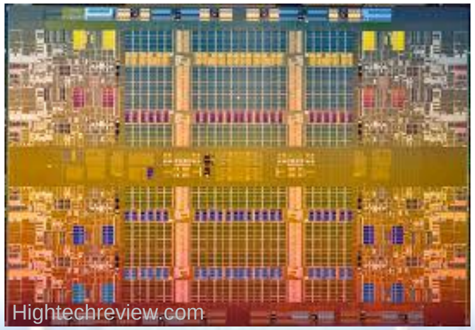
Writing Efficient CUDA Programs

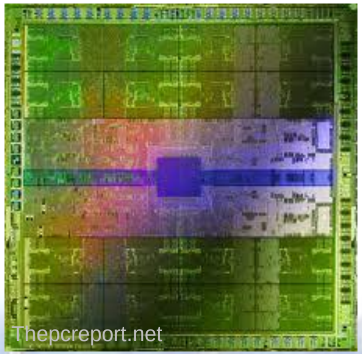
Martin Burtscher
Department of Computer Science
TEXAS STATE UNIVERSITY
SAN MARCOS
The rising STAR *of Texas*

High-End CPUs and GPUs

	Xeon X7550	Tesla C2050
Cores	8 (superscalar)	448 (simple)
Active threads	2 per core	48 per core
Frequency	2 GHz	1.15 GHz
Peak performance*	128 GFlop/s	1030 GFlop/s
Peak mem bandwidth	25.6 GB/s	144 GB/s
Maximum power	130 W	238 W
Price	$2800	$2300

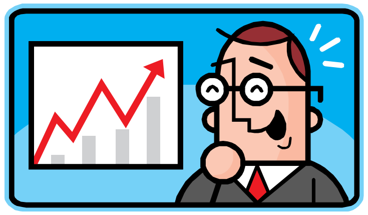
Tesla: late 2009
Xeon: early 2010





GPU Advantages

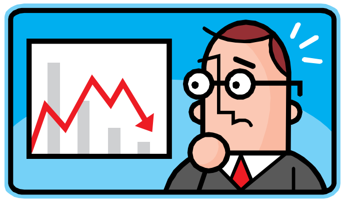
- Performance
  - 8x as many instructions executed per second
- Main memory bandwidth
  - 5.6x as many bytes transferred per second
- Cost-, energy-, and size-efficiency
  - 9.8x as much performance per dollar
  - 4.4x as much performance per watt
  - 10.4x as much performance per area



(Based on peak values)

GPU Disadvantages

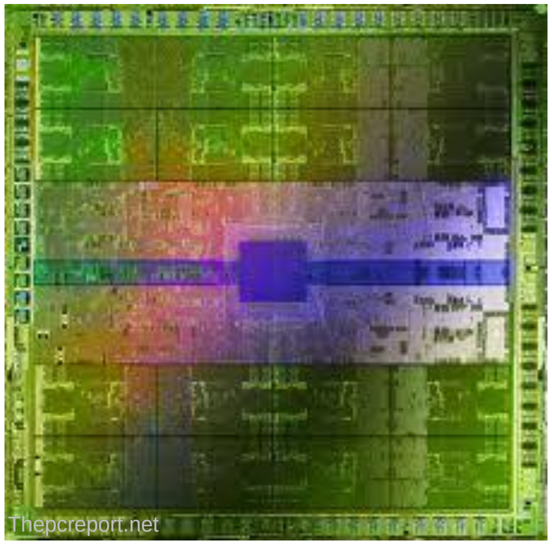
- Clearly, we should be using GPUs all the time
  - So why aren't we?
- GPUs can only execute some types of code fast
  - Need lots of data parallelism, data reuse, regularity
- GPUs are harder to program and tune than CPUs
  - In part because of poor tool (compiler) support
  - In part because of their architecture



- Requirements and arch are unlikely to change

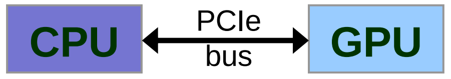
Outline

- Introduction
- CUDA overview
- N-body example
- Porting and tuning
- Other considerations
- Conclusions



CUDA Programming

- General-purpose (non-graphics) programming
  - Uses GPU as massively parallel co-processor



- SIMT (single-instruction multiple-threads)
  - Thousands of threads needed for full efficiency

- C/C++ with extensions
  - Function launch
    - Calling functions on GPU
  - Memory management
    - GPU memory allocation, copying data to/from GPU
  - Declaration qualifiers
    - Device, shared, local, etc.
  - Special instructions
    - Barriers, fences, max, etc.
  - Keywords
    - threadIdx, blockIdx

Calling GPU Kernels

- Kernels are functions that run on the GPU
  - Callable by CPU code
  - CPU can continue processing while GPU runs kernel

