

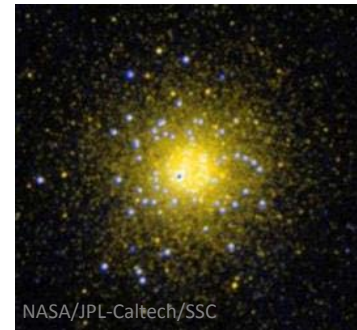
# Parallel Programming

## N-Body Simulation in CUDA

Slides based on Martin Burtscher's tutorial

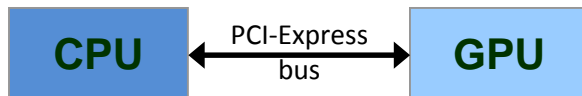
# Outline

- Review: GPU programming
- N-body example
- Porting and tuning



# CUDA Programming Model

- Non-graphics programming
  - Uses GPU as massively parallel co-processor



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

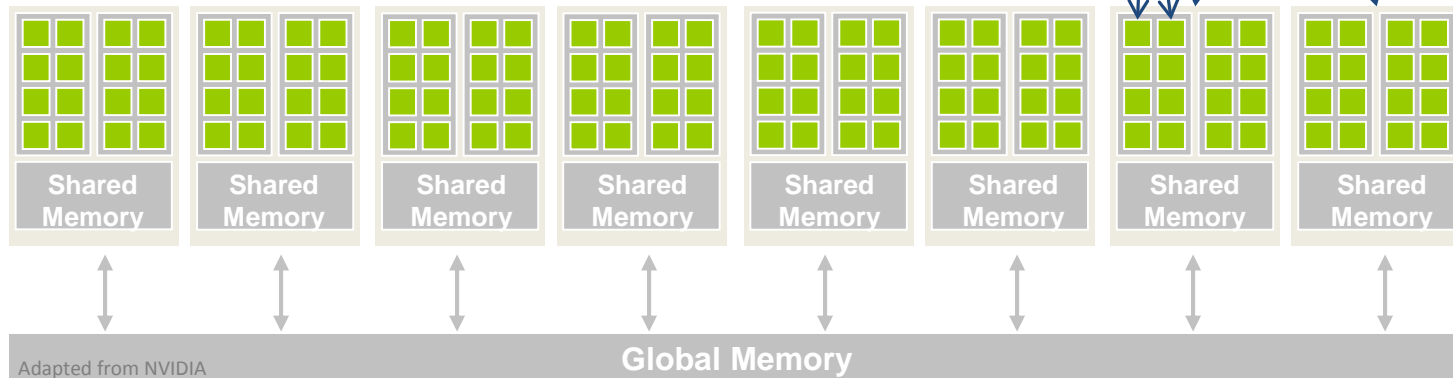
- C/C++ with extensions
  - Function launch
    - Calling functions on GPU
  - Memory management
    - GPU memory allocation, copying data to/from GPU
  - Declaration qualifiers
    - Device, shared, local, etc.
  - Special instructions
    - Barriers, fences, etc.
  - Keywords
    - threadIdx, blockIdx

# Calling GPU Kernels

- Kernels are functions that run on the GPU
    - Callable by CPU code
    - CPU can continue processing while GPU runs kernel
- ```
KernelName<<<m, n>>>(arg1, arg2, ...);
```
- Launch configuration (programmer selectable)
    - GPU spawns **m blocks** of **n threads** per block (i.e., **m\*n** threads total) that run a copy of the same function
    - Normal function parameters: passed conventionally
      - Different address space

