NVIDIA®

GPU Computing with CUDA

CScADS Workshop on Automatic Tuning

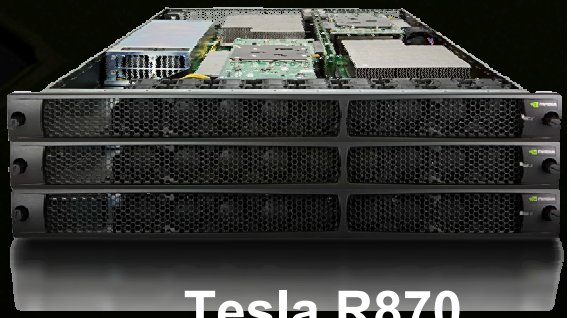
Richard Johnson

Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture is a highly parallel computing platform
- In laptops, desktops, workstations, servers
- 8-series GPUs deliver 50 to 200 GFLOPS on compiled parallel C applications
- GeForce 8800 has 128 processor cores
- Driven by the insatiable demands of PC game market, the number of cores double each year
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism

GeForce 8800

Tesla D870



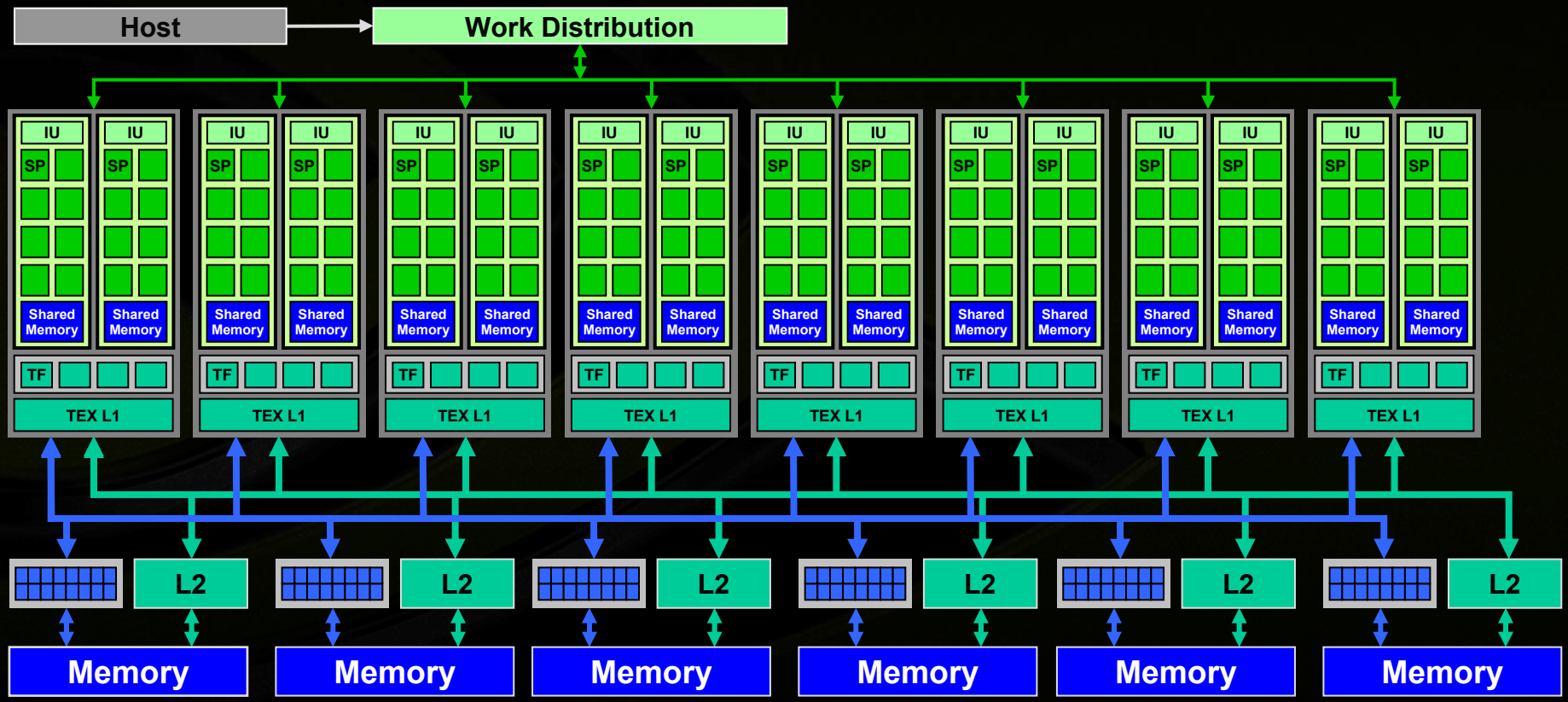
Tesla R870

Outline

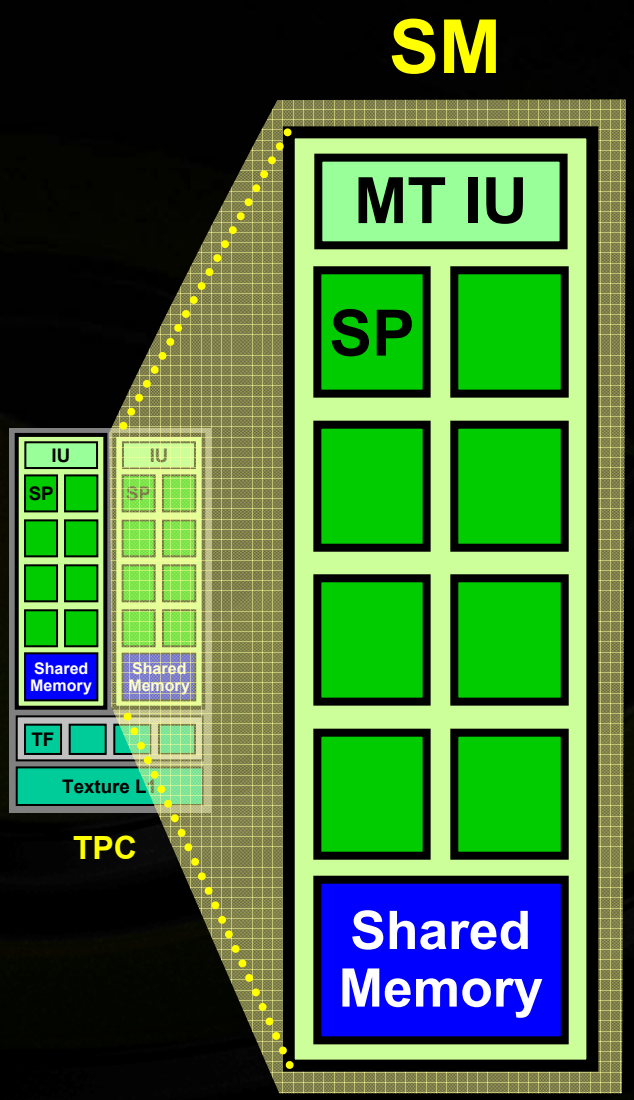
- NVIDIA GPU Computing Architecture
- GPU Computing with CUDA
- Tuning for Performance
- GPU Computing and Cuda: Real-world Experiences

NVIDIA GPU Computing Architecture

- Massively multithreaded parallel computing platform
- 128 SP Stream Processors at 1.35 GHz, 518 GFLOPS peak
- 12,288 concurrent threads, hardware managed
- GPU Computing mode enables C on Graphics Processing Unit



Streaming Multiprocessor



- Each SM has 8 Streaming Processors
  - 32 GFLOPS peak at 1.35 GHz
- Scalar ISA
  - load/store architecture
  - 32-bit integer instructions
  - IEEE 754 32-bit floating point
  - Branch, call, return, predication
  - Barrier synchronization instruction
- 768 Threads, hardware multithreaded
  - 24 SIMD warps of 32 threads
  - Independent MIMD thread execution
  - Hardware thread scheduling
- 8K registers, distributed among threads
- 16KB Shared Memory
  - Concurrent threads share data
  - Very low latency access

How to Scale GPU Computing?