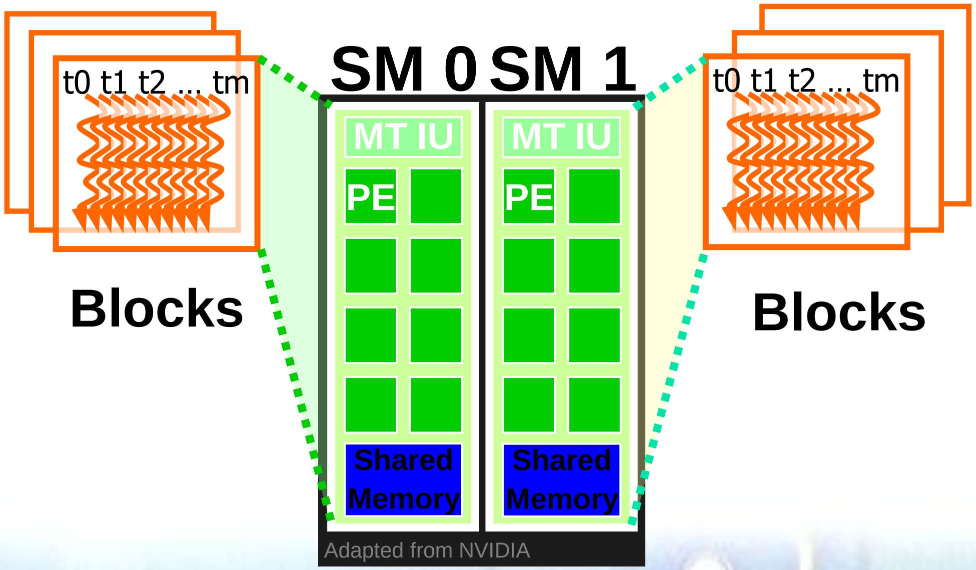
```
KernelName<<<blocks, threads>>>(arg1, arg2, ...);
```

- Launch configuration (programmer selectable)
  - Special parameters: number of blocks and threads
    - Kernel call automatically spawns m blocks with n threads (i.e., m*n threads total) that run a copy of the same function
  - Normal function parameters: passed conventionally
    - Different address space, should never pass CPU pointers

Block and Thread Allocation

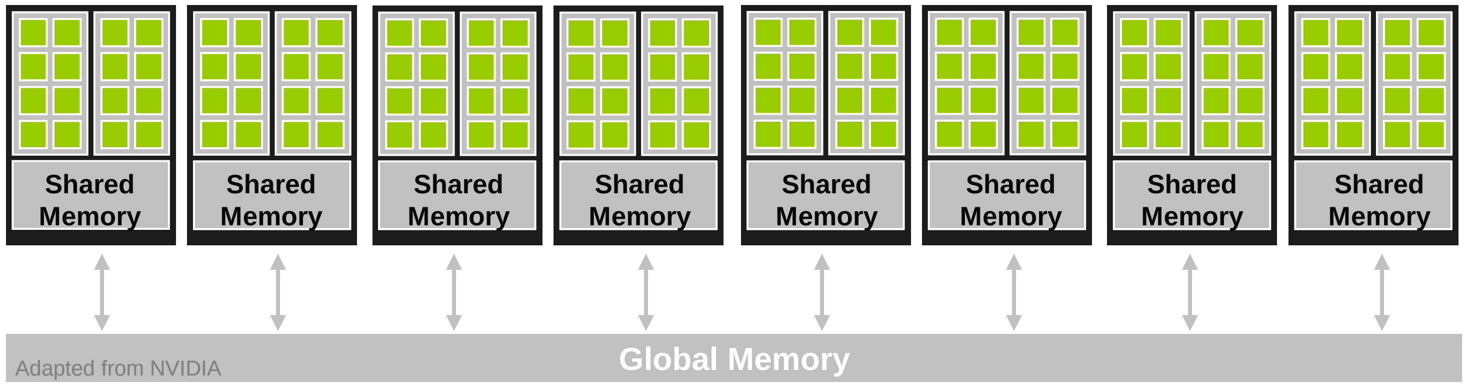
- Blocks assigned to SMs
  - Streaming multiprocessors
- Threads assigned to PEs
  - Processing elements

- Hardware limits
  - 8 resident blocks per SM
  - 768, 1024, or 1536 resident threads per SM
  - 512, 512, or 1024 threads per block
  - Above limits are lower if register or shared mem usage is too high
  - 65535 blocks per kernel



