COMP 633 - Parallel Computing

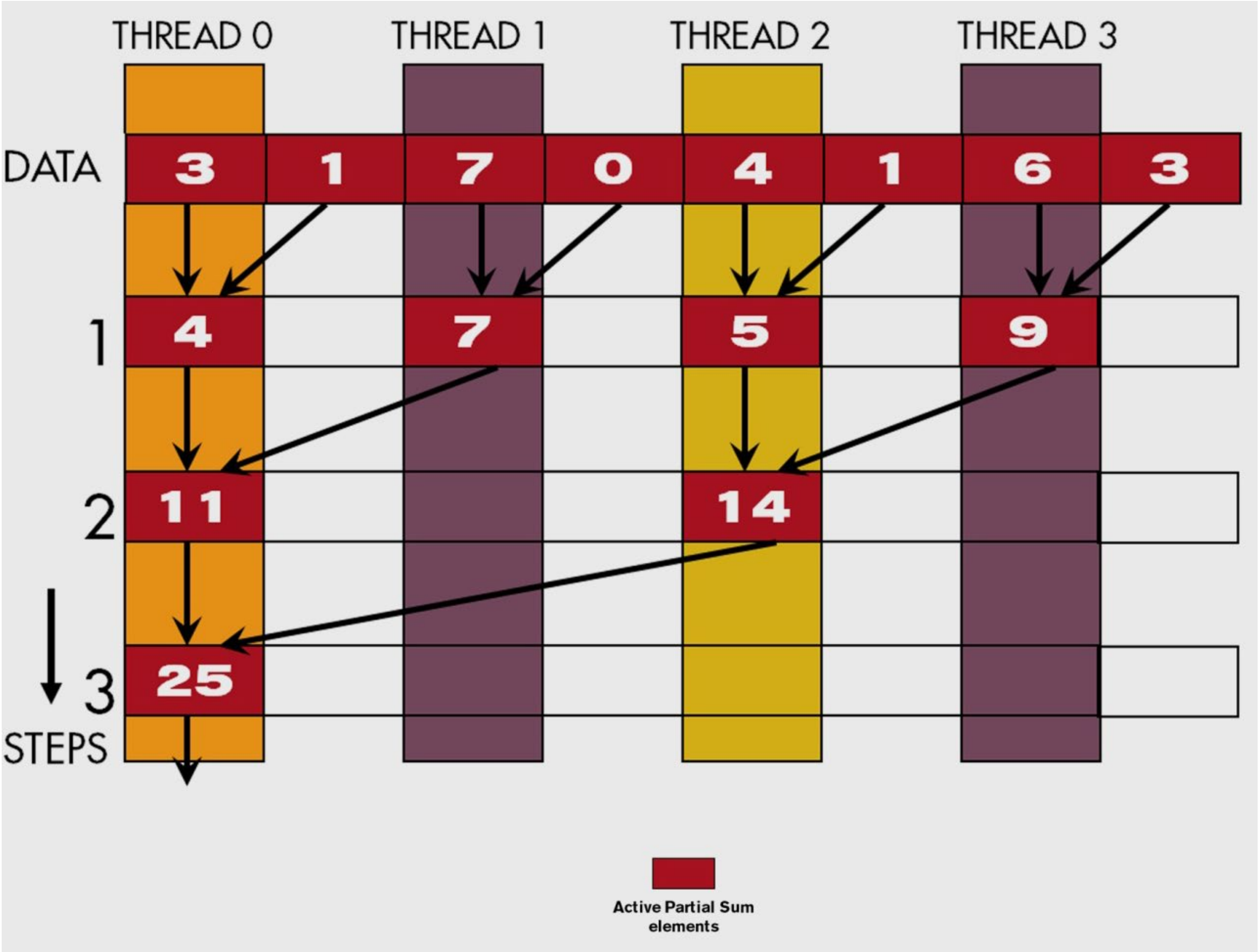
Lecture 14
October 14, 2021

Programming Accelerators

Some CUDA examples

- Sequence reduction
  - Illustrating control divergence
- Matrix multiplication
  - Illustrating shmem reuse
- Nbody computation
  - Illustrating a computation we know

Example 1: Parallel Sum Reduction



Parallel Sum Reduction

- Parallel implementation
  - halve # of active threads in each step, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads

- In-place reduction using shared memory within a block
  - The original vector of floats is in device memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0 of the partial sum vector
  - Reduces global memory traffic due to partial sum values