- GPU parallelism scales widely
  - Ranges from 8 to many 100s of cores
  - Ranges from 100 to many 1000s of threads
- Graphics performance scales with GPU parallelism
  - Data parallel mapping of pixels to threads
  - Unlimited demand for parallel pixel shader threads and cores

Challenge:

- Scale Computing performance with GPU parallelism
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

Scalability Solution

- Programmer uses multi-level data parallel decomposition
  - Decomposes problem into a sequence of steps (Grids)
  - Decomposes Grid into independent parallel Blocks (thread blocks)
  - Decomposes Block into cooperating parallel elements (threads)
- GPU hardware distributes thread blocks to available SM cores
  - GPU balances work load across any number of SM cores
  - SM core executes program that computes Block
- Each thread block computes independently of others
  - Enables parallel computing of Blocks of a Grid
  - No communication among Blocks of same Grid
  - Scales one program across any number of parallel SM cores
- Programmer writes one program for all GPU sizes
- Program does not know how many cores it uses
- Program executes on GPU with any number of cores

Outline

- NVIDIA GPU Computing Architecture
- **GPU Computing with CUDA**
- Tuning Performance

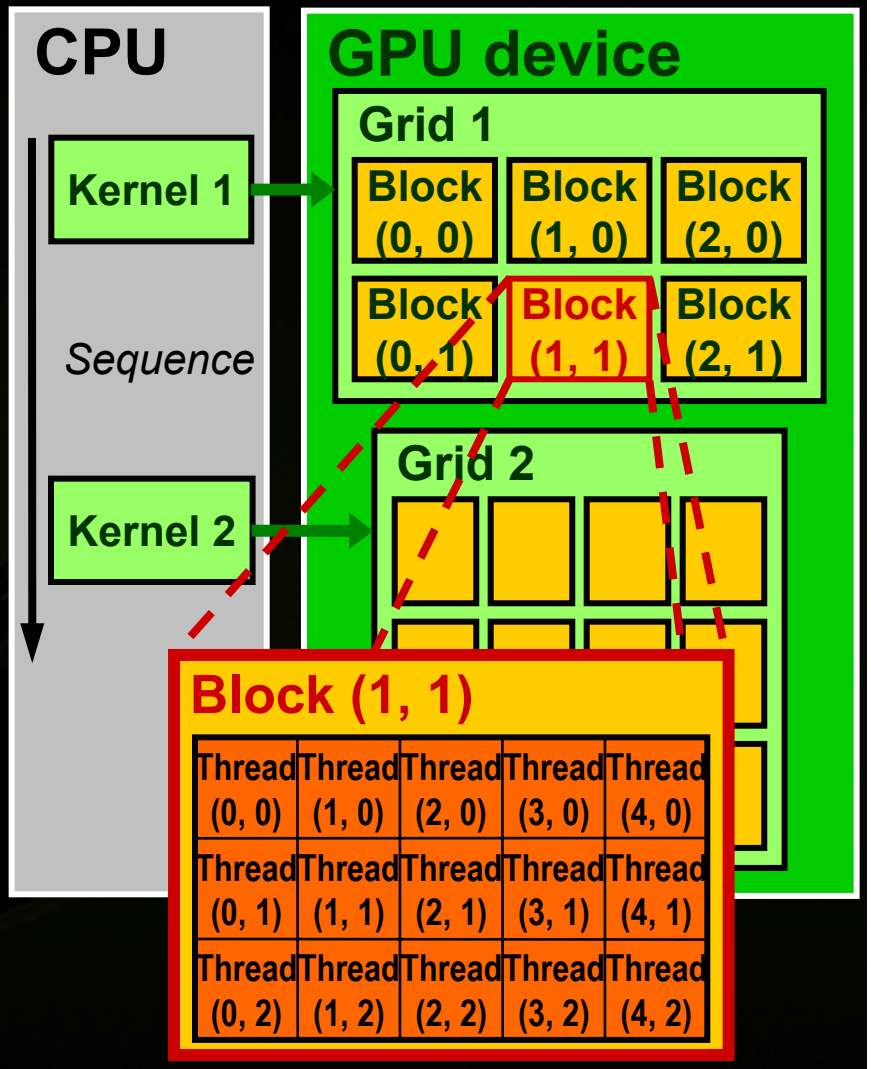
GPU Computing and Cuda: Real-world Experiences

CUDA Programming Model: Parallel Multithreaded Kernels

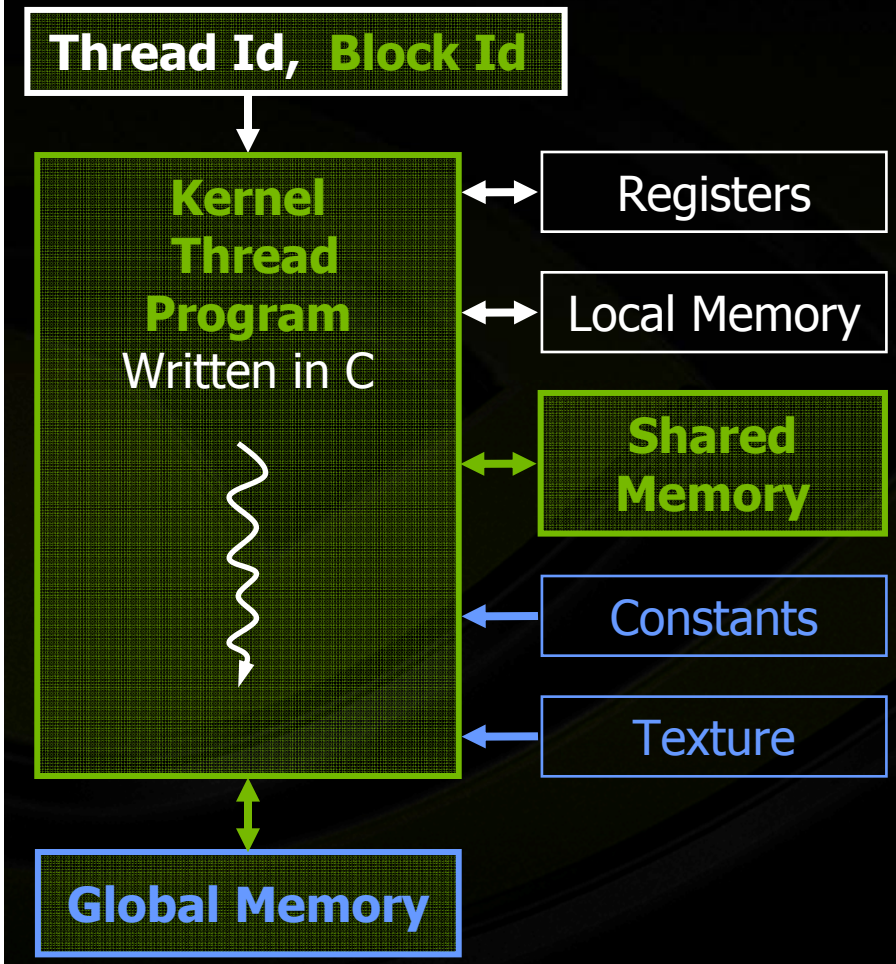
- View GPU as a computing device that:
  - Acts as a coprocessor to the CPU host
  - Has its own memory hierarchy
  - Runs many lightweight threads in parallel
- Integrated CPU + GPU application C program
  - Partitions problem into a sequence of kernels
  - Kernel C code executes on GPU
  - Sequential C code executes on CPU
- Kernels execute in parallel using multiple levels of parallelism