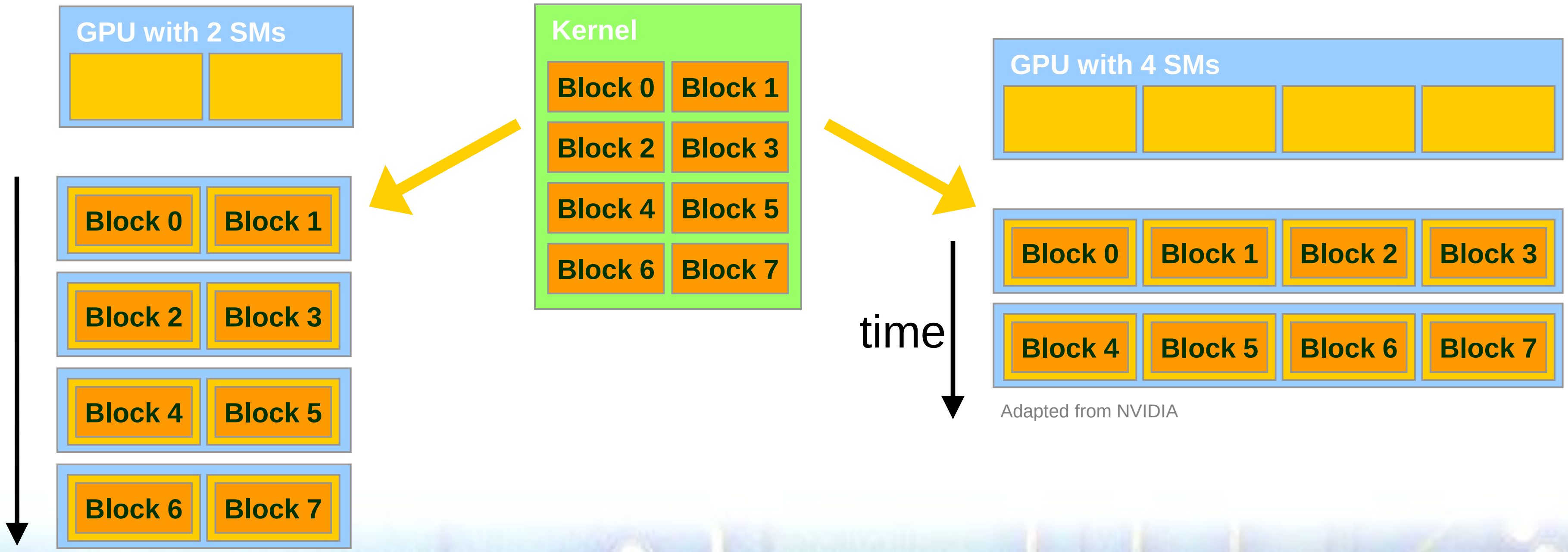
GPU Architecture

- 1 to 30 SMs (with 8, 8, or 32 PEs per SM)
- SMs have fast barriers, thread voting, shared mem
  - Very fast thread communication within block
  - Slow communication between blocks (DRAM atomics)



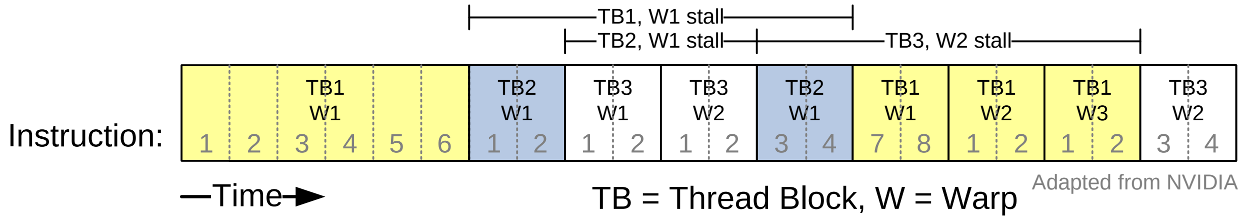
Block Scalability

- Hardware can assign blocks to SMs in any order
  - A kernel with enough blocks scales across GPUs
  - Not all blocks may be resident at the same time



Warp-Based Execution

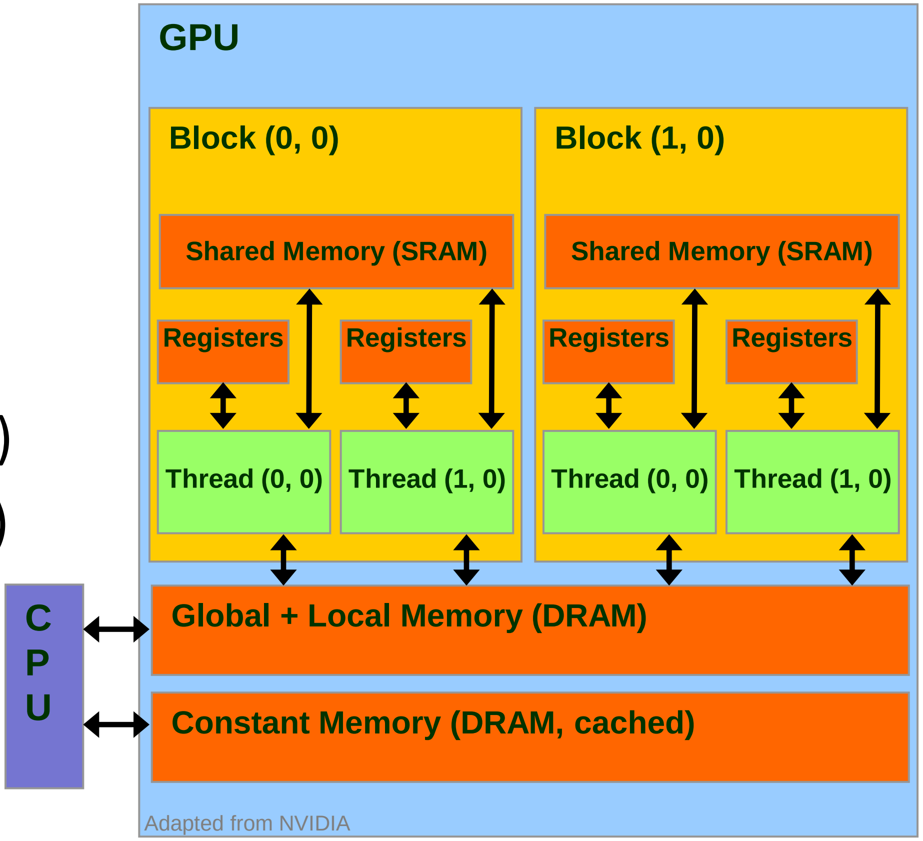
- 32 contiguous threads form a *warp*
  - Execute same instruction in same cycle (or disabled)
  - At any time, only one warp is executed per SM
  - Warps are scheduled out-of-order w.r.t. each other



- Thread divergence (reduction of parallelism)
  - Some threads in warp jump to different PC than others
  - Hardware runs subsets of warp until they re-converge

