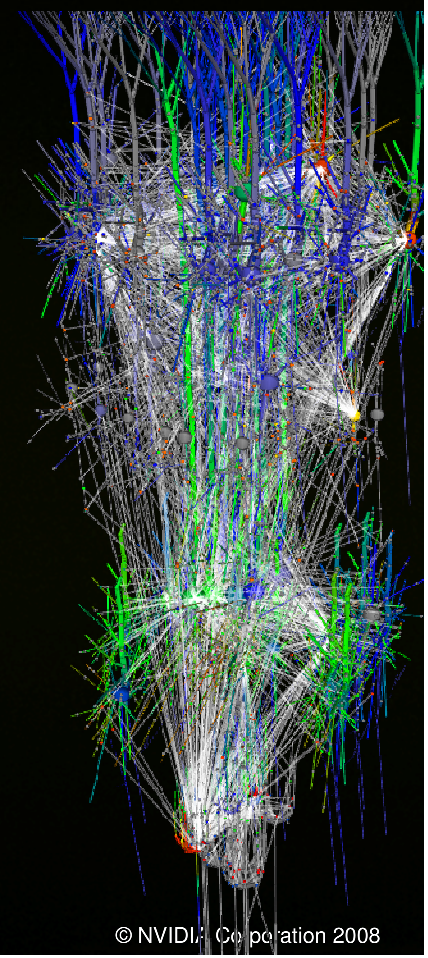
"NVIDIA CUDA … could well be the most revolutionary thing to happen in computing since the invention of the microprocessor



EvolvedMachines

- 130X Speed up
- Simulate networks of brain neurons
- Solve differential equations of ion channels
- Sensory computing: vision, olfactory

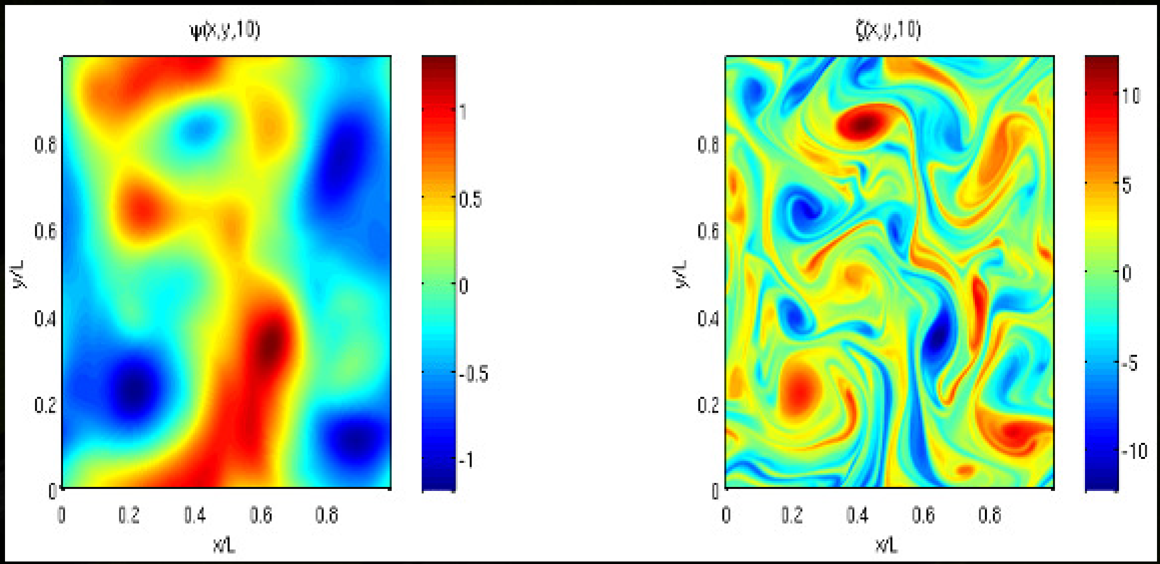




Matlab: Language of Science

17X with MATLAB CPU+GPU

http://developer.nvidia.com/object/matlab_cuda.html



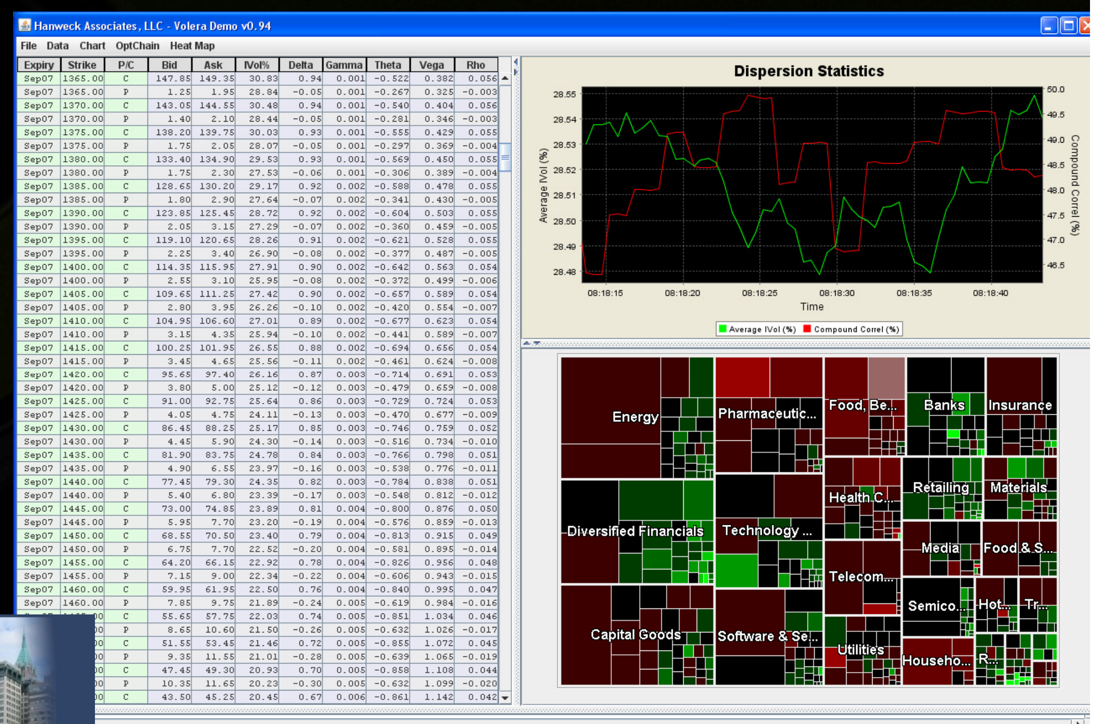
Pseudo-spectral simulation of 2D Isotropic turbulence

http://www.amath.washington.edu/courses/571-winter-2006/matlab/FS_2Dturb.m

Hanweck Associates

- VOLERA, real-time options implied volatility engine
- Accuracy results with single precision
- Evaluate all U.S. listed equity options in <1 second

[(www.hanweckassoc.com)](http://www.hanweckassoc.com)



HANWECK ASSOCIATES, LLC

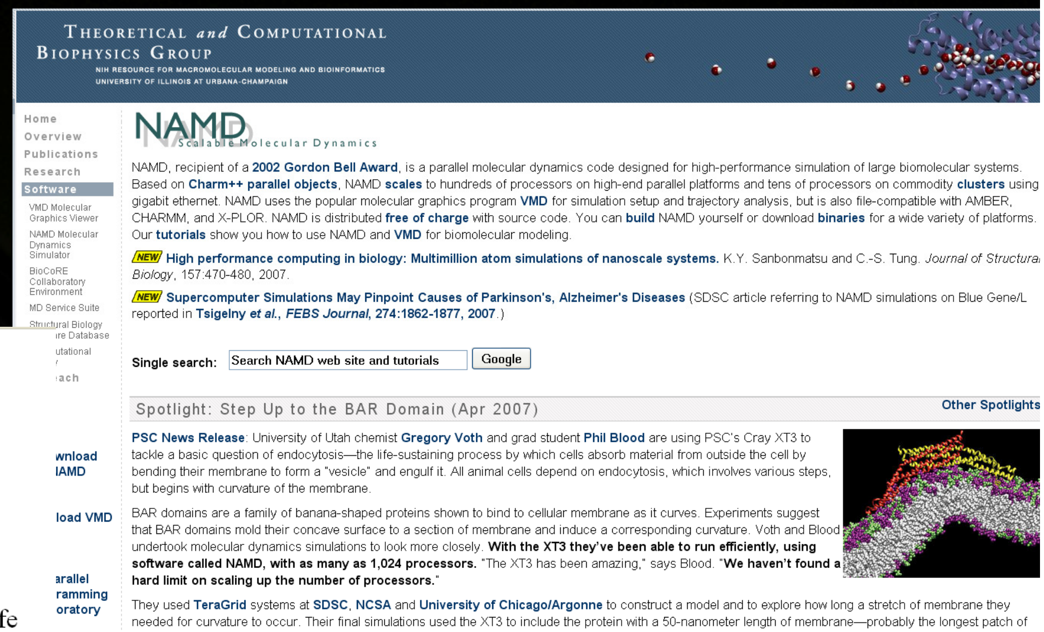
VMD/NAMD Molecular Dynamics

- 100X VMD speedup
- 240x ion placement
- Computational biology

Parallel GPUs with Multithreading: 705 GFLOPS /w 3 GPUs

- One host thread is created for each CUDA GPU
- Threads are spawned and attach to their GPU based on their host thread ID
  - First CUDA call binds that thread's CUDA context to that GPU for life
  - Handling error conditions within child threads is dependent on the thread library and, makes dealing with any CUDA errors somewhat tricky, left as an exercise to the reader…. ☺
- Map slices are computed cyclically by the GPUs
- Want to avoid false sharing on the host memory system
  - map slices are usually much bigger than the host memory page size, so this is usually not a problem for this application
- Performance of 3 GPUs is stunning!
- Power: 3 GPU test box consumes 700 watts running flat out

 21


THEORETICAL and COMPUTATIONAL BIOPHYSICS GROUP
NIH RESOURCE FOR MACROMOLECULAR MODELING AND BIOINFORMATICS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Home
Overview
Publications
Research
Software
VMD Molecular Graphics Viewer
NAMD Molecular Dynamics Simulator
BioCoRE Collaboratory Environment
MD Service Suite
Structural Biology

NAMD Scalable Molecular Dynamics

NAMD, recipient of a **2002 Gordon Bell Award**, is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. Based on **Charm++ parallel objects**, NAMD **scales** to hundreds of processors on high-end parallel platforms and tens of processors on commodity **clusters** using gigabit ethernet. NAMD uses the popular molecular graphics program **VMD** for simulation setup and trajectory analysis, but is also file-compatible with AMBER, CHARMM, and X-PLOR. NAMD is distributed **free of charge** with source code. You can **build** NAMD yourself or download **binaries** for a wide variety of platforms. Our **tutorials** show you how to use NAMD and **VMD** for biomolecular modeling.

NEW **High performance computing in biology: Multimillion atom simulations of nanoscale systems.** K.Y. Sanbonmatsu and C.-S. Tung. *Journal of Structural Biology*, 157:470-480, 2007.

NEW **Supercomputer Simulations May Pinpoint Causes of Parkinson's, Alzheimer's Diseases** (SDSC article referring to NAMD simulations on Blue Gene/L reported in **Tsigelny *et al.*, *FEBS Journal*, 274:1862-1877, 2007**.)

Single search: Search NAMD web site and tutorials Google

Spotlight: Step Up to the BAR Domain (Apr 2007) — Other Spotlights

PSC News Release: University of Utah chemist **Gregory Voth** and grad student **Phil Blood** are using PSC's Cray XT3 to tackle a basic question of endocytosis—the life-sustaining process by which cells absorb material from outside the cell by bending their membrane to form a "vesicle" and engulf it. All animal cells depend on endocytosis, which involves various steps, but begins with curvature of the membrane.