CUDA Terminology: Grids, Blocks, and Threads

- Programmer partitions problem into a sequence of kernels.
- A kernel executes as a grid of thread blocks
- A thread block is an array of threads that can cooperate
- Threads within the same block synchronize and share data in Shared Memory
- Execute thread blocks on multithreaded multiprocessor SM cores



CUDA Programming Model: Thread Memory Spaces



- Each kernel thread can read:
  - Thread Id — per thread
  - Block Id — per block
  - Constants — per grid
  - Texture — per grid
- Each thread can read and write:
  - Registers — per thread
  - Local memory — per thread
  - Shared memory — per block
  - Global memory — per grid
- Host CPU can read and write:
  - Constants — per grid
  - Texture — per grid
  - Global memory — per grid

CUDA: C on the GPU

- Single-Program Multiple-Data programming model
  - C program for a thread of a thread block in a grid
  - Extend C only where necessary
  - Simple, explicit language mapping to parallel threads

- Declare C kernel functions and variables on GPU:

```
__global__ void KernelFunc(...);
__device__ int  GlobalVar;
__shared__ int  SharedVar;
```

- Call kernel function as Grid of 500 blocks with 128 threads per block:

```
KernelFunc<<< 500, 128 >>>(...);
```

- Explicit GPU memory allocation, CPU-GPU memory transfers
  - `cudaMalloc()`, `cudaFree()`
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...

CUDA C Example: Add Arrays

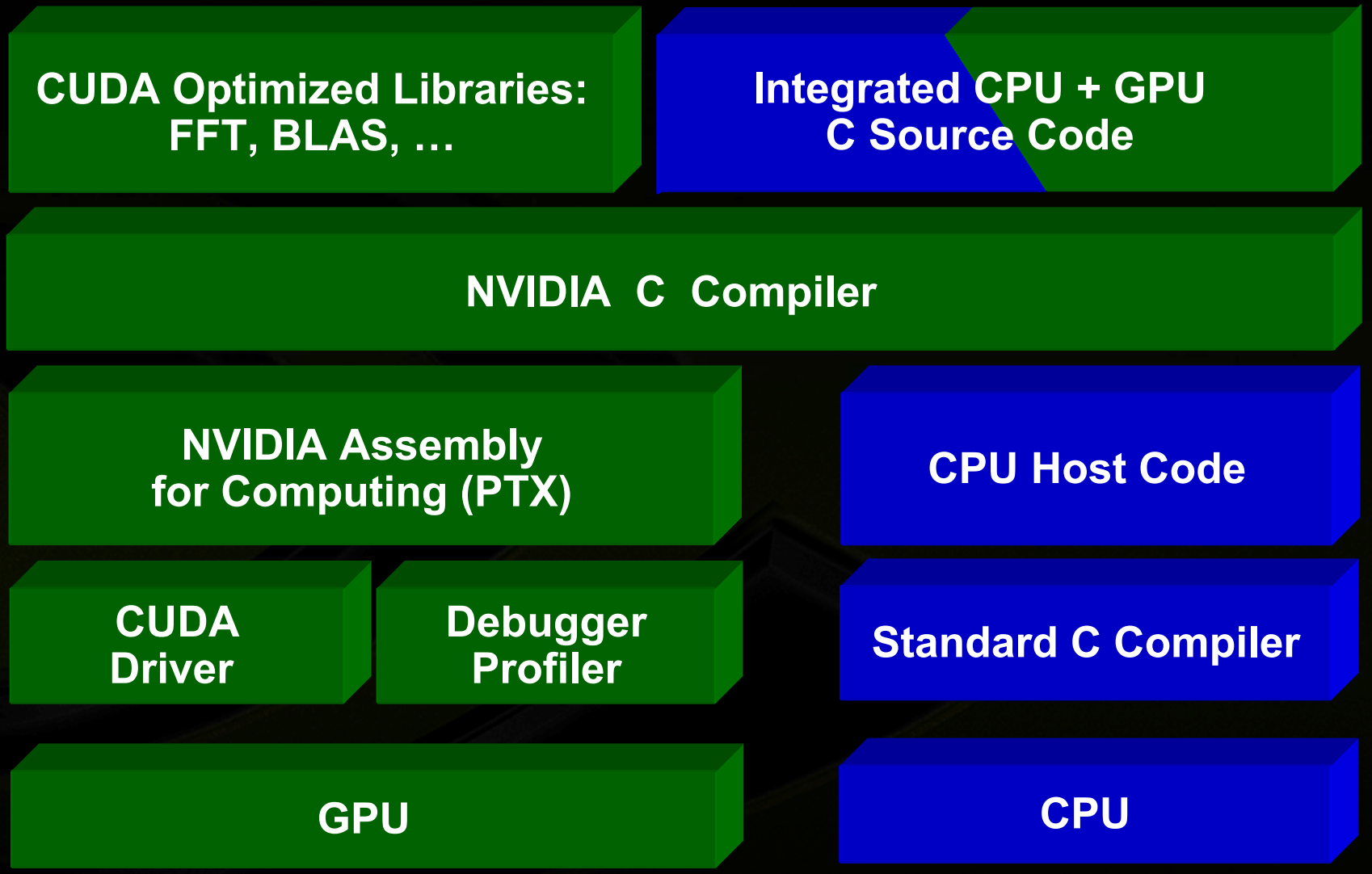
C program

```
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);
}
```