GPU Memories

- Memory types
  - Registers (r/w per thread)
  - Local mem (r/w per thread)
  - Shared mem (r/w per block)
    - Software-controlled cache
  - Global mem (r/w per kernel)
  - Constant mem (r per kernel)
- Separate from CPU
  - CPU can access global and constant mem via PCIe bus
  - Requires explicit transfer

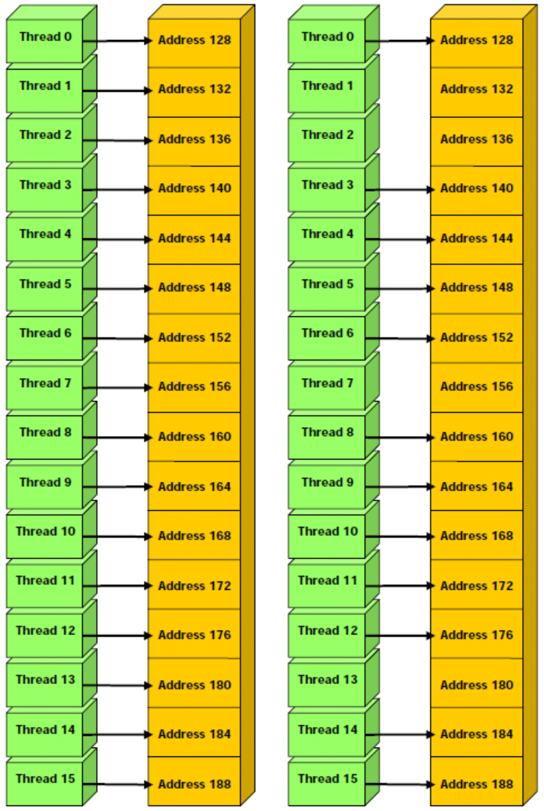


Fast Memory Accesses

- Coalesced main memory access (16/32x faster)
  - Under some conditions, HW combines multiple (half) warp memory accesses into a single coalesced access
    - CC 1.1: 64-byte aligned contiguous 4-byte words
    - CC 1.3: 64-byte aligned 64-byte line (any permutation)
    - CC 2.0: 128-byte aligned 128-byte line (cached)
- Bank-conflict-free shared memory access (16/32)
  - No superword alignment or contiguity requirements
    - CC 1.x: 16 different banks per half warp or same word
    - CC 2.0: 32 different banks + one-word broadcast

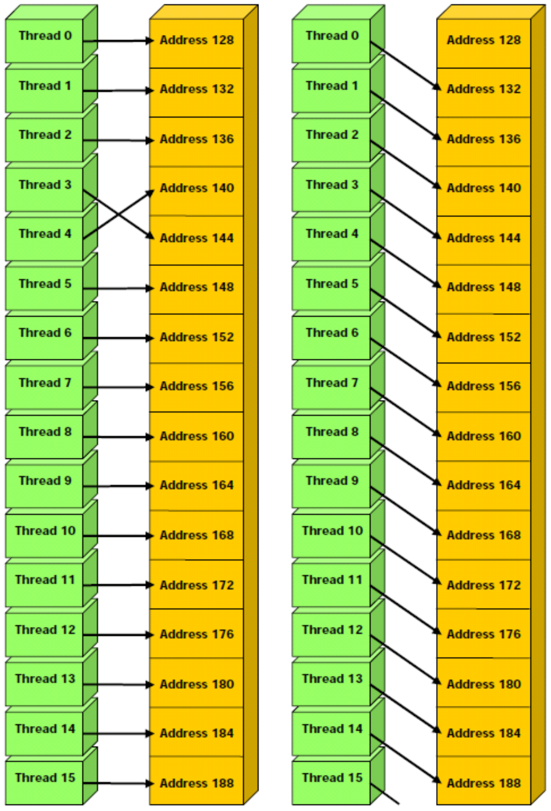
Coalesced Main Memory Accesses

single coalesced access



NVIDIA

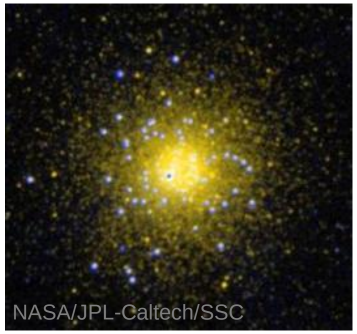
one and two coalesced accesses*



NVIDIA

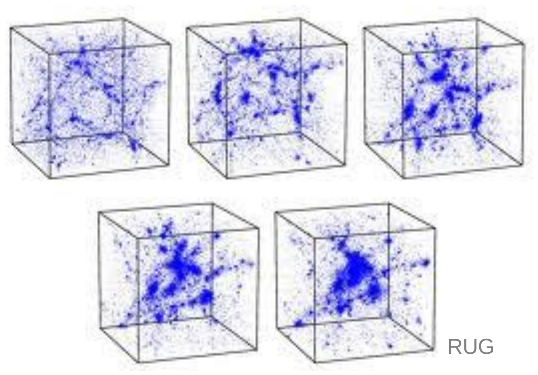
Outline

- Introduction
- CUDA overview
- N-body example
- Porting and tuning
- Other considerations
- Conclusions

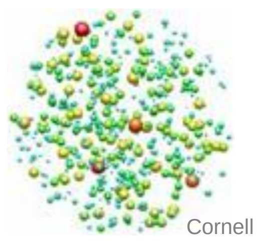


N-Body Simulation

- Time evolution of physical system
  - System consists of bodies
  - "n" is the number of bodies
  - Bodies interact via pair-wise forces