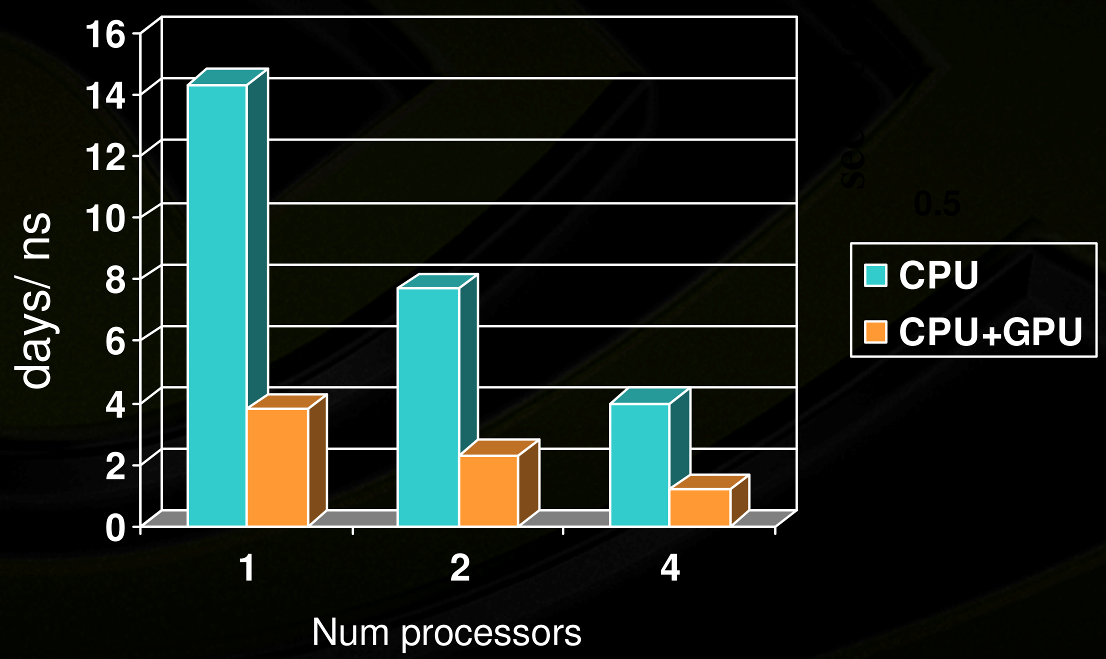
BAR domains are a family of banana-shaped proteins shown to bind to cellular membrane as it curves. Experiments suggest that BAR domains mold their concave surface to a section of membrane and induce a corresponding curvature. Voth and Blood undertook molecular dynamics simulations to look more closely. **With the XT3 they've been able to run efficiently, using software called NAMD, with as many as 1,024 processors.** "The XT3 has been amazing," says Blood. "**We haven't found a hard limit on scaling up the number of processors.**"

They used **TeraGrid** systems at **SDSC**, **NCSA** and **University of Chicago/Argonne** to construct a model and to explore how long a stretch of membrane they needed for curvature to occur. Their final simulations used the XT3 to include the protein with a 50-nanometer length of membrane—probably the longest patch of

http://www.ks.uiuc.edu/Research/vmd/projects/ece498/lecture/

NAMD acceleration on GPU cluster

- GPU cluster:
  - 1 HP DL320S (master)
  - 8 HP DL140 (compute nodes) with 3.0Ghz Woodcrest CPU
  - 8 Tesla D870



nbody Astrophysics

http://progrape.jp/cs/

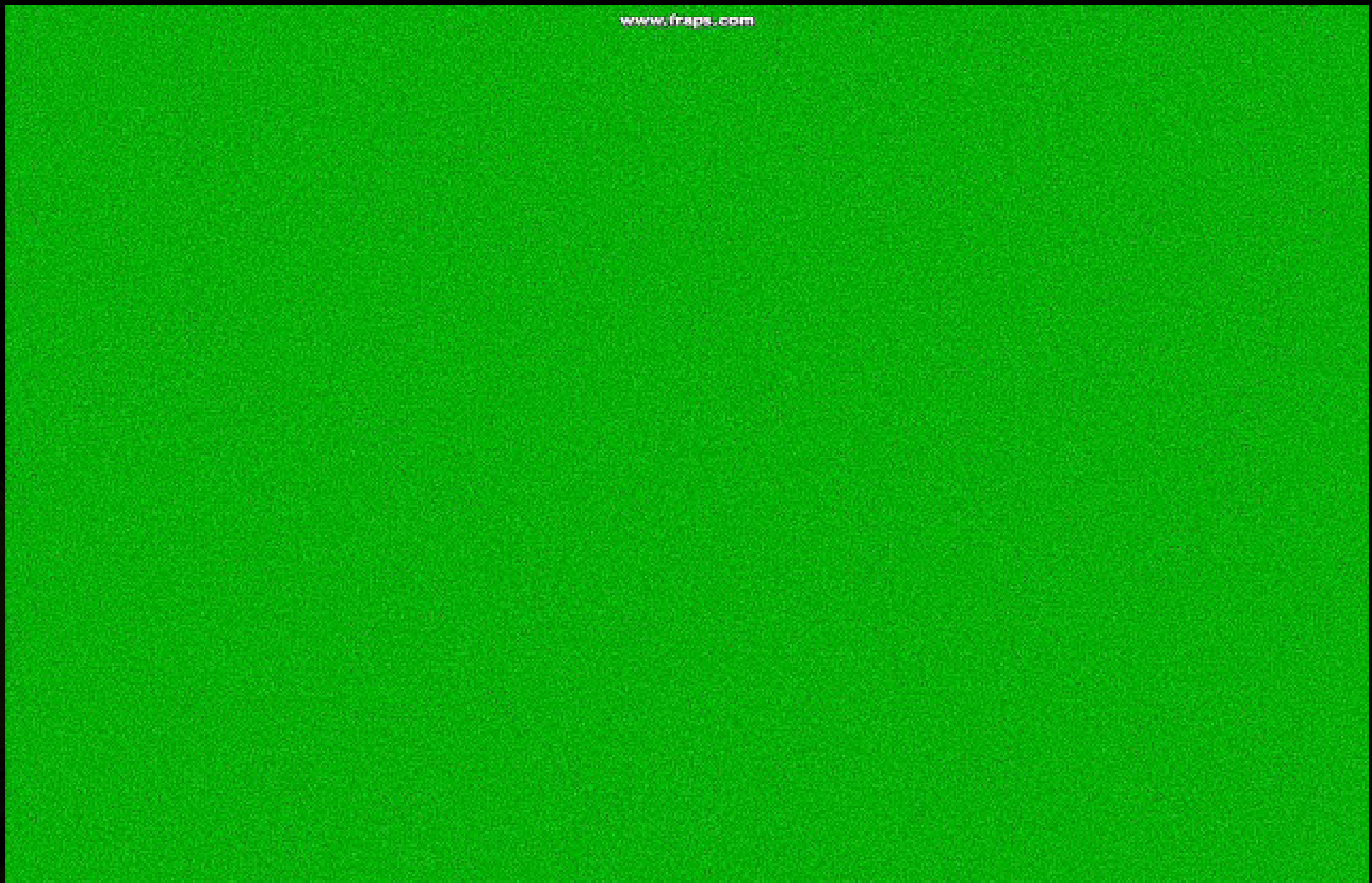
Astrophysics research

1 GF on standard PC

300+ GF on GeForce 8800GTX

Faster than GRAPE-6Af custom simulation computer

CUDA Stable Fluids Demo



CUDA port of:

Jos Stam, "Stable Fluids", In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, 121-128.

CUDA Programming Model

Examples courtesy of Michael Garland, Mark Harris, and Massimiliano Fatica / NVIDIA

CUDA Programming Model

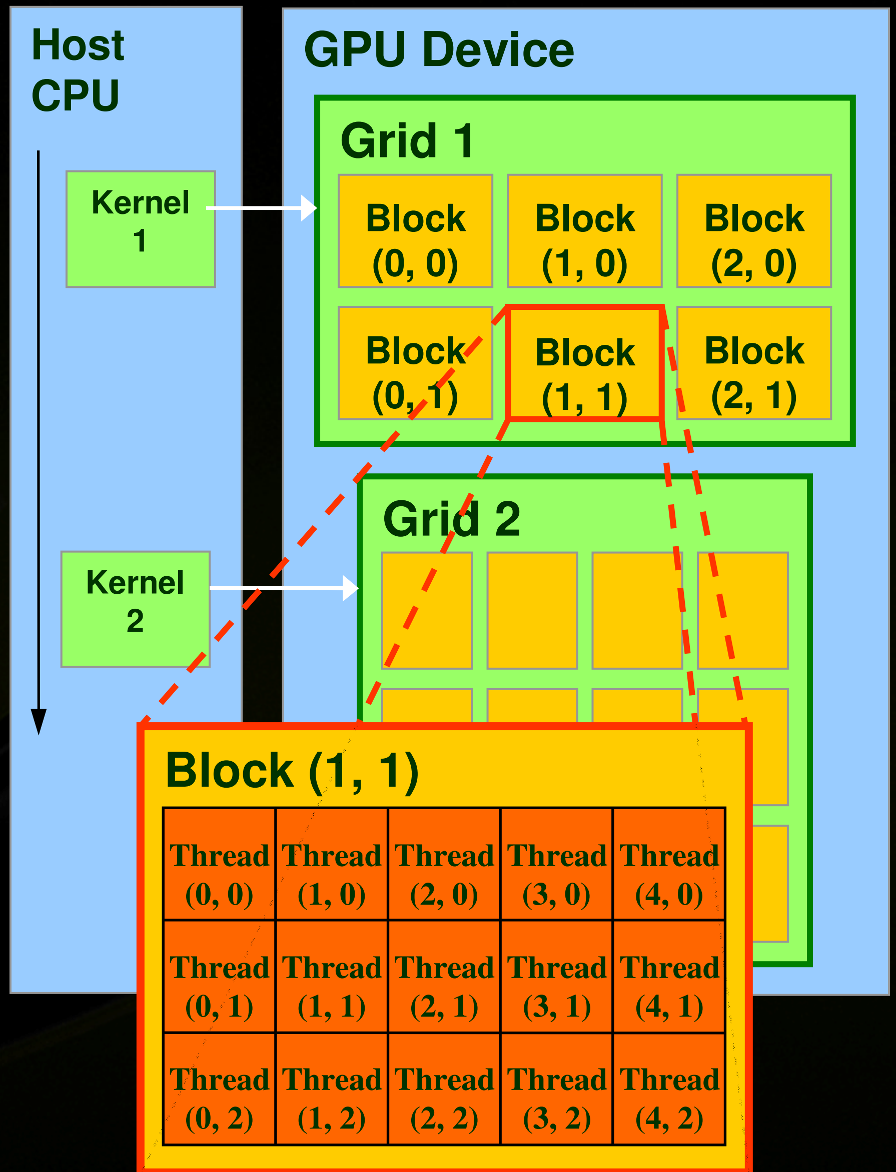
- Minimal extension of C and C++ languages
- Write a serial program that calls parallel kernels

- Serial portions execute on the host CPU
- A kernel executes as parallel threads on the GPU device
  - Kernels may be simple functions or full programs
  - Many threads execute each kernel

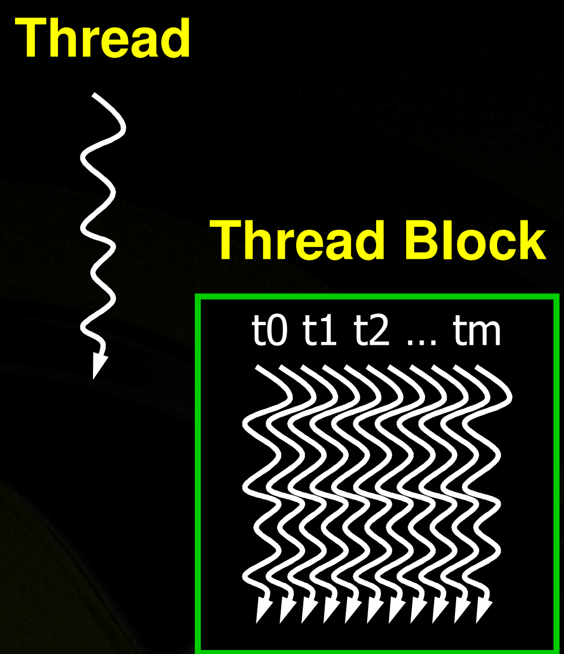
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Tiny thread creation overhead
    - Zero-overhead thread scheduling
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few
    - CUDA uses threads for fine-grained parallelism
    - CUDA uses blocks of threads for coarse-grained parallelism