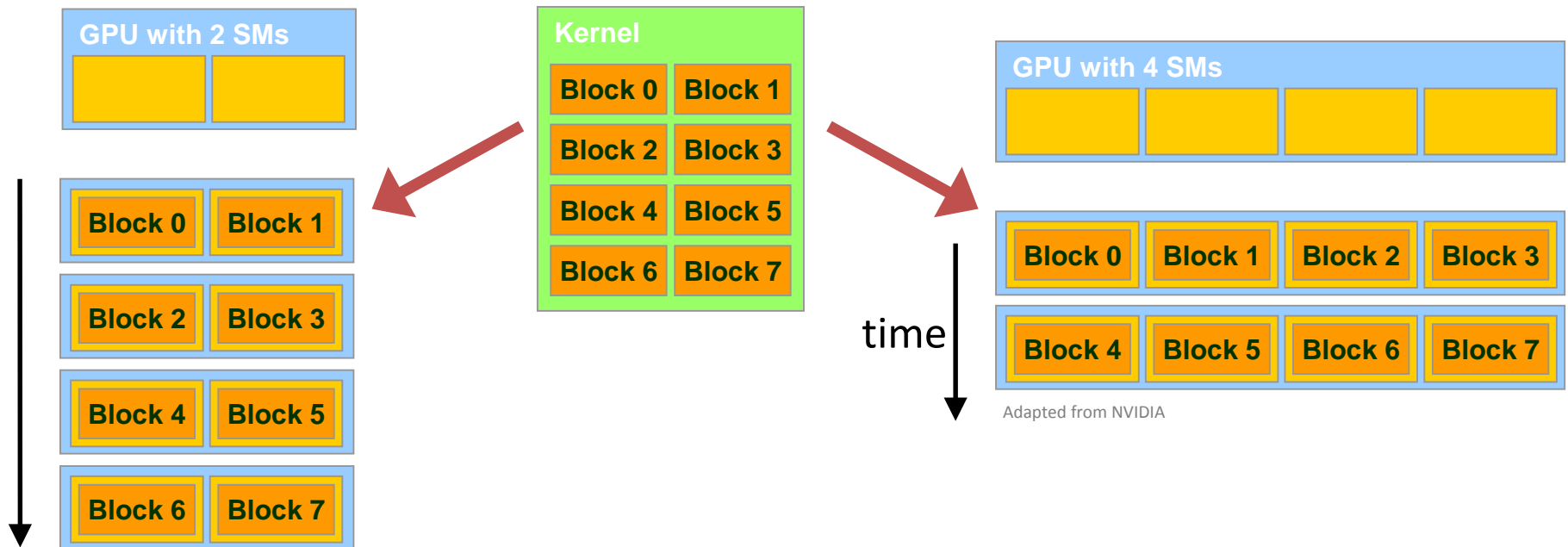
# GPU Architecture

- GPUs consist of **Streaming Multiprocessors (SMs)**
  - 1 to 30 SMs per chip (run blocks)
- SMs contain **Processing Elements (PEs)**
  - 8, 32, or 192 PEs per SM (run threads)