- Many systems can be modeled in this way
  - Star/galaxy clusters (gravitational force)
  - Particles (electric force, magnetic force)



Simple N-Body Algorithm

- Algorithm

```
Initialize body masses, positions, and velocities
Iterate over time steps {
    Accumulate forces acting on each body
    Update body positions and velocities based on force
}
Output result
```

- More sophisticated n-body algorithms exist
  - Barnes Hut algorithm
  - Fast Multipole Method (FMM)

Key Loops (Pseudo Code)

```
bodySet = // input
for timestep do {  // O($n^2$) sequential
  foreach Body b1 in bodySet {  // O($n^2$) parallel
    foreach Body b2 in bodySet {
      if (b1 != b2) {
        b1.addInteractionForce(b2);
      }
    }
  }
  foreach Body b in bodySet {   // O($n$) parallel
    b.Advance();
  }
}
// output result
```

Force Calculation C Code

```
struct Body {
  float mass, posx, posy, posz; // mass and 3D position
  float velx, vely, velz, accx, accy, accz; // 3D velocity & accel
} *body;

for (i = 0; i < nbodies; i++) {
  . . .
  for (j = 0; j < nbodies; j++) {
    if (i != j) {
      dx = body[j].posx - px; // delta x
      dy = body[j].posy - py; // delta y
      dz = body[j].posz - pz; // delta z
      dsq = dx*dx + dy*dy + dz*dz; // distance squared
      dinv = 1.0f / sqrtf(dsq + epssq); // inverse distance
      scale = body[j].mass * dinv * dinv * dinv; // scaled force
      ax += dx * scale; // accumulate x contribution of accel
      ay += dy * scale;  az += dz * scale; // ditto for y and z
    }
  }
  . . .
}
```

Outline

- Introduction
- CUDA overview
- N-body example
- Porting and tuning
- Other considerations
- Conclusions

GPU Suitability of N-Body Algorithm

- Lots of data parallelism
  - Force calculations are independent
  - Should be able to keep SMs and PEs busy
- Sufficient memory access regularity
  - All force calculations access body data in same order*
  - Should have lots of coalesced memory accesses
- Sufficient code regularity
  - All force calculations are identical*
  - There should be little thread divergence
- Plenty of data reuse
  - $O(n^2)$ operations on $O(n)$ data
  - CPU/GPU transfer time is insignificant

C to CUDA Conversion

- Two CUDA kernels
  - Force calculation
  - Advance position and velocity
- Benefits
  - Force calculation requires over 99.9% of runtime
    - Primary target for acceleration
  - Advancing kernel unimportant to runtime
    - But allows to keep data on GPU during entire simulation
    - Minimizes GPU/CPU transfers

C to CUDA Conversion

```
__global__ void ForceCalcKernel(int nbodies, struct Body *body, ...) {
  . . .
}
__global__ void AdvancingKernel(int nbodies, struct Body *body, ...) {
  . . .
}

int main(...) {
  Body *body, *bodyl;
  . . .
  cudaMalloc((void**)&bodyl, sizeof(Body)*nbodies);
  cudaMemcpy(bodyl, body, sizeof(Body)*nbodies, cuda…HostToDevice);
  for (timestep = ...) {
    ForceCalcKernel<<<1, 1>>>(nbodies, bodyl, ...);
    AdvancingKernel<<<1, 1>>>(nbodies, bodyl, ...);
  }
  cudaMemcpy(body, bodyl, sizeof(Body)*nbodies, cuda…DeviceToHost);
  cudaFree(bodyl);
  . . .
}
```

Indicates GPU kernel that CPU can call

Separate address spaces, need two pointers

Allocate memory on GPU

Copy CPU data to GPU

Call GPU kernel with 1 block and 1 thread per block

Copy GPU data back to CPU

Evaluation Methodology

- Systems and compilers
  - CC 1.1: Quadro NVS 135M, nvcc 2.2
    - 1 SM, 8 PEs, 0.8 GHz, 768 resident threads
  - CC 1.3: Quadro FX 5800, nvcc 3.2
    - 30 SMs, 240 PEs, 1.3GHz, 30720 resident threads
  - CC 2.0: Tesla C2050, nvcc 3.2
    - 14 SMs, 448 PEs, 1.15 GHz, 21504 resident threads
- Inputs and metric
  - 1k, 10k, or 100k star clusters (Plummer model)
  - Median runtime of three experiments, excluding I/O

1-Thread Performance

- Problem size
  - n=1000, step=1
  - n=10000, step=1
  - n=10000, step=1
- Slowdown rel. to CPU
  - CC 1.1: 39.3
  - CC 1.3: 72.4
  - CC 2.0: 36.7

  (Note: comparing different GPUs to different CPUs)
- Performance
  - 1 thread is one to two orders of magnitude slower on GPU than CPU
- Reasons
  - No caches (CC 1.x)
  - Not superscalar
  - Slower clock frequency
  - No SMT latency hiding

Using N Threads

- Approach
  - Eliminate outer loop
  - Instantiate n copies of inner loop, one per body
- Threading
  - Blocks can only hold 512 or 1024 threads
    - Up to 8 blocks can be resident in an SM at a time
    - SM can hold 768, 1024, or 1536 threads
    - We use 256 threads per block (greatest common divisor)
  - Need multiple blocks
    - Last block may not have full number of threads