CUDA Grids of Thread Blocks

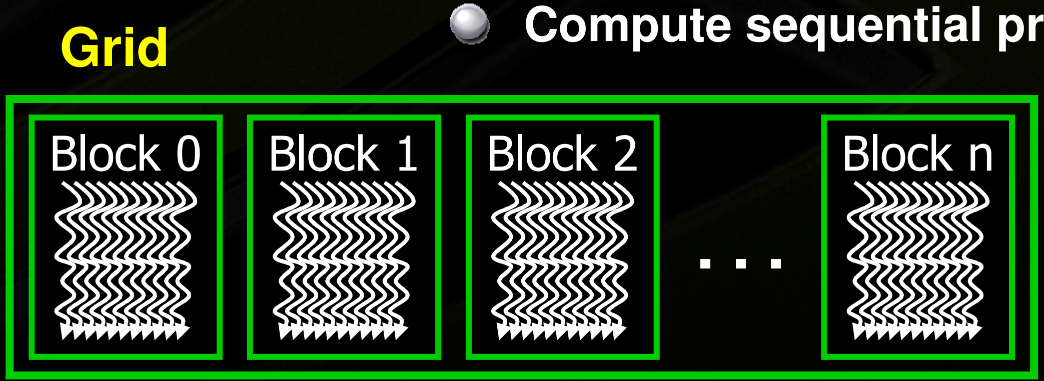
- Organize kernel threads into grids of thread blocks
- A thread block is an array of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Thread blocks of a grid execute independently



CUDA Hierarchy of thread groups

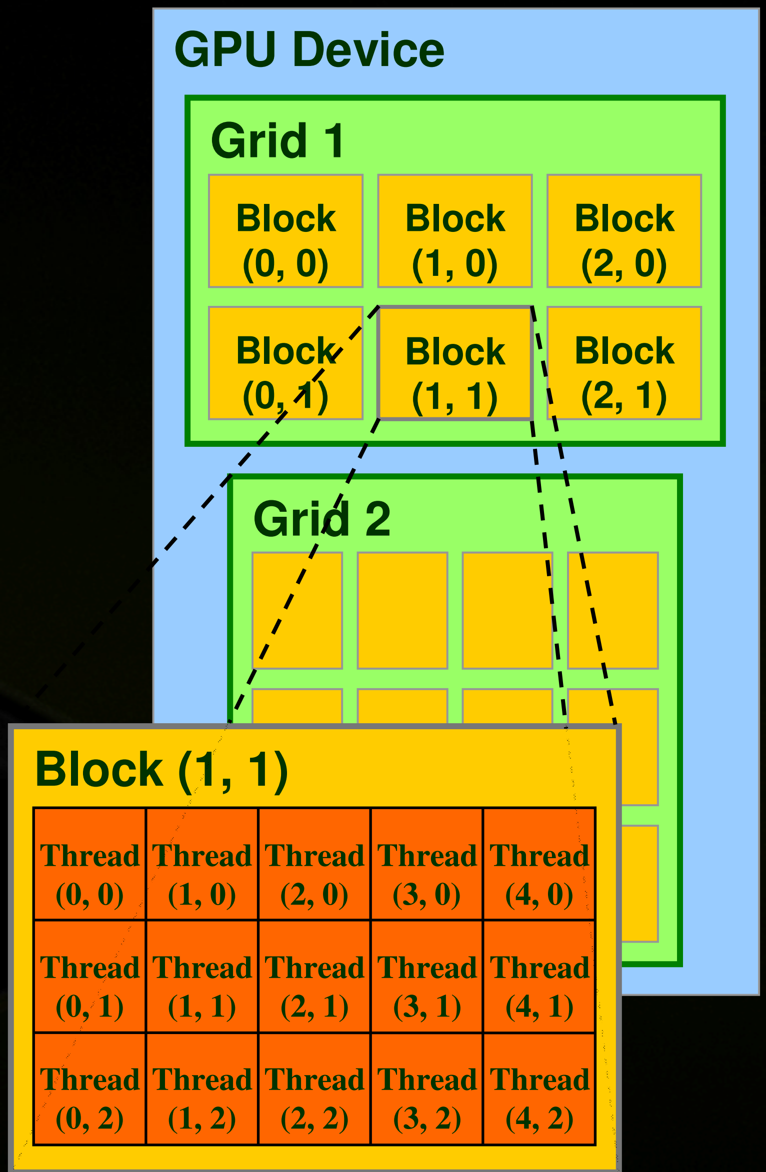


- Thread
  - Computes result elements
  - `threadIdx` is thread id number
- Thread Block
  - Computes result data Block
  - 1 to 512 threads per Thread Block
  - `blockIdx` is block id number
- Grid of Blocks
  - Computes many result blocks
  - 1 to many blocks per grid
- Sequential Grids
  - Compute sequential problem steps



CUDA Thread ID and Block ID

- Threads and blocks have IDs
  - Each thread selects what data to work on
  - Using built-in variables
- 1D, 2D, 3D blocks and grids
- Block and thread IDs
  - Built-in variables:
  - `blockIdx  .x,  .y`
  - `threadIdx .x, .y, .z`
- Grid and block dimensions
  - Built-in variables:
  - `gridDim   .x,  .y`
  - `blockDim  .x, .y, .z`



Launching parallel CUDA kernels

- Declare kernel entry procedure as `__global__`
- Extended function call syntax:

```
kernel<<<dimGrid, dimBlock>>>(... parameter list ...);
kernel<<<32, 256>>>(... parameter list ...);
```

- Specify dimensions of grid in blocks
  - Grid dimensions: x, y
    `dim3 dimGrid(16, 16);`
- Specify dimensions of the blocks in threads
  - Unspecified `dim3` dimensions are 1
  - Thread-block dimensions: x, y, z
    `dim3 dimBlock(16,16);`
- Kernel function parameters in (. . .)

SAXPY: y=ax+y in C, parallel CUDA

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

```
__global__
void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

CUDA 2D Example: Add Arrays

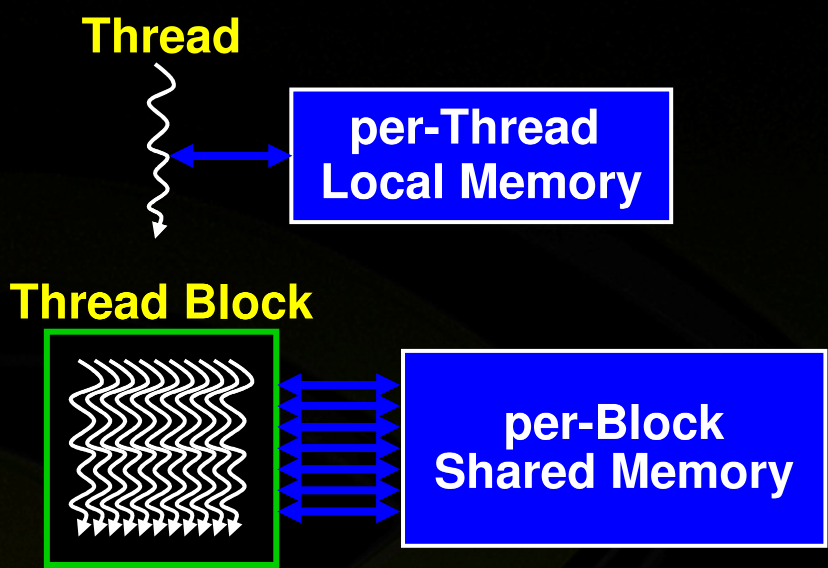
C program

```
void  addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
                idx = i + j*N;
                c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    .....
        addMatrix(a, b, c, N);
}
```

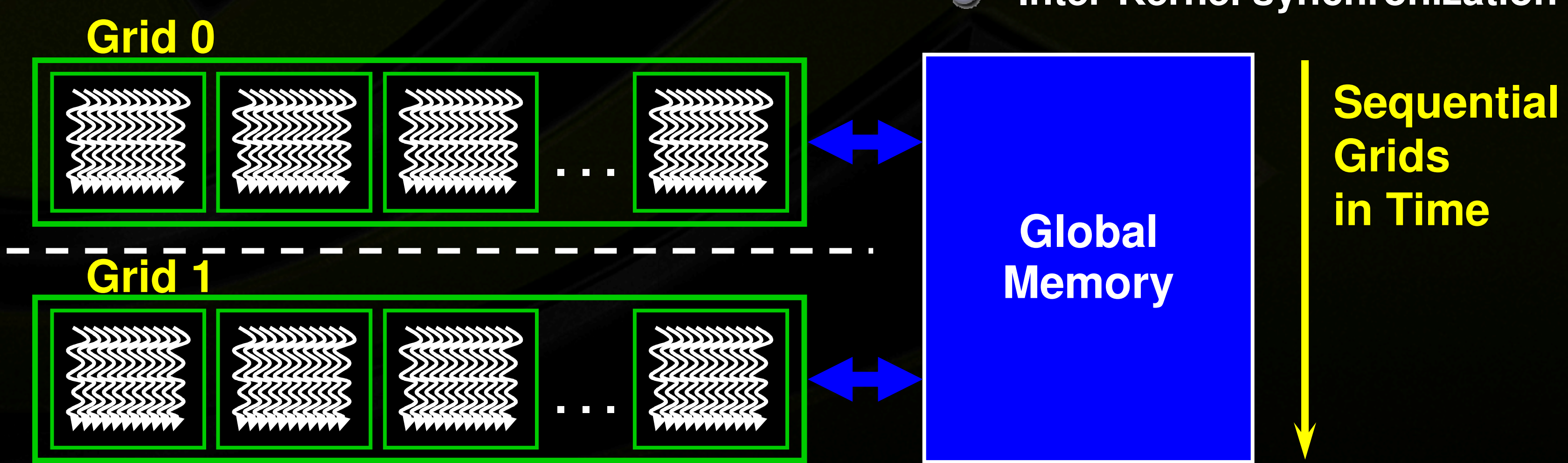
CUDA C program

```
__global__ void  addMatrixG
        (float *a, float *b, float *c, int N)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        int j = blockIdx.y*blockDim.y + threadIdx.y;
        int idx = i + j*N;
        if (i < N && j < N)
                c[idx] = a[idx] + b[idx];
}

void  main()
{
        dim3 dimBlock (blocksize, blocksize);
        dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
        addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

CUDA Parallel Memory Sharing



- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-block
  - Shared by threads of block
  - Inter-thread communication
  - Barrier synchronization
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication
  - Inter-Kernel synchronization



CUDA Kernel Variable Qualifiers

- `__device__`
  - stored in global device memory (large, high latency)
  - global memory accessible by all threads
  - lifetime: application
- `__shared__`
  - stored in per-block shared memory (small, low latency)
  - accessible by all threads in the same thread block
  - lifetime: kernel thread block
- **Unqualified variables:**
  - scalars and built-in vector types are in registers
  - arrays are stored in per-thread device memory

CUDA Synchronization

- Barrier synchronization among threads of block
  - Fast single-instruction barrier in Tesla GPUs
  - `void __syncthreads();`
  - Synchronizes all threads in a thread block
  - Once all threads have reached this point, kernel execution resumes normally
  - Use before reading shared memory written by another thread in the same block
- Global synchronization between dependent kernels
  - Waits for all thread blocks of kernel grid to complete
  - Fast synchronization and kernel launch in Tesla GPUs

CUDA Atomic Integer Operations

- Atomic operations on integers in global memory:
  - `atomicAdd(int *pmem; int value)`
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
- Requires Tesla 1.1 architecture or later GPU
- Eliminates last stage of a parallel reduction
- Useful for atomic data structure management

CUDA Memory Management

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);


// execute a kernel
kernel<<< N/blockSize, blockSize >>>(d_A, b);


// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

SpMV: Sparse Matrix-Vector Product

- SpMV: *y = Ax* for sparse *n* x *n* matrix *A*
- Sparse *n* x *n* matrix *A* stores only *m* non-zero entries
- Compressed Sparse Row (CSR) representation
- Array `Av[m]` stores non-zero values of *A*
- Array `Aj[m]` stores column index for corresponding `Av[ ]`
- Array `Ap[n+1]` stores extent of prior row
- Row `i` extends from `Ap[i]` up to but not including `Ap[i+1]`
- `Ap[0] == 0, Ap[n] == m`

$$A = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

(a) Sample matrix *A*

	Row 0	Row 2	Row 3
`Av[7] = {`	3 1	2 4 1	1 1 `}`
`Aj[7] = {`	0 2	1 2 3	0 3 `}`

`Ap[5] = { 0 2 2 5 7 }`

(b) CSR representation of matrix

SpMV: One row of y = Ax

- Given sparse matrix *A* in CSR form `Av[]`, `Aj[]`
- Compute one row of *y = Ax*
- Identical C and CUDA code below

```
float mult_row(unsigned rowsize,
      unsigned *Aj,     // column indices for row
      float *Av,        // non-zero entries for row
      float *x)         // the RHS vector
{
    float sum = 0;
    for (unsigned column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];
    return sum;
}
```

SpMV: Serial C Loop: y = Ax

- Serial code loops over all rows, calls `mult_row();`

```
void csrmul_serial(unsigned *Ap, unsigned *Aj,
  float *Av, unsigned nrows, float *x, float *y)
{
    for (unsigned row=0; row < nrows; ++row) {
        unsigned row_begin = Ap[row];
        unsigned row_end   = Ap[row+1];
        y[row] = mult_row(row_end-row_begin,
                          Aj+row_begin, Av+row_begin, x);
    }
}
```

SpMV: Parallel CUDA kernel: y = Ax

- CUDA parallel kernel code for one thread
- Each thread computes one row of vector y

```
__global__
void csrmul_kernel(unsigned *Ap, unsigned *Aj,
  float *Av, unsigned nrows, float *x, float *y)
{
    unsigned row = blockIdx.x*blockDim.x + threadIdx.x;
    if (row < nrows) {
        unsigned row_begin = Ap[row];
        unsigned row_end   = Ap[row+1];
        y[row] = mult_row(row_end-row_begin,
                          Aj+row_begin, Av+row_begin, x);
    }
}
```

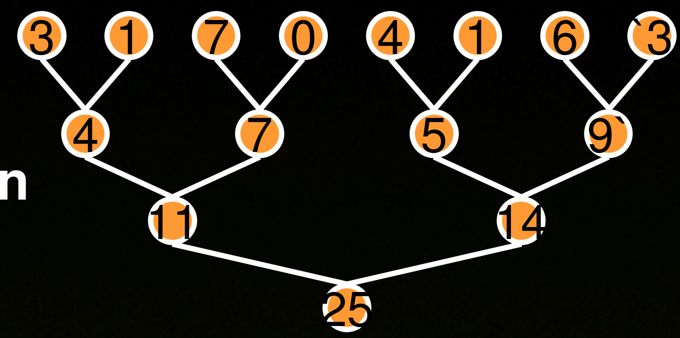
SpMV: CUDA mainline

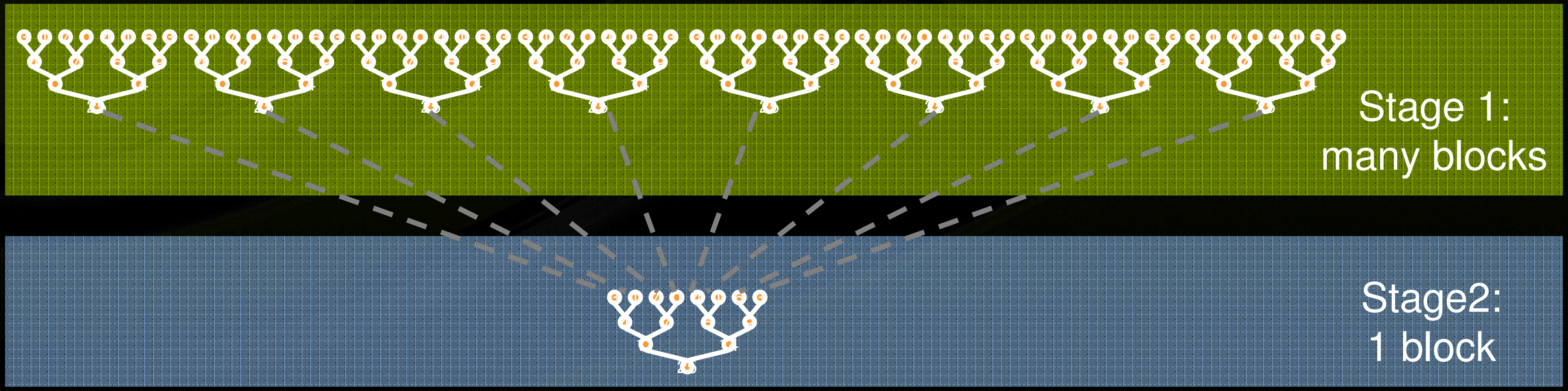
- Copy sparse matrix data A and x to device memory
  - `cudaMemcpy();`
- Invoke parallel kernel on grid of thread blocks
  - csrmul_kernel<<<dimg, dimb>>>(parameters);
- Copy result data y from device memory
  - `cudaMemcpy();`

```
unsigned blocksize = 128;      // or any size up to 512
unsigned nblocks   = (nrows + blocksize - 1)/blocksize;

csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, nrows, x, y);
```

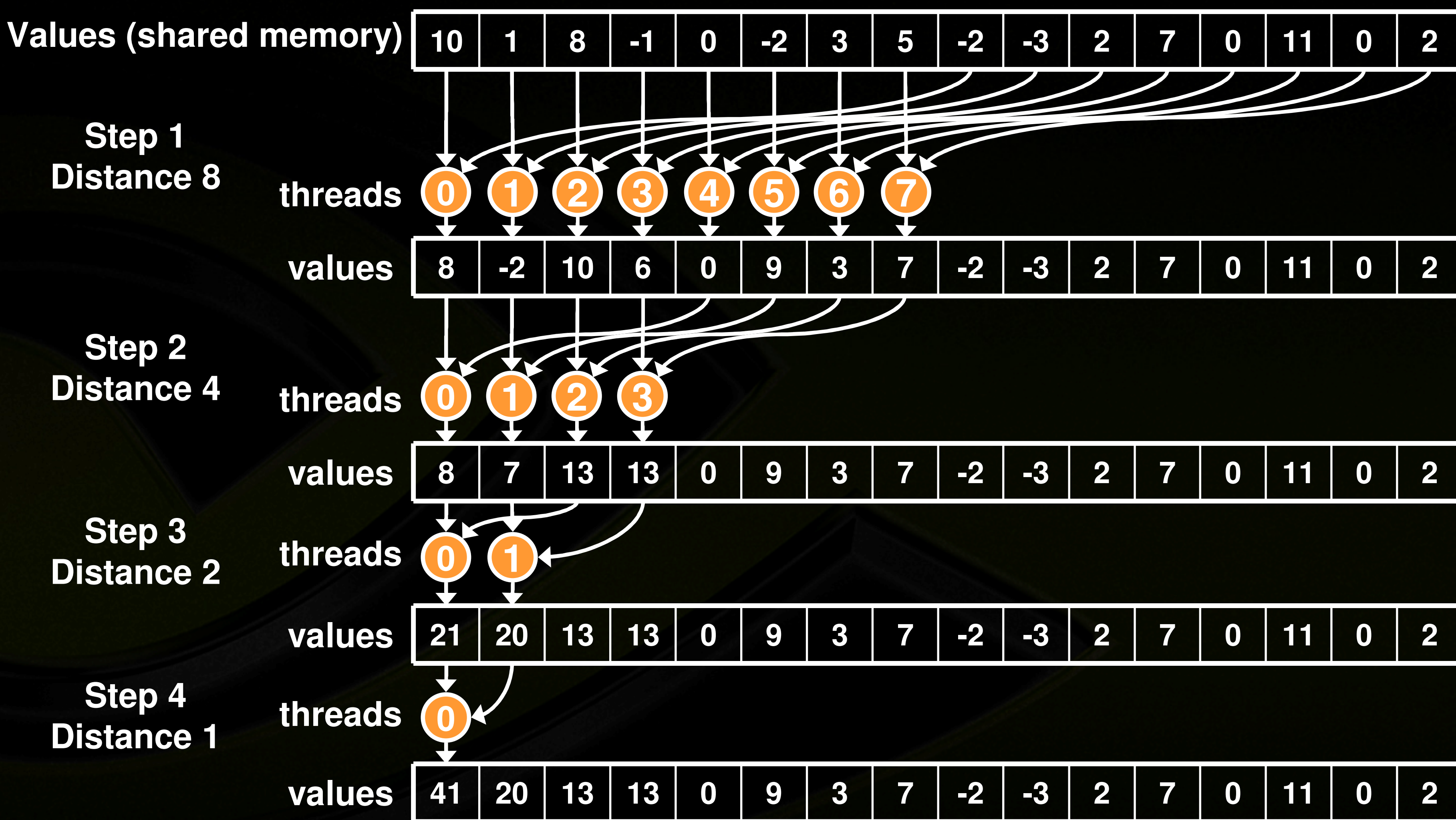
Parallel Sum Reduction

- Reduction is a common data parallel operation
  - Reduce vector to a scalar value
  - Operator: +, *, min, max, AND, OR
  - $O(\log_2 N)$ tree-based implementation
- Two stages of computation:
  - Sum within each block
  - Sum partial results from the blocks
  - Final stage repeats kernel, or uses atomicAdd()





Reduction Kernel execution



CUDA Sum Reduction Kernel

```
__global__
reduce(int *g_idata,
       int *g_odata)
{
  extern __shared__
  int data[];

  int t  = threadIdx.x;
  int b  = blockIdx.x;
  int bd = blockDim.x;
  int i  = b * bd + t;

  // load shared mem
  data[t] = g_idata[i];
  __syncthreads();
```

```
   // reduce in shared mem
  for (int s = bd/2; s>0; s >>= 1)
  {
    if (t < s)
      data[t] += data[t + s];
    __syncthreads();
  }

  // global mem += block sum
  if (t == 0)
    atomicAdd(g_odata, data[0]);
}
```

CUDA Reduction Kernel

```
__global__ void sum_kernel(int *g_input, int *g_output)
{
    extern __shared__ int s_data[];  // allocated at kernel launch

    // read input into shared memory
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[threadIdx.x] = g_input[idx];
    __syncthreads();

    // compute sum for the thread block
    for (int dist = blockDim.x/2; dist > 0; dist /= 2)
    {
        if (threadIdx.x < dist)
            s_data[threadIdx.x] += s_data[threadIdx.x + dist];
        __syncthreads();
    }
    // write the block's sum to global memory
    if (threadIdx.x==0)
        g_output[blockIdx.x] = s_data[0];
}
```

Reduction Host Source Code (1)

```
int main()
{
    // data set size in elements and bytes
    unsigned int n = 4096;
    unsigned int nbytes = n*sizeof(int);

    // launch configuration parameters
    unsigned int block_dim = 256;
    unsigned int nblocks = n / block_dim;
    unsigned int smem_bytes = block_dim*sizeof(int);

    // allocate and initialize the data on the CPU
    int *h_a=(int*)malloc(nbytes);
    for (int i=0; i < n; i++)
        h_a[i]=1;

    // allocate memory on the GPU device
    int *d_a=0, *d_out=0;
    cudaMalloc((void**)&d_a, nbytes);
    cudaMalloc((void**)&d_out, nblocks*sizeof(int));
```

Reduction Host Source Code (2)