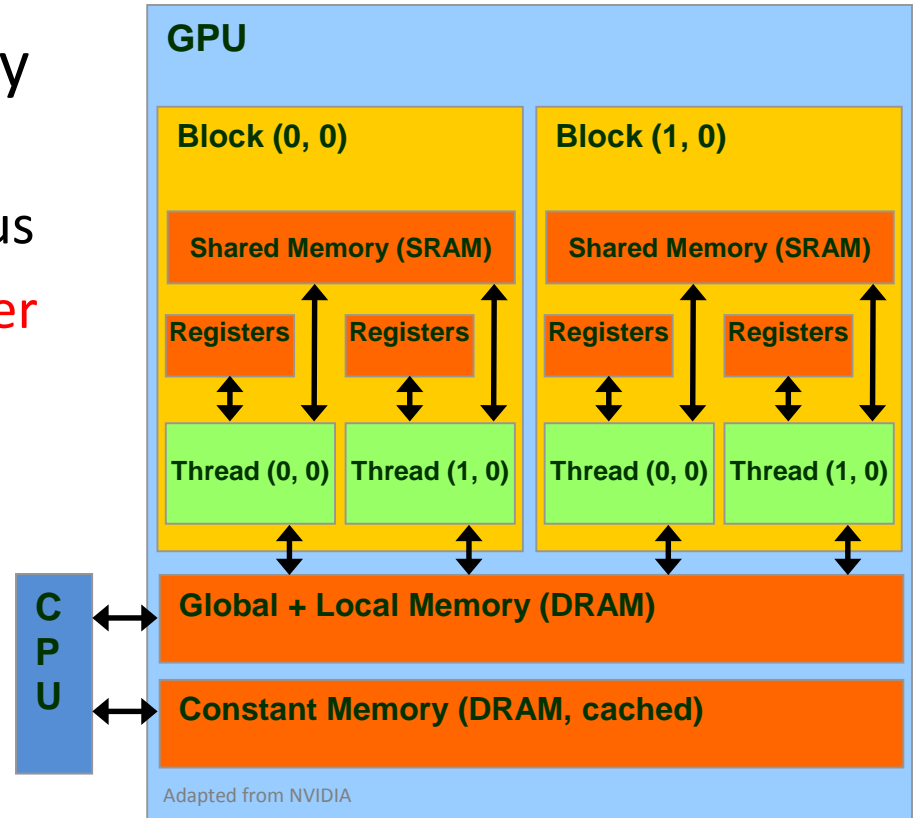
# Block Scalability

- 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



# GPU Memories

- Separate from CPU memory
  - CPU can access GPU's global & constant mem. via PCIe bus
  - Requires slow explicit transfer
- Visible GPU memory types
  - Registers (per thread)
  - Local mem. (per thread)
  - Shared mem. (per block)
    - Software-controlled cache
  - Global mem. (per kernel)
  - Constant mem. (read only)



- Slow communic. between blocks

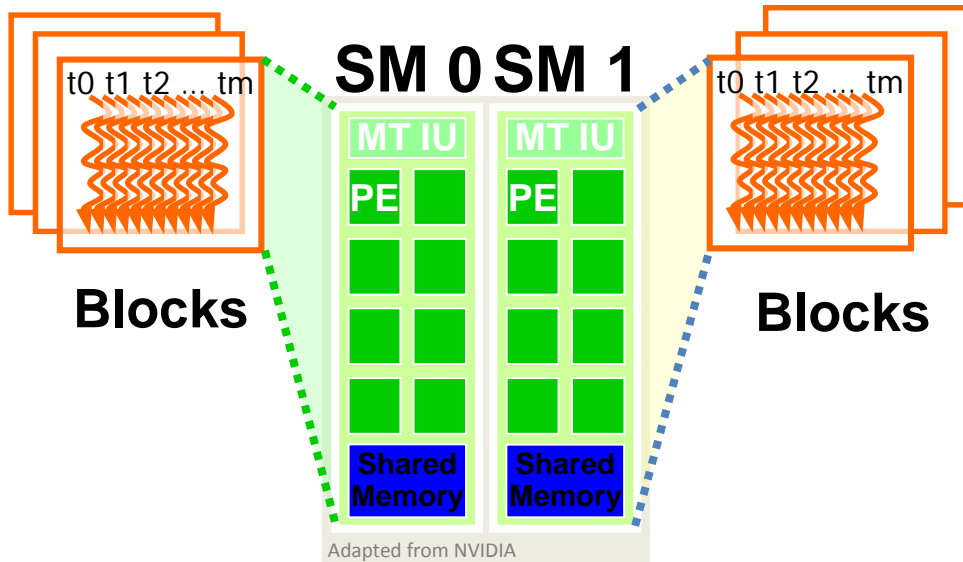
# SM Internals (Fermi and Kepler)

- Caches
  - Software-controlled shared memory
  - Hardware-controlled **incoherent** L1 data cache
  - 64 kB combined size, can be split 16/48, 32/32, 48/16
- Synchronization support
  - Fast hardware **barrier** within block (`__syncthreads()`)
  - **Fence** instructions: memory consistency & coherency
- Special operations
  - Thread **voting** (warp-based reduction operations)



# Block and Thread Allocation Limits

- Blocks assigned to SMs
  - Until first limit reached
- Threads assigned to PEs
- Hardware limits
  - 8/16 active blocks/SM
  - 1024, 1536, or 2048 resident threads/SM
  - 512 or 1024 threads/blk
  - 16k, 32k, or 64k regs/SM
  - 16 kB or 48 kB shared memory per SM
  - $2^{16}-1$  or  $2^{31}-1$  blks/kernel



# Warp-based Execution

- 32 contiguous threads form a **warp**
  - Execute **same** instruction in same cycle (or disabled)
  - Warps are scheduled out-of-order with respect to each other to hide latencies
- Thread **divergence**
  - Some threads in warp jump to different PC than others
  - Hardware runs subsets of warp until they re-converge
  - Results in reduction of parallelism (performance loss)

# Thread Divergence

- Non-divergent code

```
if (threadID >= 32) {  
    some_code;  
} else {  
    other_code;  
}
```



- Divergent code

```
if (threadID >= 13) {  
    some_code;  
} else {  
    other_code;  
}
```