Using N Threads

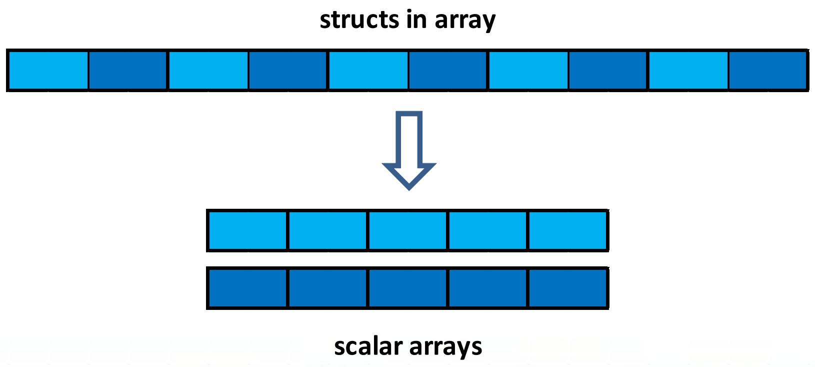
```
__global__ void ForceCalcKernel(int nbodies, struct Body *body, ...) {
  ~~for (i = 0; i < nbodies; i++) {~~
  i = threadIdx.x + blockIdx.x * blockDim.x; // compute i
  if (i < nbodies) { // in case last block is only partially used
    for (j = ...) {
      . . .
    }
  }
}
__global__ void AdvancingKernel(int nbodies, struct body *body, ...) {
  // same changes
}
#define threads 256
int main(...) {
  . . .
  int blocks = (nbodies + threads - 1) / threads; // compute block cnt
  for (timestep = ...) {
    ForceCalcKernel<<<~~1, 1~~blocks, threads>>>(nbodies, bodyl, ...);
    AdvancingKernel<<<~~1, 1~~blocks, threads>>>(nbodies, bodyl, ...);
  }
}
```

N-Thread Speedup

- Relative to 1 GPU thread
  - CC 1.1: 40 (8 PEs)
  - CC 1.3: 7781 (240 PEs)
  - CC 2.0: 6495 (448 PEs)
- Relative to 1 CPU thread
  - CC 1.1: 1.0
  - CC 1.3: 107.5
  - CC 2.0: 176.7
- Performance
  - Speedup much higher than number of PEs (5, 32, and 14.5 times)
  - Due to SMT latency hiding
- Per-core performance
  - CPU core delivers under 7.9, 4.4*, and 5* times as much performance as a GPU core (PE)

Using Scalar Arrays

- Data structure conversion
  - Arrays of structs are bad for coalescing
  - Bodies’ elements (e.g., mass fields) are not adjacent
- Optimize data structure
  - Use multiple scalar arrays, one per field (need 10)
  - Results in code bloat but often much better speed



Using Scalar Arrays

```
__global__ void ForceCalcKernel(int nbodies, float *mass, ...) {
  // change all "body[k].blah" to "blah[k]"
}
__global__ void AdvancingKernel(int nbodies, float *mass, ...) {
  // change all "body[k].blah" to "blah[k]"
}

int main(...) {
  float *mass, *posx, *posy, *posz, *velx, *vely, *velz, *accx, *accy,*accz;
  float *massl, *posxl, *posyl, *poszl, *velxl, *velyl, *velzl, ...;
  mass = (float *)malloc(sizeof(float) * nbodies); // etc
  . . .
  cudaMalloc((void**)&massl, sizeof(float)*nbodies); // etc
  cudaMemcpy(massl, mass, sizeof(float)*nbodies, cuda…HostToDevice); // etc
  for (timestep = ...) {
    ForceCalcKernel<<<1, 1>>>(nbodies, massl, posxl, ...);
    AdvancingKernel<<<1, 1>>>(nbodies, massl, posxl, ...);
  }
  cudaMemcpy(mass, massl, sizeof(float)*nbodies, cuda…DeviceToHost); // etc
  . . .
}
```

Scalar Array Speedup

- Problem size
  - n=10000, step=1
  - n=100000, step=1
  - n=100000, step=1
- Relative to struct
  - CC 1.1: 1.00
  - CC 1.3: 0.83
  - CC 2.0: 0.96
- Performance
  - Threads access same memory locations, not adjacent ones
    - Never coalesced in CC 1.1
    - Always combined but not coalesced in CC 1.3 & 2.0
  - Slowdowns presumably due to DRAM banks
- Scalar arrays
  - Still needed (see later)

Constant Kernel Parameters

- Kernel parameters
  - Lots of parameters due to scalar arrays
  - All but one parameter never change their value
- Constant memory
  - “Pass” parameters only once
  - Copy them into GPU’s constant memory
- Performance implications
  - Reduced parameter passing overhead
  - Constant memory has hardware cache

Constant Kernel Parameters

```
__constant__ int nbodiesd;
__constant__ float dthfd, epssqd, float *massd, *posxd, ...;

__global__ void ForceCalcKernel(int step) {
  // rename affected variables (add "d" to name)
}

__global__ void AdvancingKernel() {
  // rename affected variables (add "d" to name)
}

int main(...) {
  . . .
  cudaMemcpyToSymbol(massd, &massl, sizeof(void *)); // etc
  . . .
  for (timestep = ...) {
    ForceCalcKernel<<<1, 1>>>(step);
    AdvancingKernel<<<1, 1>>>();
  }
  . . .
}
```