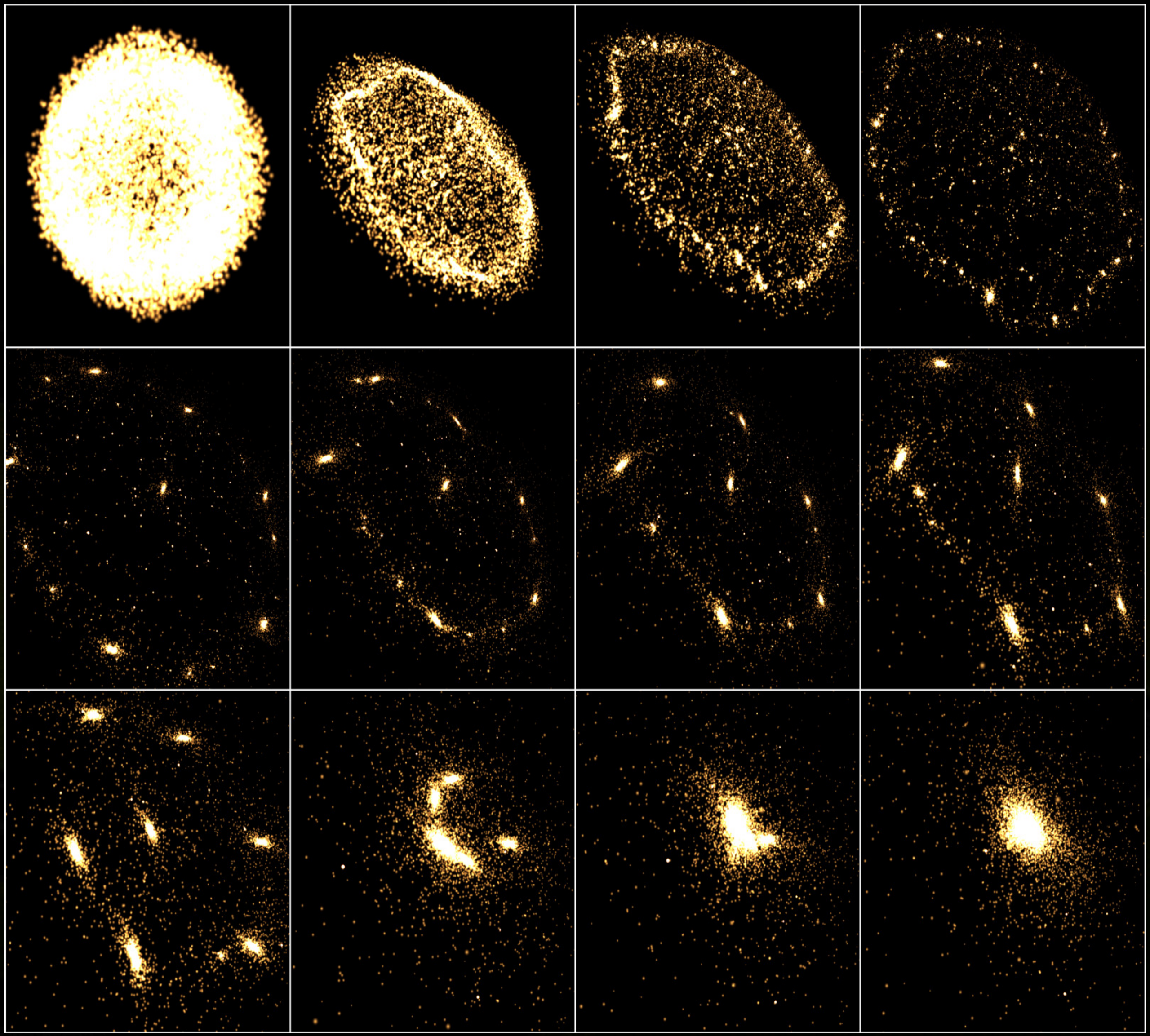
```
        ...

    // copy the input data from CPU to the GPU device
    cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);

    // two stages of kernel execution
    sum_kernel<<<nblocks, block_dim, smem_bytes>>>(d_a, d_out);
    sum_kernel<<<1, nblocks, nblocks*sizeof(int)>>>(d_out, d_out);

    // copy the output from GPU device to CPU and print
    cudaMemcpy(h_a, d_out, sizeof(int), cudaMemcpyDeviceToHost);
    printf("%d\n", h_a[0]);

    // release resources
    cudaFree(d_a);
    cudaFree(d_out);
    free(h_a);

    return 0;
}
```

N-Body Simulation

Courtesy of Mark Harris

- Numerically simulate evolution of system of *N* bodies
  - Each body continuously interacts with all other bodies
- Examples:
  - Astronomical and astrophysical simulation
  - Molecular dynamics simulation
  - Fluid dynamics simulation
  - Radiometric transfer (Radiosity, multiple scattering, etc.)
- N^2 interactions to compute per time step
  - For the brute force *all-pairs* approach we discuss here

CUDA N-Body Simulation



10B interactions / s

16K bodies
44 FPS
x 20 FLOPS / interaction
x 16K^2 interactions / frame
= 240 GFLOP/s

= 50x tuned CPU implementation on Intel Core 2 Duo

GeForce 8800 GTX GPU

Highly Parallel
High Arithmetic Intensity

Papers about N-Body on CUDA

- "Fast N-Body Simulation with CUDA"
  - Nyland, L., Harris, M., and Prins, J.
  - *GPU Gems 3*
- *"Accelerating Molecular Modeling Applications with Graphics Processors"*
  - *John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, Klaus Schulten*
  - *J. Comp. Chem. (Submitted)*
- "The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units"
  - Hamada, T. and T. Iitaka.
  - Submitted to *NewAstronomy*, 5 Mar, 2007
- "High Performance Direct Gravitational N-body Simulations on Graphics Processing Units – II: An implementation in CUDA"
  - Belleman, R. G., J. Bedorf, S. Portegies Zwart.
  - Accepted for publication in *NewAstronomy*
- "Graphic-Card Cluster for Astrophysics (GraCCA) -- Performance Tests"
  - Schive, H-Y, C-H Chien, S-K Wong, Y-C Tsai, T. Chiueh.
  - Submitted to *NewAstronomy*, 20 July, 2007
  - Cluster of 32 GeForce 8800 GTX GPUs: 7.1 TFLOP/s measured!

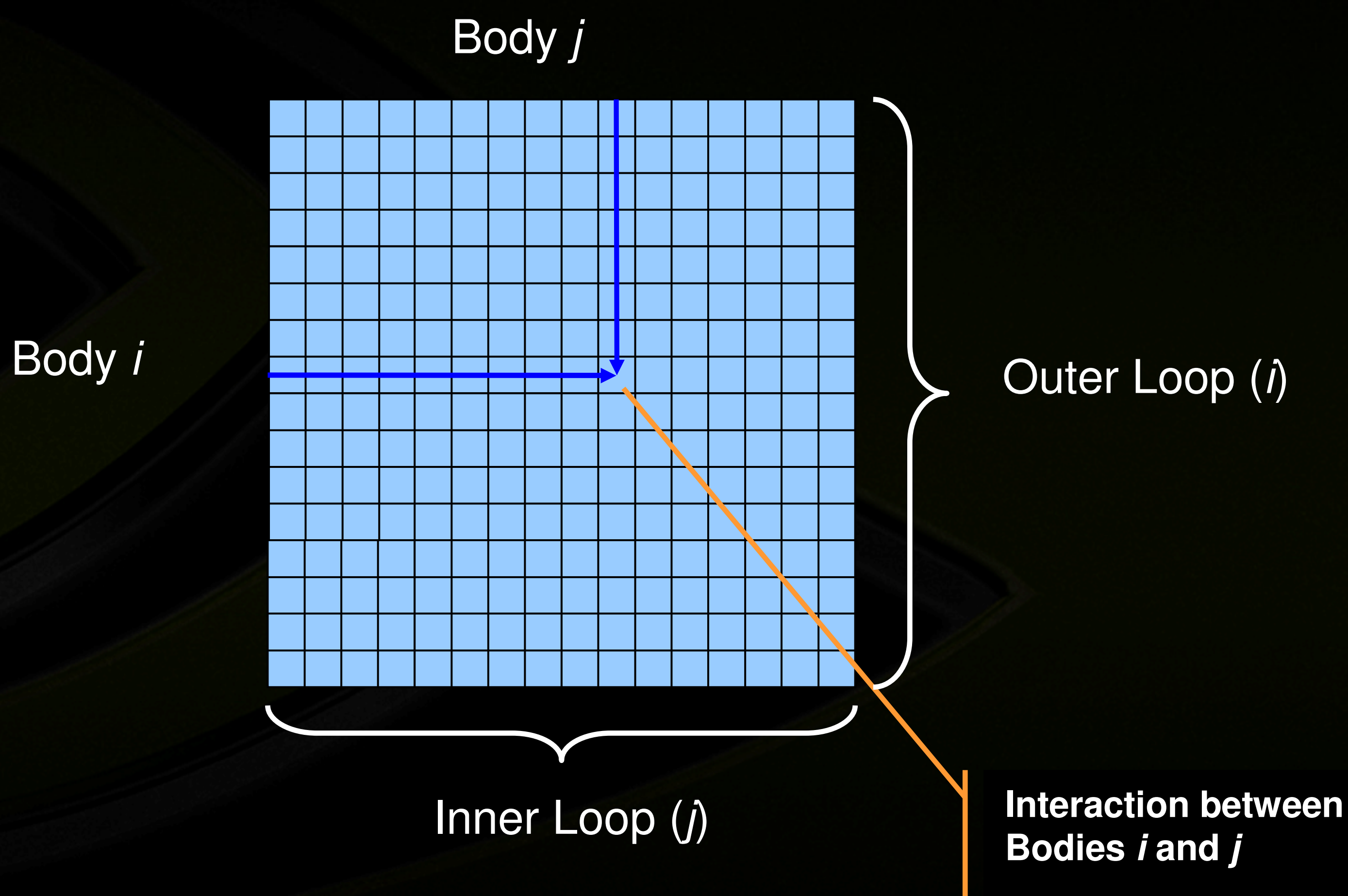
Sequential N-Body Algorithm

```
foreach body i {
  accel = 0;
  pos_i = position[i]
  foreach body j {
    pos_j = position[j]
    accel +=
      computeAcceleration(pos_i, pos_j)
  }
  // Leapfrog-Verlet integration*
  velocity[i] += accel * timestep
  position[i] += velocity[i] * timestep
}
```

*Any integration scheme can be used

Sequential N-Body Algorithm

- **Conceptual grid of interactions between (*i*,*j*) pairs**



Approach to N-Body Parallelism

- This is *very* parallel: one thread per body
  - Acceleration on all bodies can be computed in parallel
- Blocks of *p* threads process *p* bodies at a time

```
forall bodies i in parallel {
  accel = 0;
  pos_i = position[i]
  foreach body j {
    pos_j = position[j]
    accel +=
      computeAcceleration(pos_i, pos_j)
  }
}
```