Some Observations on the naïve reduction kernel

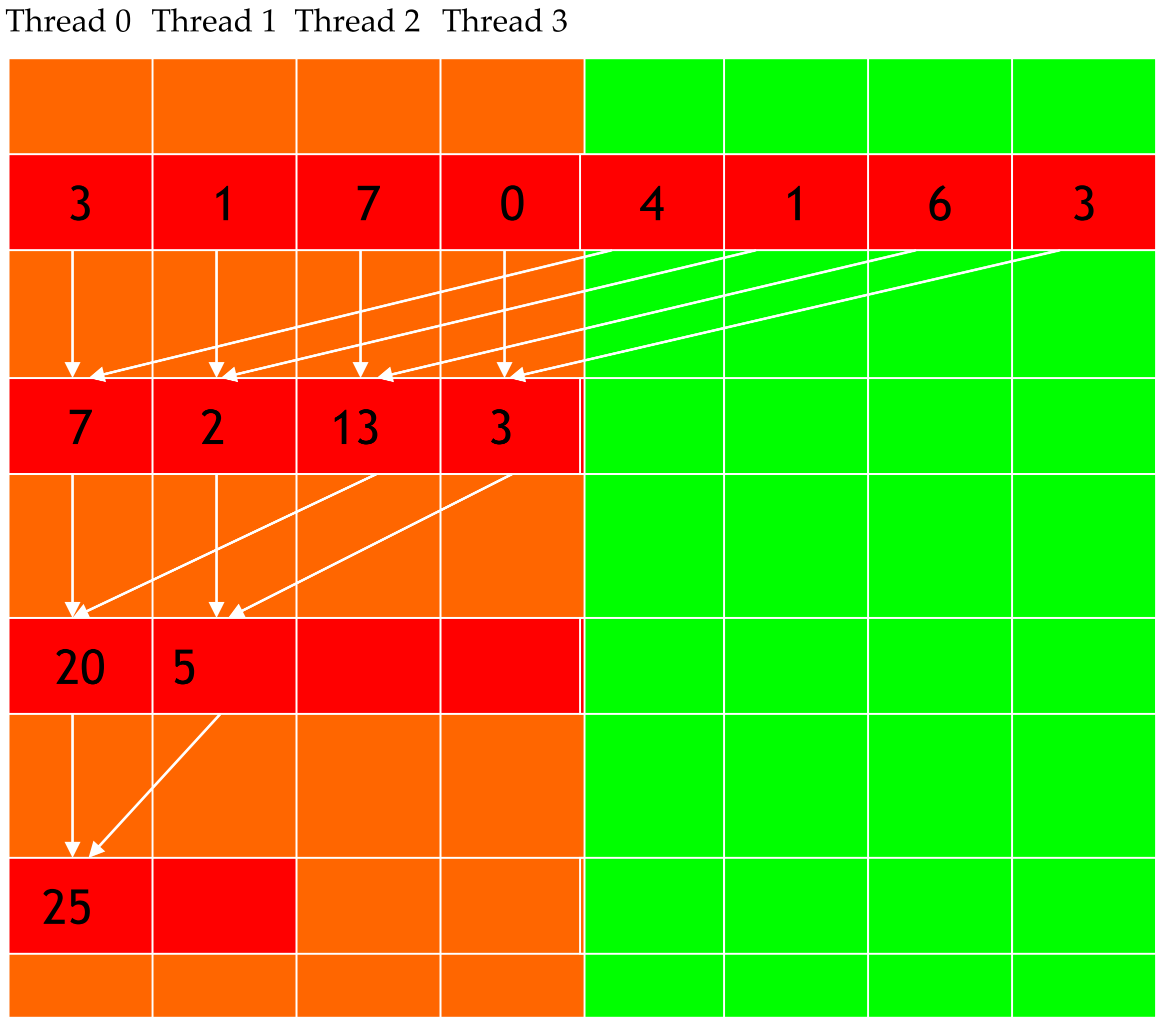
- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources

- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the `if` test, poor resource utilization but no divergence
    - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- For reduction, compact the partial sums to the front locations in the partialSum[ ] array
- Keep the active threads consecutive