Constant Mem Parameter Speedup

- Problem size
  - n=128, step=10000
  - n=1000, step=10000
  - n=1000, step=10000
- Speedup
  - CC 1.1: 1.017
  - CC 1.3: 1.015
  - CC 2.0: 1.016
- Performance
  - Minimal speedup
  - Only useful for very short kernels that are often invoked
- Benefit
  - Less shared memory used (may be crucial)

Using the RSQRT Instruction

- Slowest kernel operation
  - Computing one over the square root is very slow
  - GPU has slightly imprecise but fast 1/sqrt instruction (frequently used in graphics code to calculate inverse of distance to a point)
- IEEE floating-point accuracy compliance
  - CC 1.x is not entirely compliant
  - CC 2.x is compliant but offers faster non-compliant instructions

Using the RSQRT Instruction

```
for (i = 0; i < nbodies; i++) {
  . . .
  for (j = 0; j < nbodies; j++) {
    if (i != j) {
      dx = body[j].posx - px;
      dy = body[j].posy - py;
      dz = body[j].posz - pz;
      dsq = dx*dx + dy*dy + dz*dz;
      ~~dinv = 1.0f / sqrtf(dsq + epssq);~~
      dinv = rsqrtf(dsq + epssq);
      scale = body[j].mass * dinv * dinv * dinv;
      ax += dx * scale;
      ay += dy * scale;
      az += dz * scale;
    }
  }
  . . .
}
```

RSQRT Speedup

- Problem size
  - n=10000, step=1
  - n=100000, step=1
  - n=100000, step=1
- Speedup
  - CC 1.1: 1.00
  - CC 1.3: 0.99
  - CC 2.0: 1.83

- Performance
  - No change for CC 1.x
    - Compiler automatically uses less precise RSQRT as most FP ops are not fully precise anyhow
  - 83% speedup for CC 2.0
    - Over entire application
    - Compiler defaults to precise instructions
    - Explicit use of RSQRT indicates imprecision okay

Using 2 Loops to Avoid If Statement

- "if (i != j)" causes thread divergence
  - Break loop into two loops to avoid if statement

```
for (j = 0; j < nbodies; j++) {
  if (i != j) {
    dx = body[j].posx - px;
    dy = body[j].posy - py;
    dz = body[j].posz - pz;
    dsq = dx*dx + dy*dy + dz*dz;
    dinv = rsqrtf(dsq + epssq);
    scale = body[j].mass * dinv * dinv * dinv;
    ax += dx * scale;
    ay += dy * scale;
    az += dz * scale;
  }
}
```

Using 2 Loops to Avoid If Statement

```
for (j = 0; j < i; j++) {
  dx = body[j].posx - px;
  dy = body[j].posy - py;
  dz = body[j].posz - pz;
  dsq = dx*dx + dy*dy + dz*dz;
  dinv = rsqrtf(dsq + epssq);
  scale = body[j].mass * dinv * dinv * dinv;
  ax += dx * scale;
  ay += dy * scale;
  az += dz * scale;
}
for (j = i+1; j < nbodies; j++) {
  dx = body[j].posx - px;
  dy = body[j].posy - py;
  dz = body[j].posz - pz;
  dsq = dx*dx + dy*dy + dz*dz;
  dinv = rsqrtf(dsq + epssq);
  scale = body[j].mass * dinv * dinv * dinv;
  ax += dx * scale;
  ay += dy * scale;
  az += dz * scale;
}
```

Loop Duplication Speedup

- Problem size
  - n=10000, step=1
  - n=100000, step=1
  - n=100000, step=1
- Speedup
  - CC 1.1: 1.02
  - CC 1.3: 0.55
  - CC 2.0: 1.00

- Performance
  - No change for 1.1 & 2.0
    - Divergence moved to loop
  - 45% slowdown for CC 1.3
    - Unclear why
- Discussion
  - Not a useful optimization
  - Code bloat
  - A little divergence is okay (only 1 in 3125 iterations)

Blocking using Shared Memory

- Code is memory bound
  - Each warp streams in all bodies’ mass and position
- Block inner loop
  - Read block of mass & position info into shared mem
  - Requires barrier (fast hardware barrier within SM)
- Advantage
  - A lot fewer main memory accesses
  - Remaining accesses are fully coalesced (due to usage of scalar arrays)

Blocking using Shared Memory

```
__shared__ float posxs[threads], posys[threads], poszs[…], masss[…];
j = 0;
for (j1 = 0; j1 < nbodiesd; j1 += THREADS) { // first part of loop
  idx = tid + j1;
  if (idx < nbodiesd) { // each thread copies 4 words (fully coalesced)
    posxs[id] = posxd[idx];  posys[id] = posyd[idx];
    poszs[id] = poszd[idx];  masss[id] = massd[idx];
  }
  __syncthreads(); // wait for all copying to be done
  bound = min(nbodiesd - j1, THREADS);
  for (j2 = 0; j2 < bound; j2++, j++) { // second part of loop
    if (i != j) {
      dx = posxs[j2] – px;  dy = posys[j2] – py;  dz = poszs[j2] - pz;
      dsq = dx*dx + dy*dy + dz*dz;
      dinv = rsqrtf(dsq + epssqd);
      scale = masss[j2] * dinv * dinv * dinv;
      ax += dx * scale;  ay += dy * scale;  az += dz * scale;
    }
  }
}
```