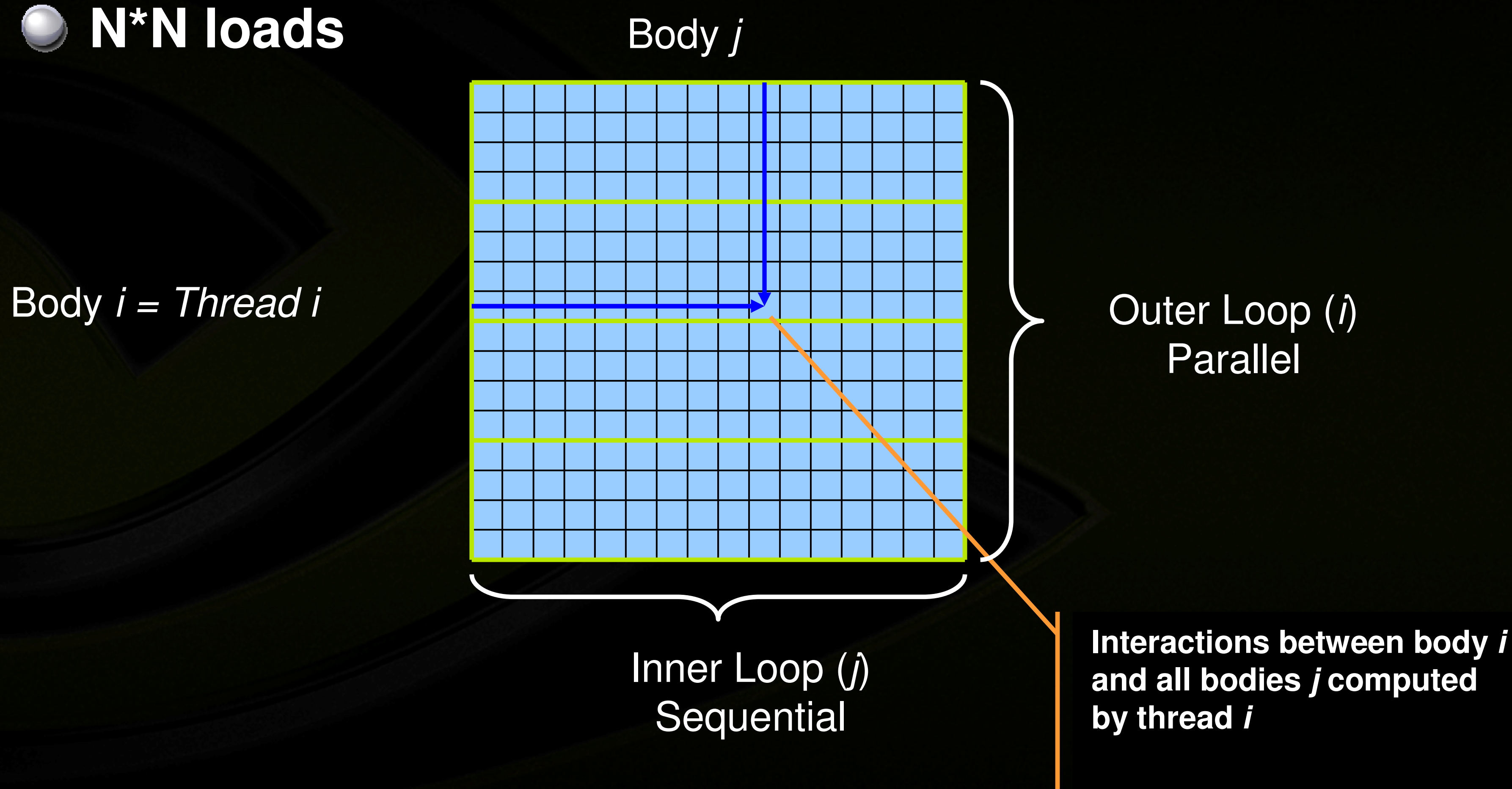
Inefficient Parallel Approach

```
forall bodies i in parallel
{
  accel = 0
  pos_i = position[i]
  foreach body j
  {
    pos_j = position[j]
    accel +=
   computeAccel(pos_i,pos_j);
  }
}
```

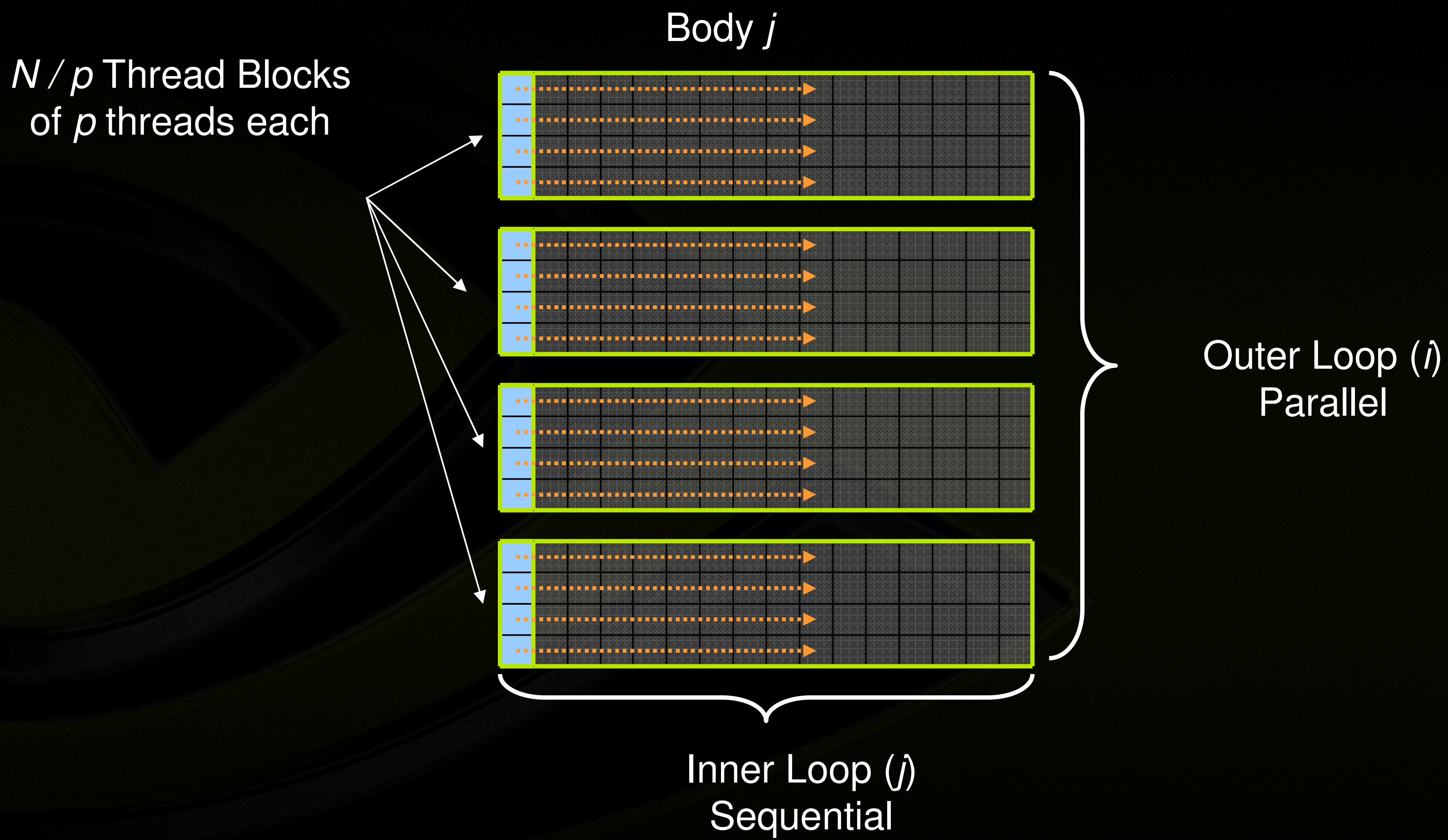
- **Every thread loads all body positions from off-chip memory**
- **N^2 loads: Bandwidth bound**
- **86 GB/s peak / 16 bytes per position = 5.4B interactions/s *theoretical* peak**
- **108 GFLOP/s < ½ what G80 achieves on efficient n-body code**

Inefficient Parallel Implementation

- N threads
- N*N computations
- N*N loads

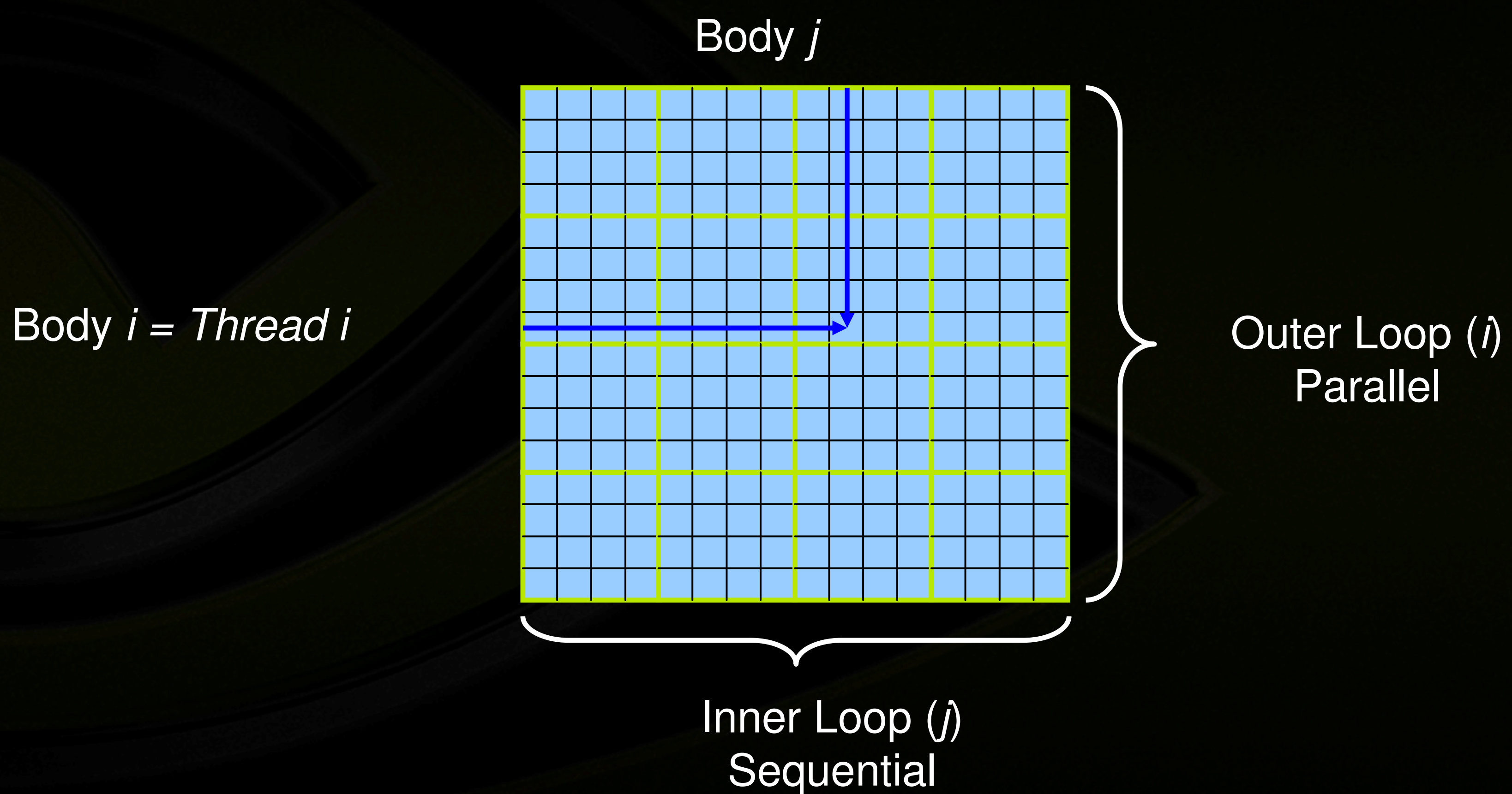


Inefficient CUDA Implementation



Shared Memory Solves B/W Bottleneck

- **Use fast on-chip per-block shared memory**
  - **Share blocks of body positions between threads**
  - **Break grid into conceptual *tiles***