# Parallel Memory Accesses

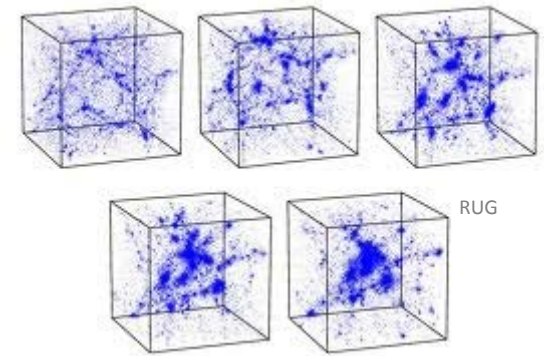
- **Coalesced** main memory access
  - Under some conditions, HW combines multiple (half) warp memory accesses into a single coalesced access
- **Bank-conflict-free** shared memory access
  - No superword alignment or contiguity requirements

# Warnings for GPU Programming

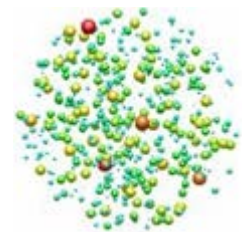
- GPUs can only execute some types of code fast
  - Need lots of data parallelism, data reuse, & regularity
- GPUs are harder to program and tune than CPUs
  - poor tool support
  - architecture
  - poor support for irregular code

# 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 { // 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
        }
    }
    ...
}
```

# N-body Algorithm Suitability for GPU

- 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, *body1;  
    . . .  
    cudaMalloc((void**)&body1, sizeof(Body)*nbodies);  
    cudaMemcpy(body1, body, sizeof(Body)*nbodies, cudaMemcpyHostToDevice);  
    for (timestep = ...) {  
        ForceCalcKernel<<<1, 1>>>(nbodies, body1, ...);  
        AdvancingKernel<<<1, 1>>>(nbodies, body1, ...);  
    }  
    cudaMemcpy(body, body1, sizeof(Body)*nbodies, cudaMemcpyDeviceToHost);  
    cudaFree(body1);  
    . . .  
}
```

# Evaluation Methodology

- Systems and compilers
  - CC 1.3: Quadro FX 5800, nvcc 3.2
    - 30 SMs, 240 PEs, 1.3 GHz, 30720 resident threads
  - CC 2.0: Tesla C2050, nvcc 3.2
    - 14 SMs, 448 PEs, 1.15 GHz, 21504 resident threads
  - CC 3.0: GeForce GTX 680, nvcc 4.2
    - 8 SMs, 1536 PEs, 1.0 GHz, 16384 resident threads
- Input and metric
  - 1k, 10k, or 100k star clusters (Plummer model)
  - Median runtime of three experiments, excluding I/O

# 1-Thread Performance

- Problem size
  - $n=10000$ ,  $\text{step}=1$
  - $n=10000$ ,  $\text{step}=1$
  - $n=3000$ ,  $\text{step}=1$
- Slowdown rel. to CPU
  - CC 1.3: 72.4
  - CC 2.0: 36.7
  - CC 3.0: 68.1

(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 1.3)
  - 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 8/16 blocks can be resident in an SM at a time
    - SM can hold 1024, 1536, or 2048 threads
    - Use 256 threads per block (works for all three GPUs)
  - Need multiple blocks
    - Last block may not need all of its 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,...) // same changes  
  