Blocking Speedup

- Problem size
  - n=10000, step=1
  - n=100000, step=1
  - n=100000, step=1
- Speedup
  - CC 1.1: 8.2
  - CC 1.3: 3.7
  - CC 2.0: 1.1

- Performance
  - Great speedup for CC 1.x
  - Little speedup for CC 2.0
    - Has hardware data cache
- Discussion
  - Very important optimization for memory bound code
  - Even with L1 cache

Loop Unrolling

- CUDA compiler
  - Generally good at unrolling loops with fixed bounds
  - Does not unroll inner loop of our example code
- Use pragma to unroll

```
#pragma unroll 8
for (j2 = 0; j2 < bound; j2++, j++) {
  if (i != j) {
    dx = posxs[j2] – px;  dy = posys[j2] – py;  dz = poszs[j2] - pz;
    dsq = dx*dx + dy*dy + dz*dz;
    dinv = rsqrtf(dsq + epssqd);
    scale = masss[j2] * dinv * dinv * dinv;
    ax += dx * scale;  ay += dy * scale;  az += dz * scale;
  }
}
```

Loop Unrolling Speedup

- Problem size
  - n=10000, step=1
  - n=100000, step=1
  - n=100000, step=1
- Speedup
  - CC 1.1: 1.06
  - CC 1.3: 1.07
  - CC 2.0: 1.16
- Performance
  - Noticeable speedup
  - All three GPUs
- Discussion
  - Can be useful
  - May increase register usage, which may lower maximum number of threads per block and result in slowdown

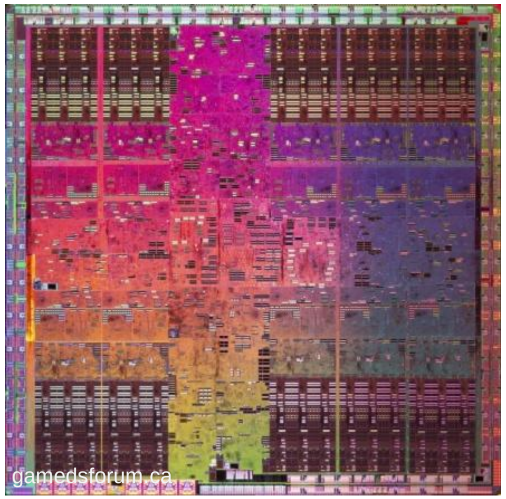
CC 2.0 Absolute Performance

- Problem size
  - n=100000, step=1
- Runtime
  - 612 ms
- FP operations
  - 326.7 GFlop/s (SP)
  - 219.2 GFlops/s (DP)
- Main mem throughput
  - 1.035 GB/s, 1.388 GB/s

- Not peak performance
  - Only 32% of 1030 GFlop/s
    - Peak assumes FMA every cyc
  - 3 sub (1c), 3 fma (1c), 1 rsqrt (8c), 3 mul (1c), 3 fma (1) = 20c for 20 Flop
  - 63% of realistic peak of 515.2 GFlop/s
    - Assumes no non-FP ops
  - With int ops = 31c for 20 Flop
  - 99% of actual peak of 330.45 GFlop/s

Outline

- Introduction
- CUDA overview
- N-body example
- Porting and tuning
- Other considerations
- Conclusions



Things to Consider

- Minimize PCIe transfers
  - Implementing entire algorithm on GPU, even some slow serial code sections, might be overall win
- Locks and synchronization
  - Lightweight locks & barriers often possible within SM
  - Slow across different SMs
- CC 2.0’s hardware L1 caches are not coherent
  - Disable or use volatile & fences to avoid deadlocks
- Can stream data to/from GPU while computing

Warp-Based Execution

```
// wrong on GPU, correct on CPU
do {
  cnt = 0;
  if (ready[i] != 0) cnt++;
  if (ready[j] != 0) cnt++;
} while (cnt < 2);
ready[k] = 1;

// correct
do {
  cnt = 0;
  if (ready[i] != 0) cnt++;
  if (ready[j] != 0) cnt++;
  if (cnt == 2) ready[k] = 1;
} while (cnt < 2);
```

- Problem
  - Thread divergence
  - Loop exiting threads wait for other threads in warp to also exit
  - “ready[k] = 1” is not executed until all threads in warp are done with loop
  - Possible deadlock

Hybrid Execution

- CPU needed
  - CPU always needed for program launch and most I/O
  - CPU much faster on serial program segments
- GPU 10 times faster than CPU on parallel code
  - Running 10% of problem on CPU is hardly worthwhile
  - Complicates programming and requires data transfer
    - Best CPU data structure is often not best for GPU
- PCIe bandwidth much lower than GPU bandwidth
  - 1.6 to 6.5 GB/s versus 144 GB/s
  - Merging CPU and GPU on same die (like AMD's Fusion APU) will make finer grain switching possible

Outline

- Introduction
- CUDA overview
- N-body example
- Porting and tuning
- Other considerations
- Conclusions

Summary and Conclusions

- Step-by-step porting and tuning of CUDA code
  - Example: n-body simulation
- GPUs have very powerful hardware
  - Only exploitable with some codes
  - Even harder to program and optimize for than CPUs

- Acknowledgments
  - Keshav Pingali: overall project support
  - TACC, NVIDIA: hardware resources
  - NSF, IBM, NEC, Intel, UT Austin, Texas State: funding