Example with 4 threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
    {
        __syncthreads();
        if (t < stride)
           partialSum[t] += partialSum[t+stride];
    }
```

A Quick Analysis

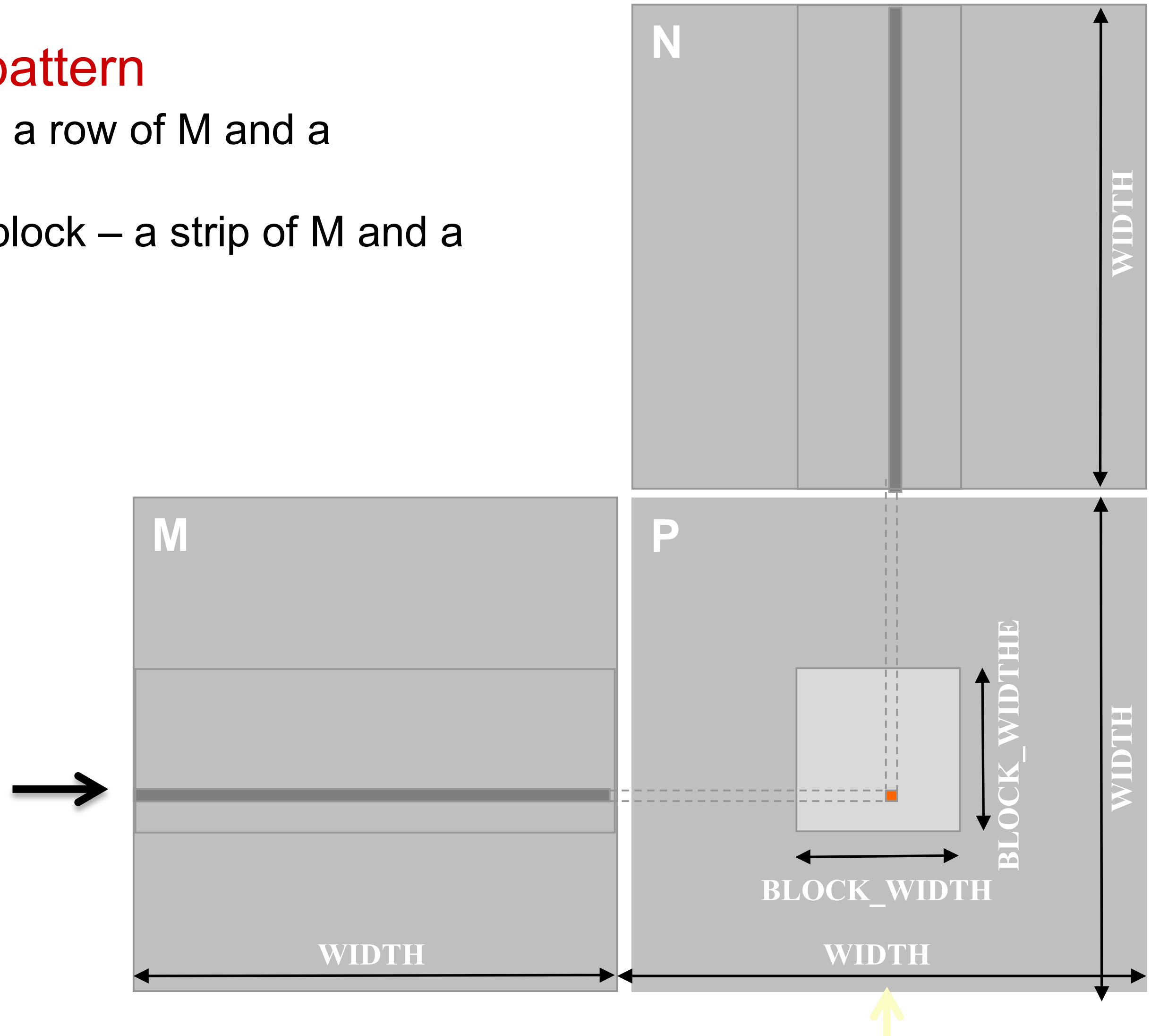
- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

Example 2

- Matrix multiplication
  - Illustrating shmem use

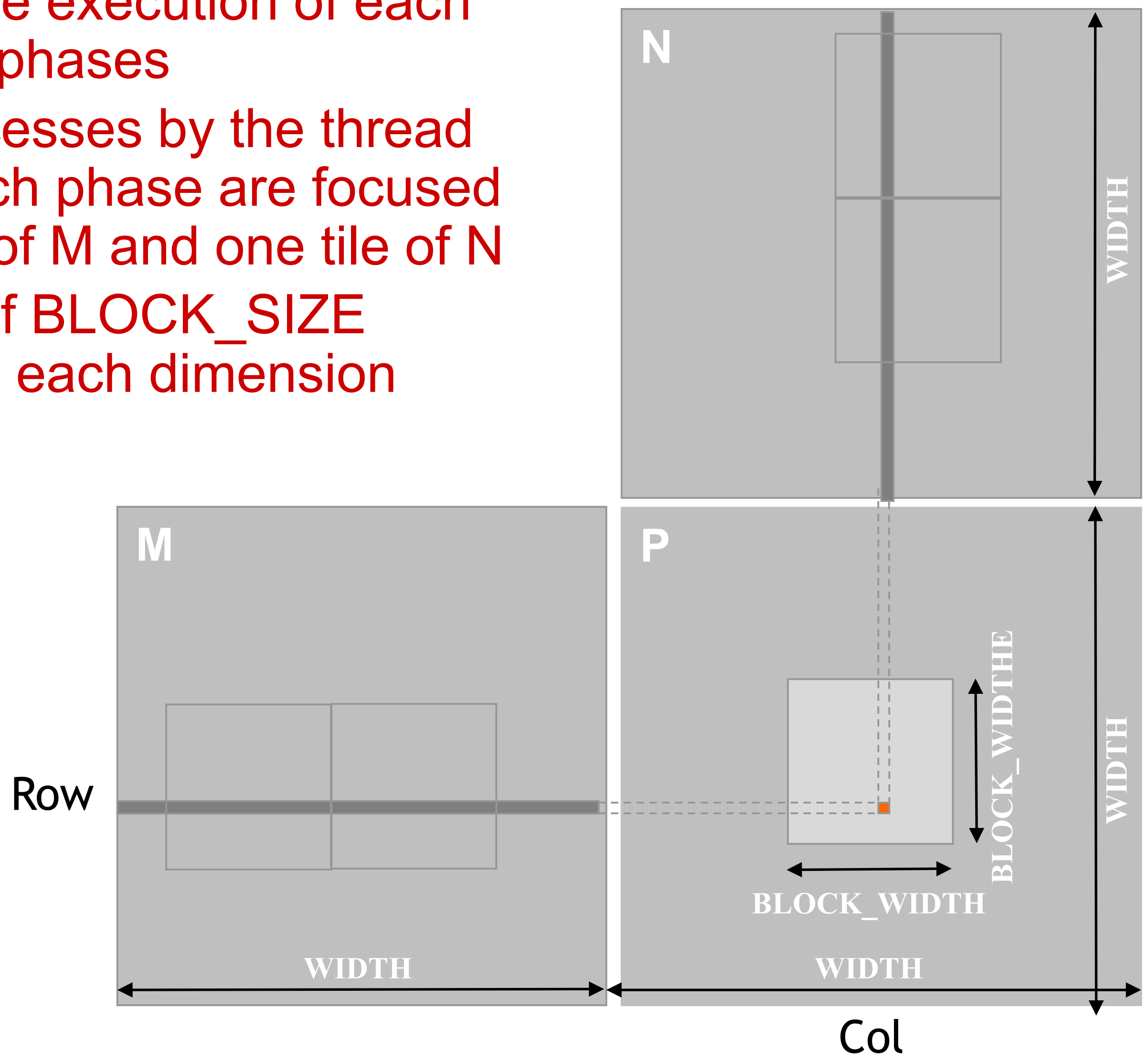
Matrix Multiplication

- Data access pattern
  - Each thread - a row of M and a column of N
  - Each thread block – a strip of M and a strip of N



Tiled Matrix Multiplication

- Break up the execution of each thread into phases
- so data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in tiled code

Phase 0 Load for Block (0,0)