#define threads 256  
int main(...) {  
    . . .  
    int blocks = (nbodies + threads - 1) / threads; // compute block cnt  
    for (timestep = ...) {  
        ForceCalcKernel<<<1, 1blocks, threads>>>(nbodies, body1, ...);  
        AdvancingKernel<<<1, 1blocks, threads>>>(nbodies, body1, ...);  
    }  
}
```

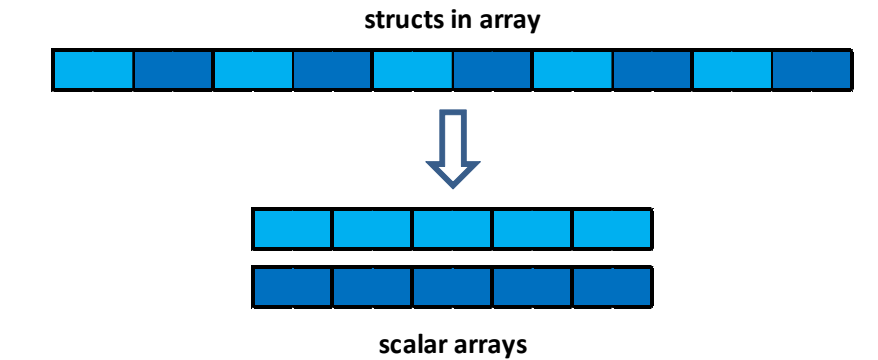


# N Thread Speedup

- Relative to 1 GPU thread
  - CC 1.3: **7781** (240 PEs)
  - CC 2.0: **6495** (448 PEs)
  - CC 3.0: **12150** (1536 PEs)
- Relative to 1 CPU thread
  - CC 1.3: **107.5**
  - CC 2.0: **176.7**
  - CC 3.0: **176.2**
- Performance
  - Speedup much higher than number of PEs (32, 14.5, and 7.9 times)
  - Due to SMT latency hiding
- Per-core performance
  - CPU core delivers up to 4.4, 5, and 8.7 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<<<blocks, threads>>>(nbodies, massl, posxl, ...);  
        AdvancingKernel<<<blocks, threads>>>(nbodies, massl, posxl, ...);  
    }  
    cudaMemcpy(mass, massl, sizeof(float)*nbodies, cuda...DeviceToHost); // etc  
    . . .  
}
```

# Scalar Array Speedup

- Problem size
  - $n=100000$ ,  $\text{step}=1$
  - $n=100000$ ,  $\text{step}=1$
  - $n=300000$ ,  $\text{step}=1$
- Relative to struct
  - CC 1.3: **0.83**
  - CC 2.0: **0.96**
  - CC 3.0: **0.82**
- Performance
  - Threads access **same** memory locations, not adjacent ones
    - Always combined but not really coalesced access
  - Slowdowns may be due to DRAM page/TLB misses
- 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, &mass1, sizeof(void *)); // etc
    . . .
    for (timestep = ...) {
        ForceCalcKernel<<<1, 1>>>(step);
        AdvancingKernel<<<1, 1>>>();
    }
    . . .
}
```

# Constant Mem. Parameter Speedup

- Problem size
  - n=1000, step=10000
  - n=1000, step=10000
  - n=3000, step=10000
- Speedup
  - CC 1.3: 1.015
  - CC 2.0: 1.016
  - CC 3.0: 0.971
- Performance
  - Minimal perf. impact
  - May be useful for very short kernels that are often invoked
- Benefit
  - Less shared memory used on CC 1.3 devices

# Using the RSQRTF Instruction

- Slowest kernel operation
  - Computing one over the square root is very slow
  - GPU has slightly imprecise but fast  $1/\text{sqrt}$  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 and above are compliant but also offer 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=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 0.99
  - CC 2.0: 1.83
  - CC 3.0: 1.64
- Performance
  - Little change for CC 1.3
    - Compiler automatically uses less precise RSQRTE 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 RSQRTE indicates imprecision okay

# Using 2 Loops to Avoid If Statement

- “if (i != j)” creates code 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=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 0.55
  - CC 2.0: 1.00
  - CC 3.0: 1.00
- Performance
  - No change for 2.0 & 3.0
    - Divergence moved to loop
  - 45% slowdown for CC 1.3
    - Unclear reason
- 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' masses and positions
- Use shared memory in inner loop
  - Read block of mass & position info into shared mem
  - Requires barriers (fast hardware barrier within SM)
- Advantage
  - A lot fewer main memory accesses
  - Remaining main memory 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;
        }
    }
    __syncthreads(); // wait for all force calculations to be done
}
```