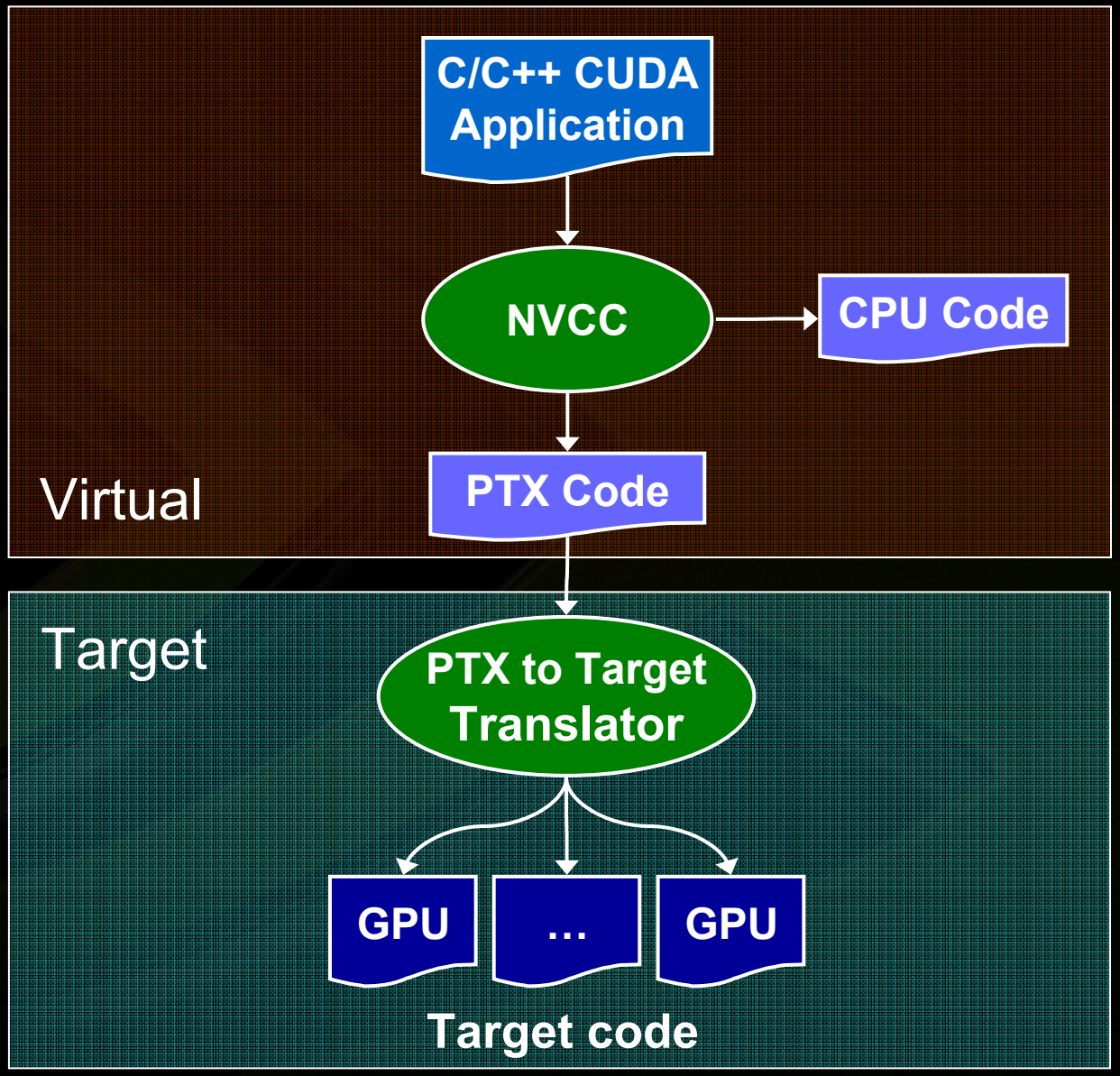
CUDA C program

```
__global__ void addMatrixG
      (float *a, float *b, float *c, int N)
{
      int i = blockIdx.x * dimBlock.x + threadIdx.x;
      int j = blockIdx.y * dimBlock.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 Software Development Kit

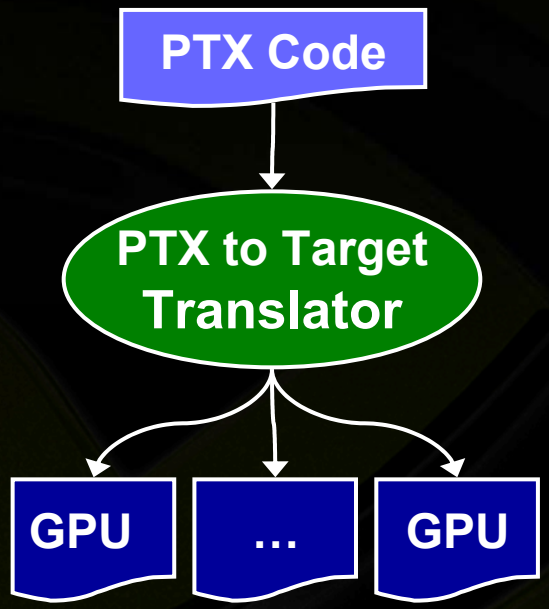


Compiling CUDA



Virtual to Target ISA Translation

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7},[$r9+0];
mad.f32           $f1,$f5,$f3,$f1;
```



Target code

```
0x103c8009 0x0fffffff
0xd00e0609 0xa0c00780
0x100c8009 0x00000003
0x21000409 0x07800780
```

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Distribution format for applications
  - Install-time translation
  - “fat binary” caches target-specific versions
- Target-specific translation optimizes for:
  - ISA diffences
  - Resource allocation
  - Performance

Outline

- NVIDIA GPU Computing Architecture
- GPU Computing with CUDA
- **Tuning for Performance**
- GPU Computing and Cuda: Real-world Experiences

Typical Programming Paradigms

- Load; Process; Store; Repeat
  - Thread block reads global data into shared memory
  - Threads compute using shared memory
  - Threads store results in global memory
  - Repeat

  Big impact when data is used multiple times

- Within a grid, blocks execute independently
  - Enables manycore scalability

Balancing Resources

- Fully utilizing each SM requires balancing and coordinating resources shared among the SM's threads.
  - Limit on number of threads per SM
  - Limit on number of thread blocks per SM
  - Shared memory
  - Shared register file
  - Coordinating memory accesses to minimize bank conflicts
- Parameterize application to allow exploration of trade-offs
  - Thread block size – degree of thread parallelism
  - Data tile size
  - Number of results computed by each thread
  - Degree of unrolling
  - Prefetching
  - ...