$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$

$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$

$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$

$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

$N_{0,0}$ $N_{0,1}$

$N_{1,0}$ $N_{1,1}$

Shared Memory

$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$

$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$

$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$

$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Shared Memory

$M_{0,0}$ $M_{0,1}$

$M_{1,0}$ $M_{1,1}$

$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$

$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$

$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$

$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Phase 0 Use for Block (0,0) (iteration 0)

$$\begin{matrix} N_{0,0} & N_{0,1} & N_{0,2} & N_{0,3} \\ N_{1,0} & N_{1,1} & N_{1,2} & N_{1,3} \\ N_{2,0} & N_{2,1} & N_{2,2} & N_{2,3} \\ N_{3,0} & N_{3,1} & N_{3,2} & N_{3,3} \end{matrix}$$

$$\begin{matrix} N_{0,0} & N_{0,1} \\ N_{1,0} & N_{1,1} \end{matrix}$$

Shared Memory

Shared Memory

$$\begin{matrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \end{matrix}$$

$$\begin{matrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{matrix}$$

$$\begin{matrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \end{matrix}$$

Phase 0 Use for Block (0,0) (iteration 1)

$$\begin{matrix} N_{0,0} & N_{0,1} & N_{0,2} & N_{0,3} \\ N_{1,0} & N_{1,1} & N_{1,2} & N_{1,3} \\ N_{2,0} & N_{2,1} & N_{2,2} & N_{2,3} \\ N_{3,0} & N_{3,1} & N_{3,2} & N_{3,3} \end{matrix}$$