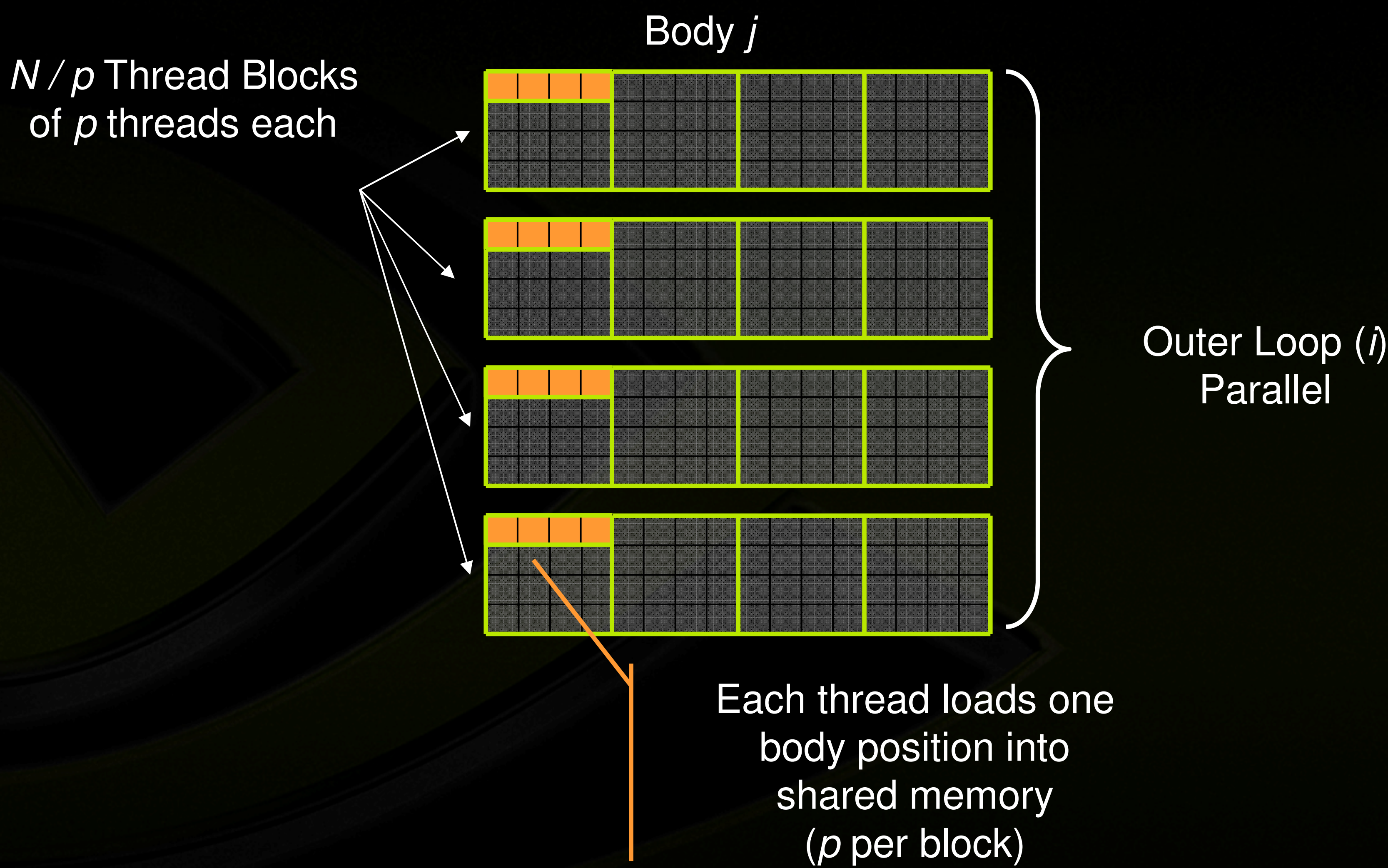
CUDA Tiled Parallel Approach

```
forall bodies i in parallel {
  accel = 0;
  pos_i = position[i]
  foreach tile q {
    forall threads p in thread block in parallel {
      shared[p] = position[q*tile_size + p]
    }
    synchronize threads in block
    foreach body j in tile q {
      pos_j = shared[j]
      accel +=
        computeAcceleration(pos_i, pos_j)
    }
    synchronize threads in block
  }
}
```

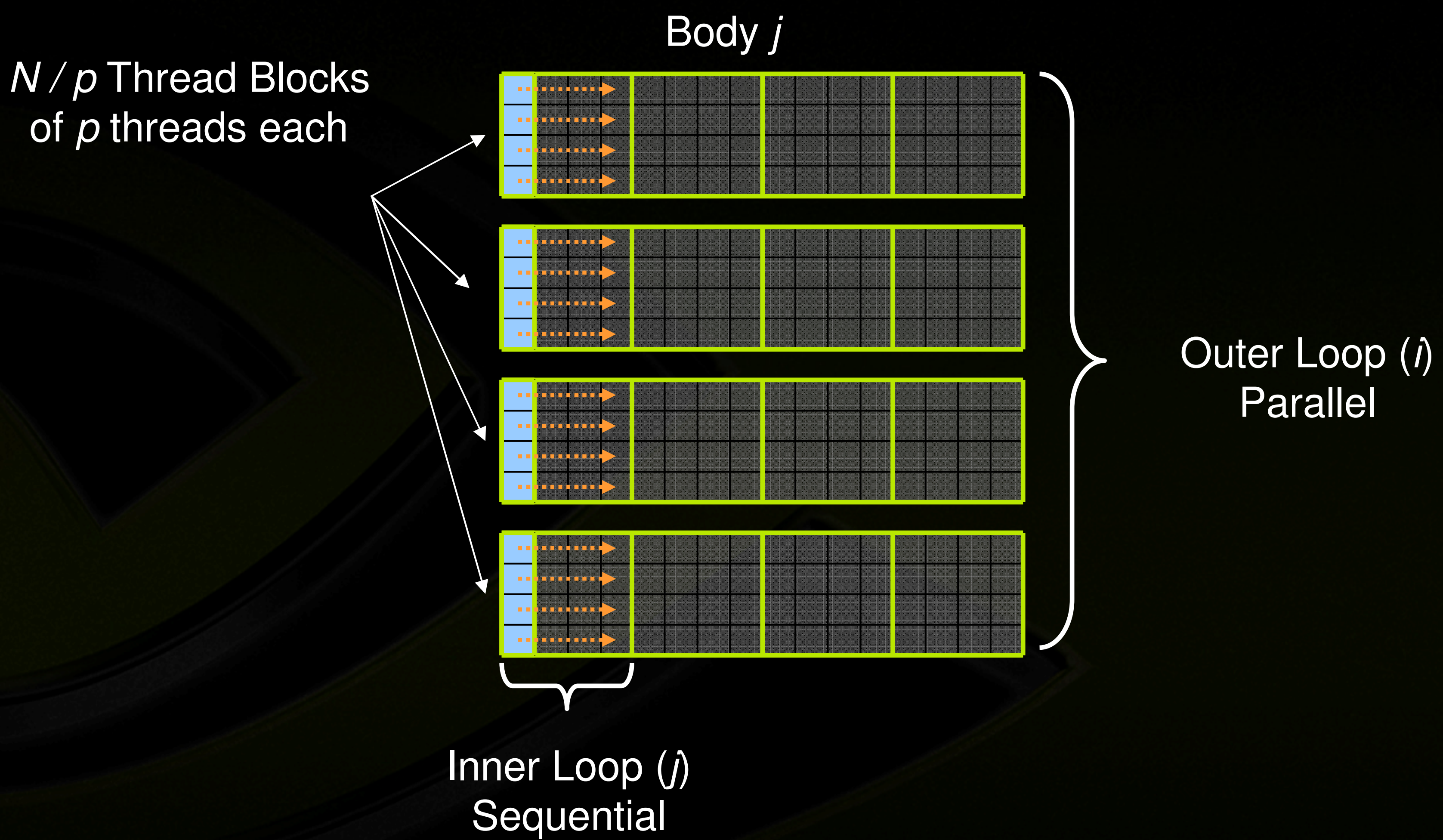
CUDA Tiled Parallel Approach

- Sequential inner loop split into *N*/*p* sub-loops over tiles
  - Threads in a block cooperatively load *p* positions within a tile to shared memory
- Reduces # of loads to N^2 / p
  - Typically use *p* = 256 threads, so big savings!
  - Compute bound, good performance
  - 10B interactions / s = 205+ GFLOP/s

CUDA Tiled Parallel Implementation



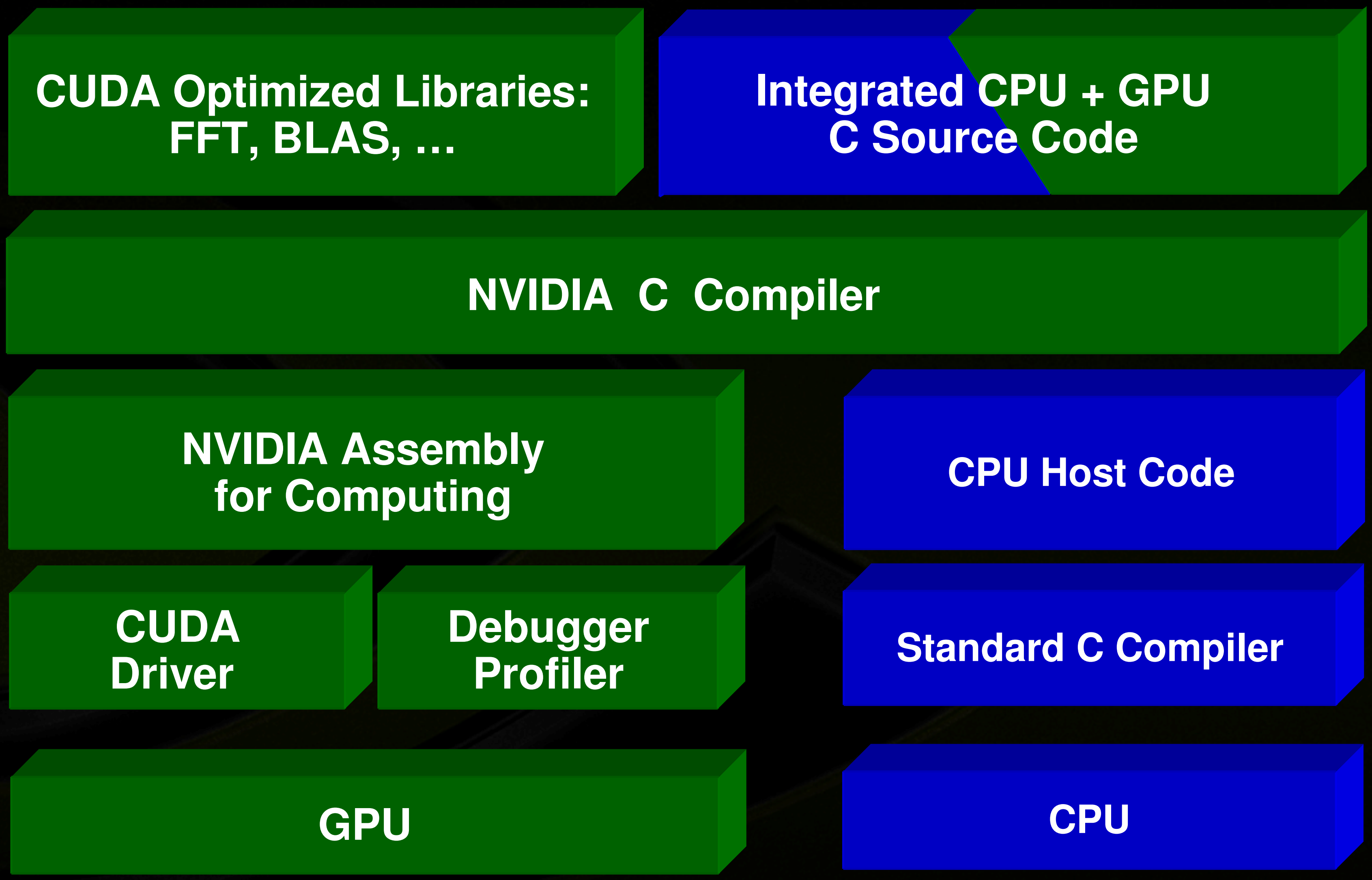
CUDA Tiled Parallel Implementation



N-Body Physics on CUDA

- All-pairs gravitational N-body physics of 16,384 stars
- 240 GFLOPS on NVIDIA GeForce 8800 – see GPU Gems 3