Manual Tuning of Matrix Multiply

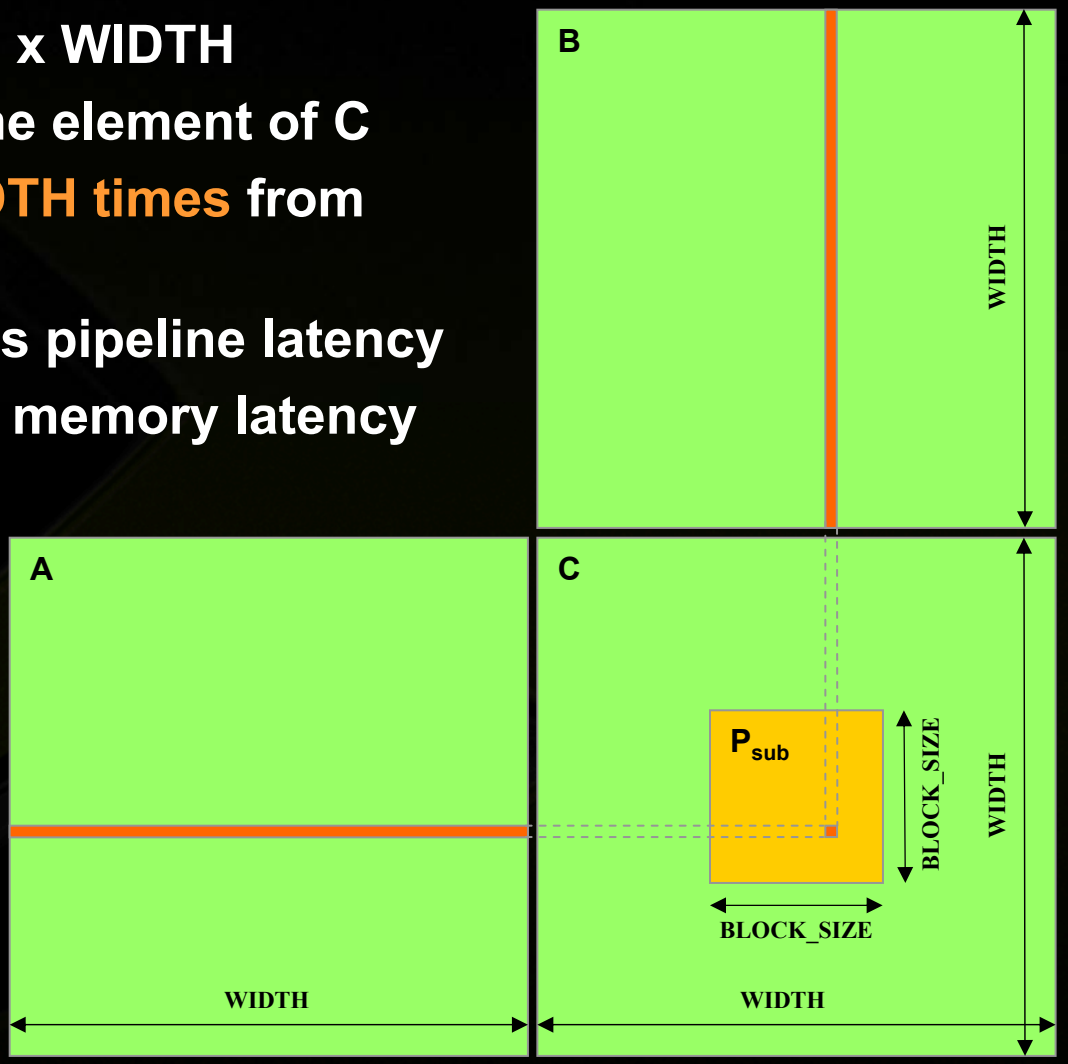
Multiply two 4096 x 4096 element matrices

Let's consider several versions of the kernel…

- Naïve implementation
  - each thread computes one result, no collaboration
- Tiled implementation
  - Increase compute-to-load ratio
  - Reuse data in shared memory
- Tiled and unrolled implementation
  - Further increases compute-to-load ratio
- Expanded tiled implementation
  - Each thread computes multiple result values
  - Data tile 4x larger than thread array
  - Further increases compute-to-load ratio, decreases SM utilization

Naïve Implementation

- C = A * B of size WIDTH x WIDTH
- Each thread handles one element of C
- A and B are loaded WIDTH times from global memory
- Thread parallelism hides pipeline latency
- Block parallelism hides memory latency



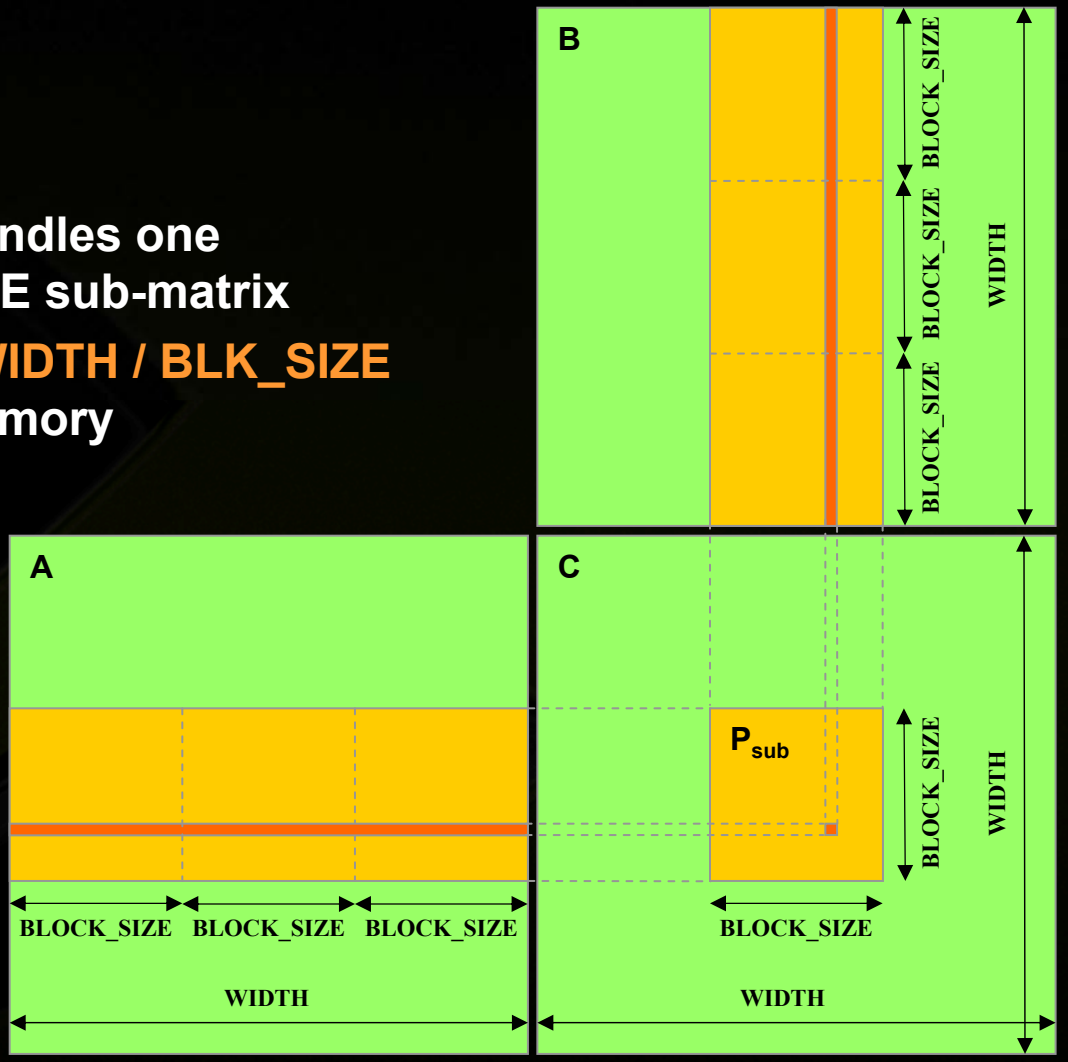
Naïve Kernel


```
Tx = threadIdx.x;    Ty = threadIdx.y;
Bx = blockIdx.x;     By = blockIdx.y;

X = Bx * BLOCK_SIZE + Tx;
Y = By * BLOCK_SIZE + Ty;

idxA = Y * WIDTH;    idxB = X;
idxC = Y * WIDTH + X;

Csub = 0.0;
for (i=0; I < WIDTH; i++) {
  Csub += A[idxA] * B[idxB];
  idxA += 1;
  idxB += WIDTH;
}
C[idxC] = Csub;
```