# Blocking Speedup

- Problem size
  - $n=100000$ ,  $\text{step}=1$
  - $n=100000$ ,  $\text{step}=1$
  - $n=300000$ ,  $\text{step}=1$
- Speedup
  - CC 1.3: 3.7
  - CC 2.0: 1.1
  - CC 3.0: 1.6
- Performance
  - Great speedup for CC 1.3
  - Some speedup for others
    - 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 (and pad arrays)

```
#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
  - $n=100000$ ,  $\text{step}=1$
  - $n=100000$ ,  $\text{step}=1$
  - $n=300000$ ,  $\text{step}=1$
- Speedup
  - CC 1.3: 1.07
  - CC 2.0: 1.16
  - CC 3.0: 1.07
- Performance
  - Insignificant 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
  - $n=100000$ ,  $\text{step}=1$
- Runtime
  - 612 ms
- FP operations
  - 326.7 GFlop/s
- Main mem throughput
  - 1.035 GB/s
- Not peak performance
  - Only 32% of 1030 GFlop/s
    - Peak assumes FMA every cycle
  - 3 sub (1c), 3 fma (1c), 1 rsqrt (8c), 3 mul (1c), 3 fma (1c) = 20c for 20 Flop
  - 63% of realistic peak of 515.2 GFlop/s
    - Assumes no non-FP operations
  - With int ops = 31c for 20 Flop
  - 99% of actual peak of 330.45 GFlop/s

# Eliminating the If Statement

- Algorithmic optimization
  - Potential softening parameter avoids division by zero
  - If-statement is not necessary and can be removed
    - Eliminates thread divergence

```
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;  
  }  
}
```

# If Elimination Speedup

- Problem size
  - $n=100000$ , step=1
  - $n=100000$ , step=1
  - $n=300000$ , step=1
- Speedup
  - CC 1.3: 1.38
  - CC 2.0: 1.54
  - CC 3.0: 1.64
- Performance
  - Large speedup
  - All three GPUs
- Discussion
  - No thread divergence
  - Allows compiler to schedule code much better

# Rearranging Terms

- Generated code is suboptimal
  - Compiler does not emit as many fused multiply-add (FMA) instructions as it could
  - Rearrange terms in expressions to help compiler
    - Need to check generated assembly code

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

# FMA Speedup

- Problem size
  - $n=100000$ ,  $\text{step}=1$
  - $n=100000$ ,  $\text{step}=1$
  - $n=300000$ ,  $\text{step}=1$
- Speedup
  - CC 1.3: 1.03
  - CC 2.0: 1.05
  - CC 3.0: 1.06
- Performance
  - Small speedup
  - All three GPUs
- Discussion
  - Seemingly needless transformations may make a difference

# Higher Unroll Factor

- Problem size
  - $n=100000$ ,  $\text{step}=1$
  - $n=100000$ ,  $\text{step}=1$
  - $n=300000$ ,  $\text{step}=1$
- Speedup
  - CC 1.3: 1.01
  - CC 2.0: 1.04
  - CC 3.0: 0.93
- Unroll 128 times
  - Avoid looping overhead
  - Now that there are no *ifs*
- Performance
  - Little speedup/slowdown
- Discussion
  - Carefully choose unroll factor (manually tune)



# Compiler Flags

- Problem size
  - n=100000, step=1
  - n=100000, step=1
  - n=300000, step=1
- Speedup
  - CC 1.3: 1.00
  - CC 2.0: 1.18
  - CC 3.0: 1.15
- -use\_fast\_math
  - “-ftz=true” suffices  
(flush denormals to zero)
  - Makes SP FP operations faster except on CC 1.3
- Performance
  - Significant speedup
- Discussion
  - Use faster but less precise operations when prudent

# Final Absolute Performance

- CC 2.0 Fermi GTX 480
  - Problem size
    - $n=100000$ , step=1
  - Runtime
    - 296.1 ms
  - FP operations
    - 675.6 GFlop/s (SP)
    - 66% of peak performance
    - 261.1 GFlops/s (DP)
  - Main mem throughput
    - 2.139 GB/s
- CC 3.0 Kepler GTX 680
  - Problem size
    - $n=300000$ , step=1
  - Runtime
    - 1073 ms
  - FP operations
    - 1677.6 GFlop/s (SP)
    - 54% of peak performance
    - 88.7 GFlops/s (DP)
  - Main mem throughput
    - 5.266 GB/s

# Hybrid Execution

- CPU always needed for program launch and 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 192 GB/s
  - But can send data while CPU & GPU are computing
  - Merging CPU and GPU on same die (e.g., AMD's Fusion APU) makes finer grain switching possible

# Summary

- Step-by-step porting and tuning of CUDA code
  - Example: n-body simulation
- GPUs have **very powerful hardware**
  - But only exploitable with some code
  - Harder to program and optimize than CPU hardware