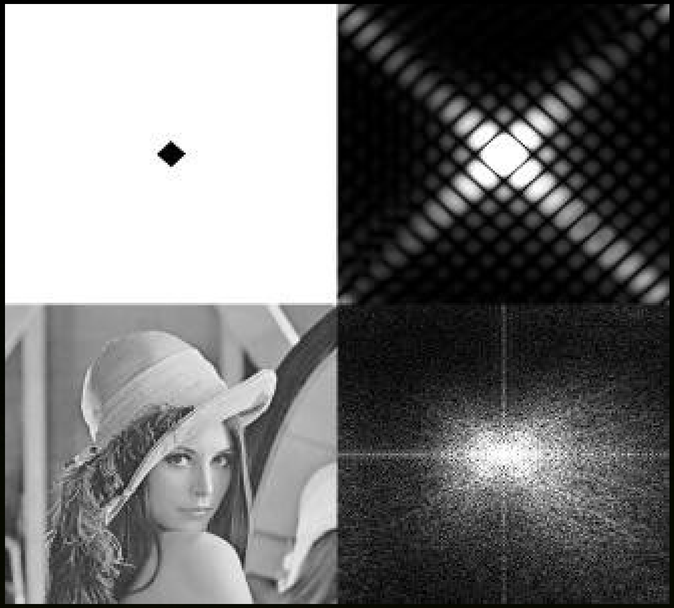
CUDA Software Development Kit



CUBLAS Library

- Self-contained BLAS library
  - Application needs no direct interaction with CUDA driver
- Currently a subset of BLAS core functions
  - Single/Real Routines, BLAS1 Complex, CGEMM
- Simple to use:
  - Create matrix and vector objects in GPU memory
  - Fill them with data
  - Call sequence of CUBLAS functions
  - Upload results back from GPU to host
- Column-major storage and 1-based indexing
  - For maximum compatibility with existing Fortran apps

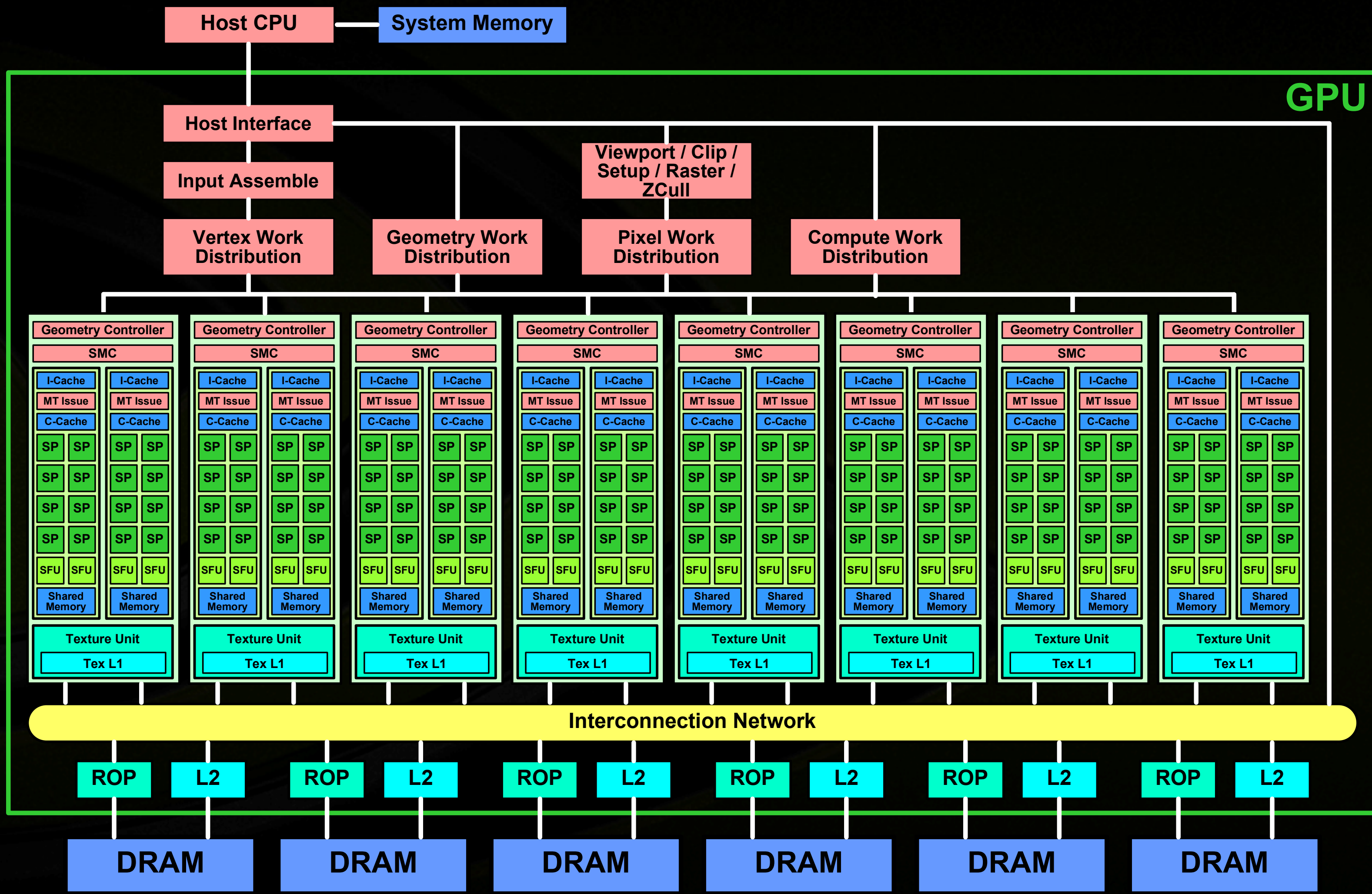
CUFFT Library



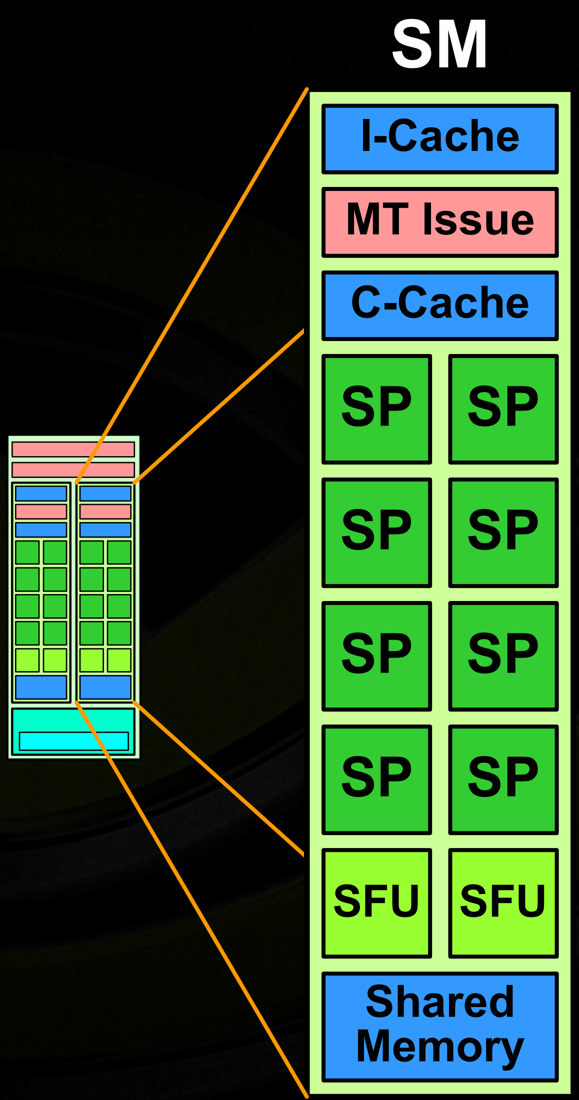
- Efficient FFT on CUDA
- Features
  - 1D, 2D, and 3D FFTs of complex and real-valued signal data
  - Batch execution for multiple 1D transforms in parallel
  - Transform sizes (for 1D) in the range [2, 16M]
  - Transform sizes (for 2D and 3D) in the range [2, 16384]

Tesla Unifies Graphics & Computing

- **Tesla unified computing and graphics architecture**
- **Tesla C870: 128 SP Thread Processor cores at 1.35 GHz**



SM Multithreaded Multiprocessor



- SM has 8 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
  - 32-bit and 64-bit integer
  - 8K 32-bit registers
- SM has 2 SFU Special Function Units
- Scalar ISA
  - Memory load/store, texture fetch
  - Branch, call, return
  - Barrier synchronization instruction
- Multithreaded Instruction Unit
  - 768 Threads, hardware multithreaded
  - 24 SIMT warps of 32 threads
  - Independent thread execution
  - Hardware thread scheduling
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store

SM SIMT Multithreaded Execution

Single-Instruction Multi-Thread instruction scheduler

time

warp 8 instruction 11

warp 1 instruction 42

warp 3 instruction 95

⋮

warp 8 instruction 12

warp 3 instruction 96

- Weaving: first parallel thread technology
- Warp: the set of 32 parallel threads that execute a SIMT instruction
- SIMT: Single-Instruction Multi-Thread

- SM hardware implements zero-overhead warp and thread scheduling
- Each SM executes up to 768 concurrent threads, as 24 SIMT warps of 32 threads

- Threads can execute independently
- SIMT warp diverges and converges when threads branch independently
- Best efficiency and performance when threads of a warp execute together
- SIMT across threads (not just SIMD data) provides easy single-thread scalar programming with SIMD efficiency

Thread Processor Datapath

- Executes 32-bit IEEE floating point instructions:
  - FADD, FMUL, FMAD, FMIN, FMAX, FSET, F2I, I2F
- Performs 32-bit integer instructions:
  - IADD, IMUL24, IMAD24, IMIN, IMAX, ISET, I2I
  - SHR, SHL, AND, OR, XOR
- Fully pipelined
  - Latency and area optimized
- IEEE 754 compliant FADD, FMUL
  - Round to nearest even, round toward zero
  - Handles special numbers, NaNs, infinities properly
  - Flushes denormal operands and results to zero

Special Function Unit (SFU)

- Executes transcendental function instructions
  - RCP, RSQRT, EXP2, LOG2, SIN, COS
  - 2 SFUs per SM yields ¼ instruction throughput
- Evaluates function approximations
  - Quadratic interpolation with Enhanced Minimax Approximation
  - Interpolates pixel attributes
- Accuracy ranges from 22.5 to 24.0 bits
  - 1/x in the interval [1,2) is 24 bits, 1 ulp

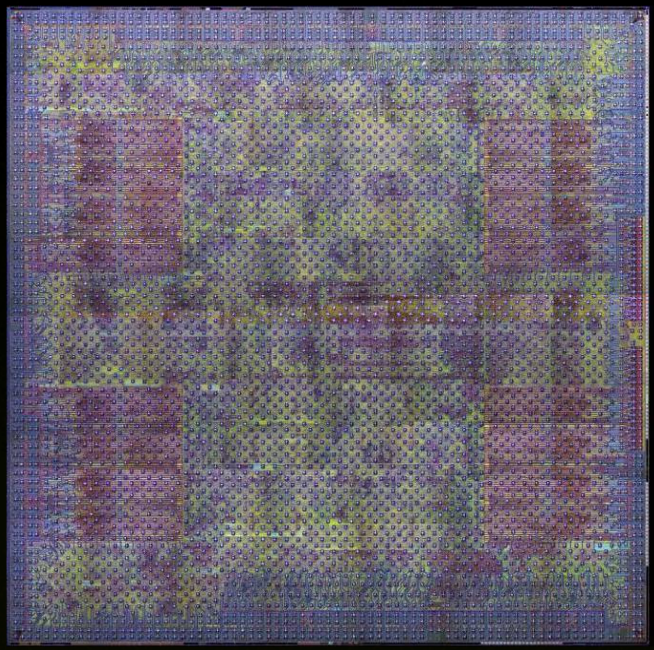
Tesla C870 GPU Implementation

- 681 million transistors
- 470 mm^2 in 90 nm CMOS

- 128 thread processors
- 518 GFLOPS peak
- 1.35 GHz processor clock

- 1.5 GB DRAM
- 76 GB/s peak
- 800 MHz GDDR3 clock
- 384 pin DRAM interface

- ATX form factor card
- PCI Express x16
- 170 W max with DRAM



Summary

- Transition to scalable parallel programming is being led by unified graphics and computing GPUs
- CUDA scalable programming model
  - Provides readily understood abstractions
  - Hierarchy of thread groups, shared memory, synchronization
  - Fine grained and coarse-grained parallelism
  - Productive environment for developing parallel software
  - Great for teaching scalable parallel programming
  - Maps to GPUs today, later to other parallel architectures
- CUDA and ubiquitous parallel GPUs are democratizing parallel programming

//www.nvidia.com/CUDA