- Inner loop:
  - ld; ld; fmad; add; add; add; cmp; bra;
  - 1/4 are loads, 1/8 are fmad
- Compute-to-memory ratio is 1:1
- Required bandwidth: 172.8 GB/s
  128 * 1.35GHz * 4 bytes/ld * ¼ instr
- Peak available bandwidth: 86.4 GB/s
  - only half of required bandwidth
- Measured perf: up to 17.2 GFLOPS
- Performance limited by available memory bandwidth

Tiled Implementation

- C = A * B
- With tiling:
  - Each thread block handles one BLK_SIZE x BLK_SIZE sub-matrix
  - A and B are loaded WIDTH / BLK_SIZE times from global memory
- Substantial reduction in memory traffic



Tiled Kernel

```
__shared__ float As[16][16];
__shared__ float Bs[16][16];

Csub = 0.0;
for (…) {  // iterate across tiles
   As[ty][tx] = A[idxA];
   Bs[ty][tx] = B[idxB];
   idxA += 16;  idxB += 16 * WIDTH;
   __syncthreads();

   for (i=0; i < 16; i++) {
      Csub += As[ty][i] * Bs[i][tx];
   }
   __syncthreads();
}
C[idxC] = Csub;
```

- 16x16 block allows 3 blocks/SM
  - hides sync latency
  - 16 regs/thread limits us to 2 blocks/SM
- Each threads loads 2 values from global mem, uses 32 values from shared mem
- Measured perf: 47.5 GFLOPS
- Performance limited by inner-loop overhead

Tiled and Unrolled Kernel


```
Csub = 0.0;
for (…) {  // iterate across blocks
   As[ty][tx] = A[idxA];
   Bs[ty][tx] = B[idxB];
   idxA += 16;  idxB += 16 * WIDTH;
   __syncthreads();

   // completely unroll inner loop
   Csub += As[  0][i] * Bs[i][  0];
   Csub += As[  1][i] * Bs[i][  1];
   …
   Csub += As[15][i] * Bs[i][15];
   }
   __syncthreads();
}
C[idxC] = Csub;
```

- Inner loop completely unrolled
  - Eliminates loop overhead
  - Address arithmetic optimized
- Measured perf: 85.6 GFLOPS
- Performance again limited by compute-to-memory ratio
- Idea: increase data tile to 32x32
  - Plenty of shared memory
  - Each thread computes four values

Expanded Tiled Kernel

- 16x16 thread block, with 32x32 data tile
  - Each thread computes four result values
- Maximizes compute-to-memory ratio
  - Each thread block uses 4KB of shared memory
  - Increased register usage limits us to one block/SM
- Unrolling experiments:
  - no unrolling: 70.7 GFLOPS
  - Fully unrolled: insufficient registers
  - Unroll-by-2: 86.1 GFLOPS
  - Unroll-by-4: 118.4 GFLOPS

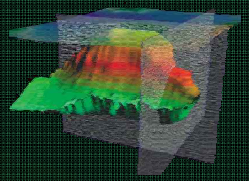
Autotuning Opportunities

- Balancing shared resources to maximize performance
  - Requires control over compiler optimizations, program parameters
  - New GPU applications typically go through multiple revisions to achieve best performance; intuition not always a reliable guide
- Blocking in massively parallel applications
  - Discovering best tile size for shared memory data reuse
  - Discovering best degree of thread-level parallelism
  - Discovering best number of results to compute per thread
- GPU Onload
  - Discovering portions of CPU code to “onload” to GPU
- Tuning application performance for new generations of GPUs
  - Current tools allow programs to scale to future GPUs
  - But maximum performance requires re-tuning

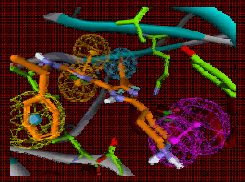
Outline

- NVIDIA GPU Computing Architecture
- GPU Computing with CUDA
- Tuning for Performance
- GPU Computing and Cuda: Real-world Experiences

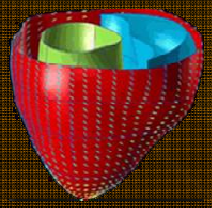
GPU Computing Application Areas



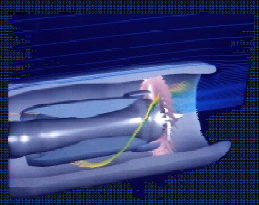
Computational Geoscience



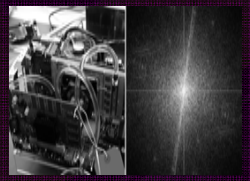
Computational Chemistry



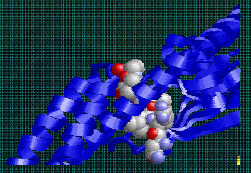
Computational Medicine



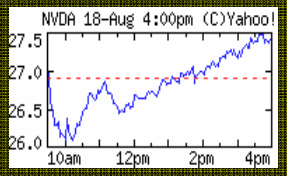
Computational Modeling



Computational Science



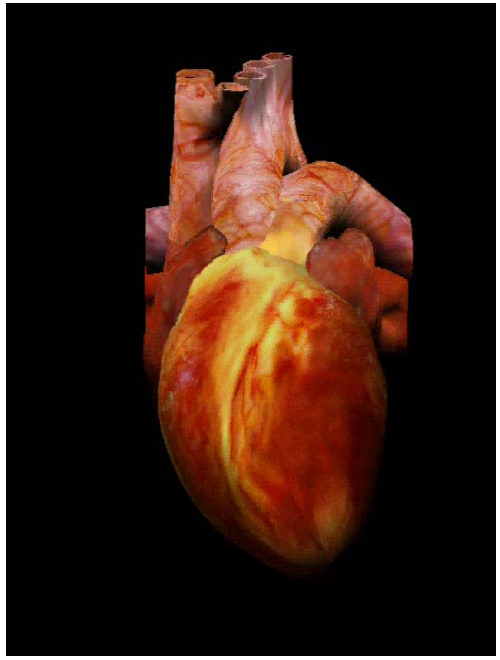
Computational Biology



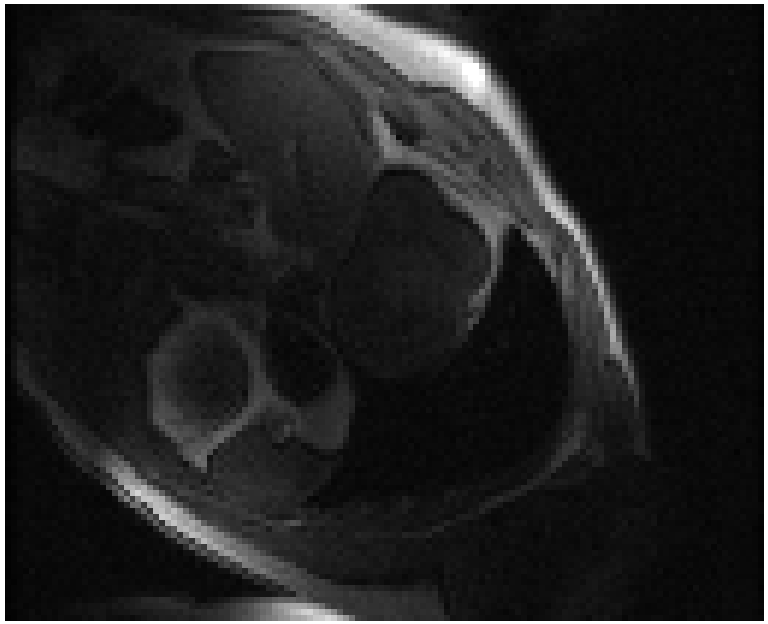
Computational Finance

Image Processing

Dynamic Real-Time MRI



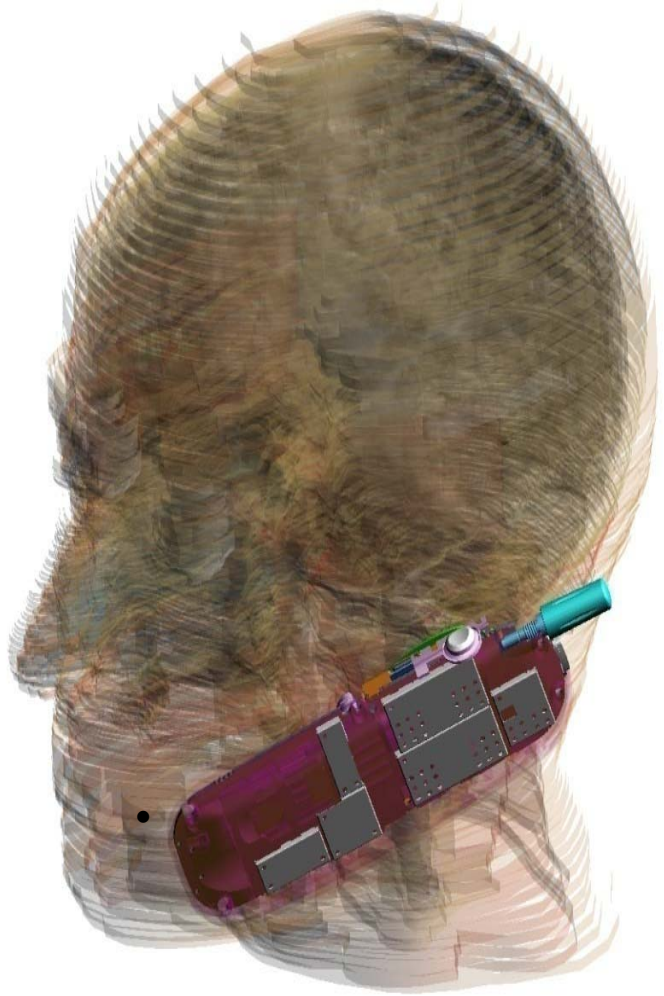
Bioengineering Institute, University of Auckland,
IUPS Physiome Project
http://www.bioeng.auckland.ac.nz/movies/database/
cardiovascular_system/textured-heart-beat.mpg



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

G80 GPU is 245x CPU

EM: Cell Phone Simulation



Agilent Technologies

MOTOROLA

NOKIA
Connecting People

LG

Sony Ericsson

FDA

SAMSUNG

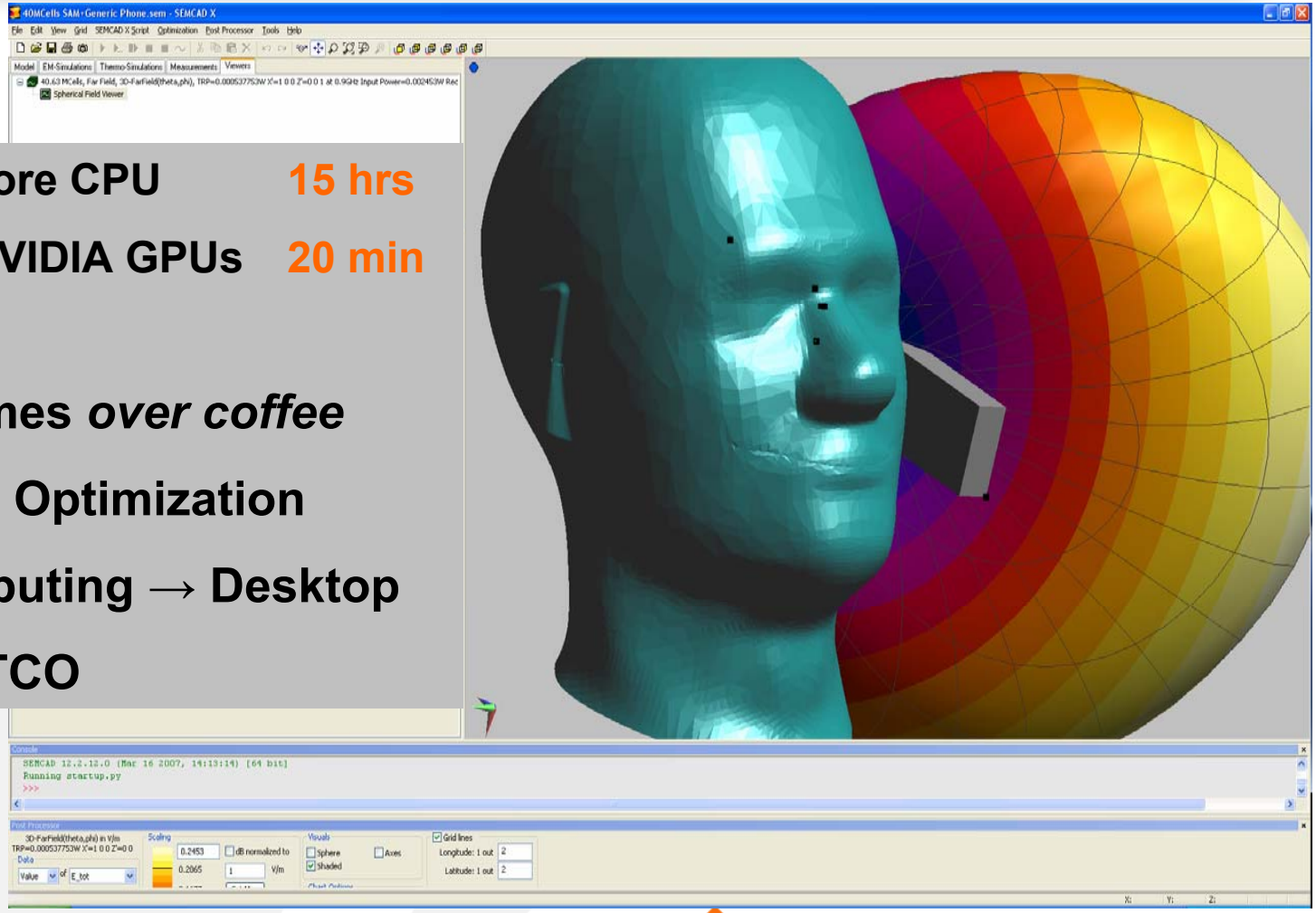
Ryan Schneider, CTO
acceleware
PROCESSING SUPERPOWER

The “Race”

HP xw9400 with 4 core CPU **15 hrs**

HP xw9400 with 4 NVIDIA GPUs **20 min**

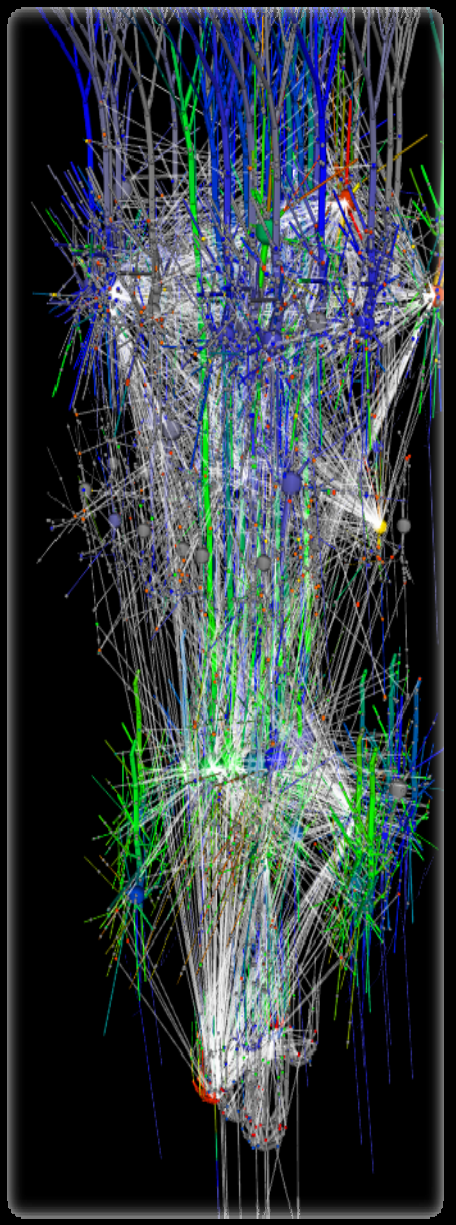
- ***Overnight*** **becomes** ***over coffee***
- **Computer-Aided Optimization**
- **45X: Supercomputing → Desktop**
- **More, for same TCO**



acceleware
PROCESSING SUPERPOWER

Evolved machines

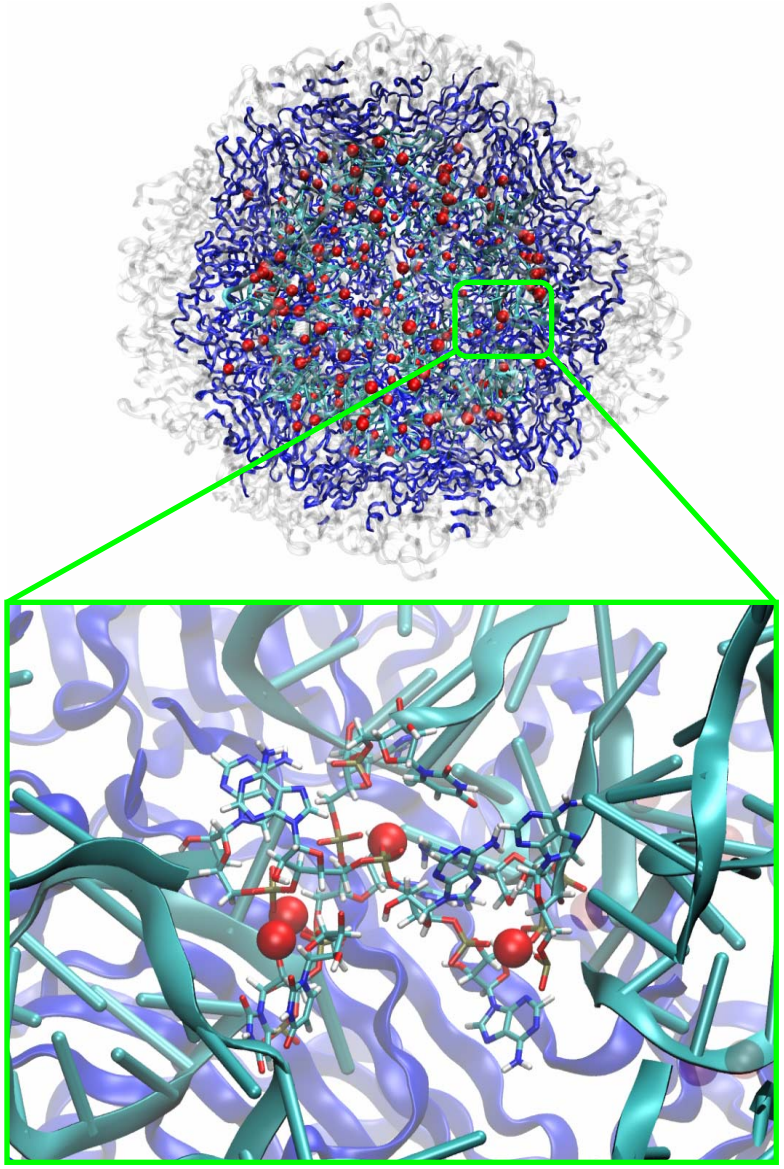
Paul Rhodes - CEO



- Computational Neuroscience Simulation
- We already achieve 130x acceleration over current x86 with 1 board of 2 NVIDIA G80 GPUs
- > 1 Tflop / board
- >12 Tflops / rack (4u enclosure)
- 1 rack may soon exceed the Top500 number 19 (which has 4,096 cores)

Preparing Virus for Molecular Simulation

- Key task: placement of ions inside and around the virus
- 110 CPU-hours on SGI Altix Itanium2
- Larger viruses could require thousands of CPU-hours
- 27 GPU-minutes on G80 GPU
- Over 240 times faster - ion placement can now be done on a desktop machine!



John Stone
Beckman Institute, University of Illinois

Summary

- GPU architectures have evolved to be well-suited for solving data-parallel problems, and they continue to deliver increasingly higher performance
- As a simple extension to the C programming language, CUDA provides easy access to high performance GPU computing
- CUDA and the 8-series GPUs expose a rich environment for automatic tuning of application performance
  - Placement: GPU vs. CPU
  - Hierarchy: memory and processor
  - Blocking: selecting threads per thread block, results per thread
  - Balancing shared resources
  - Target-specific re-tuning for maximum performance