$$\begin{matrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \end{matrix}$$

Shared Memory

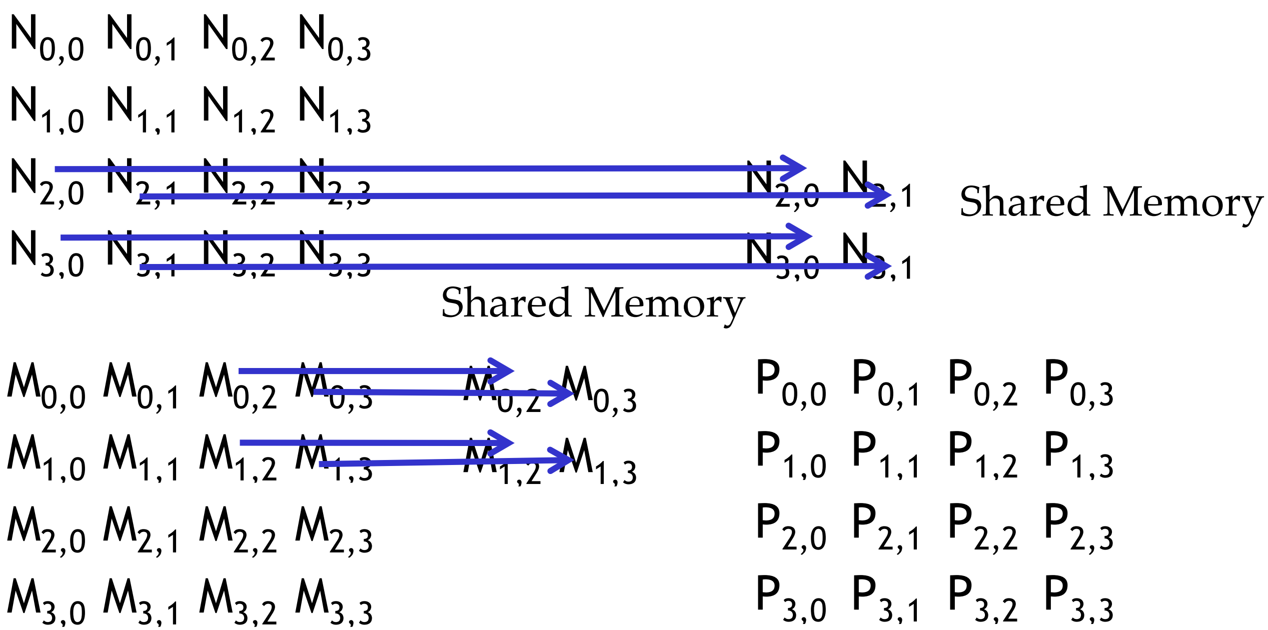
$$\begin{matrix} N_{0,0} & N_{0,1} \\ N_{1,0} & N_{1,1} \end{matrix}$$

Shared Memory

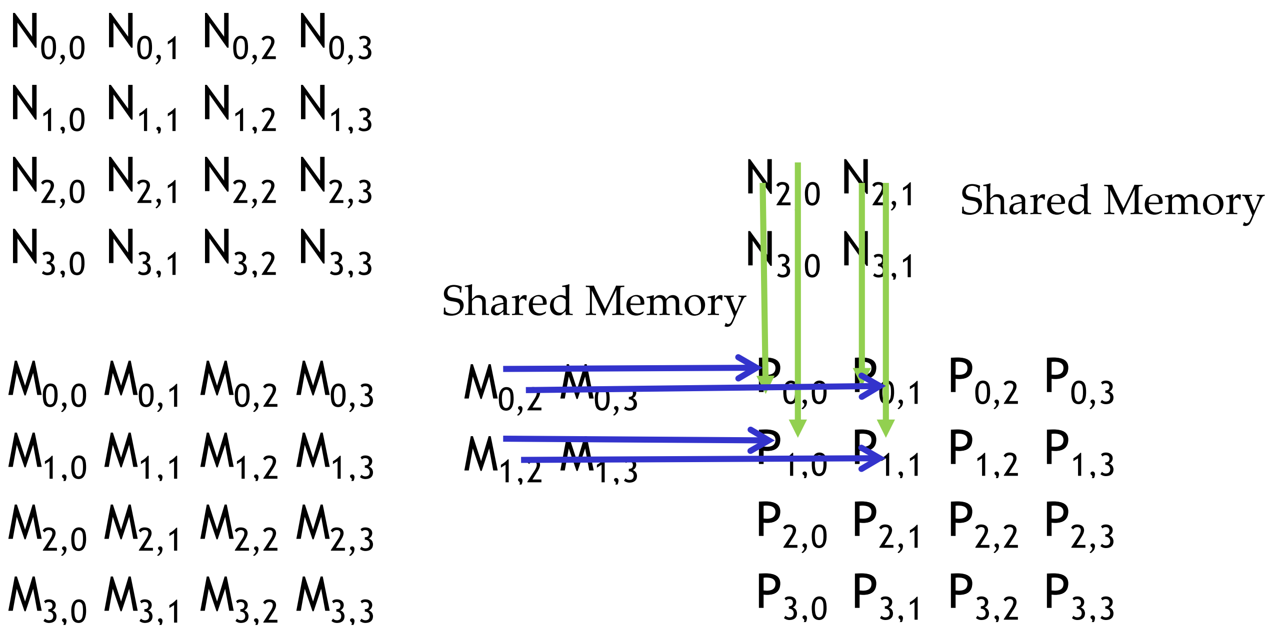
$$\begin{matrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{matrix}$$

$$\begin{matrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \end{matrix}$$

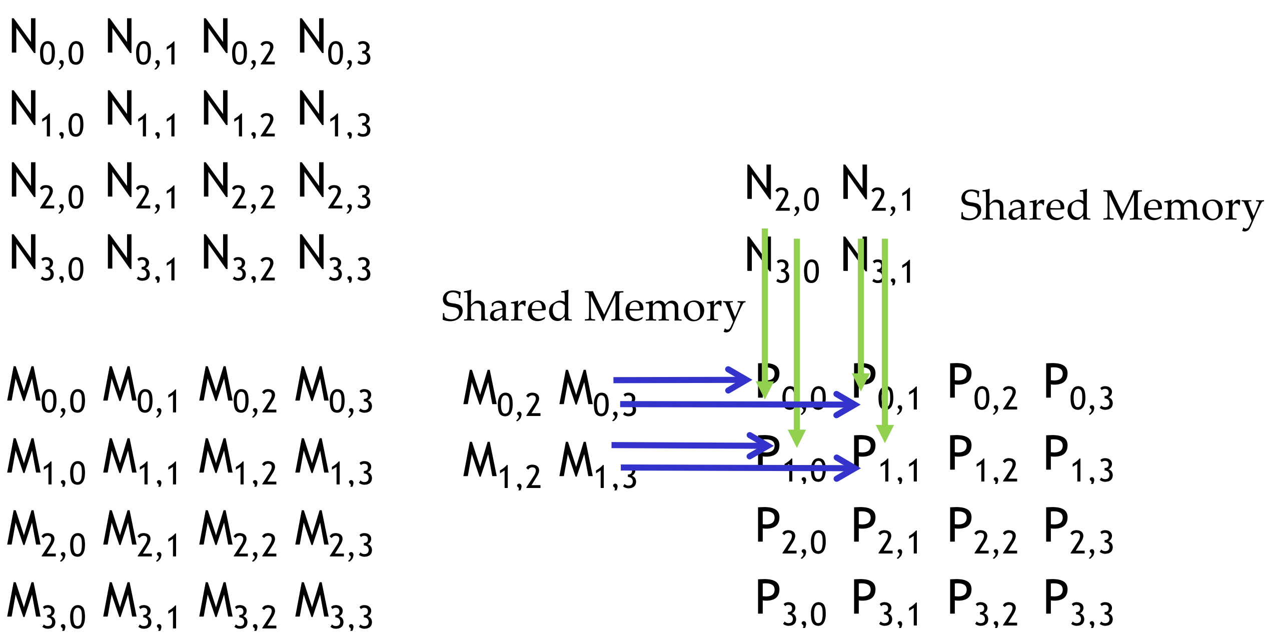
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)



Phase 1 Use for Block (0,0) (iteration 1)



Execution Phases of Toy Example

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Shared memory allows each value to be accessed by multiple threads

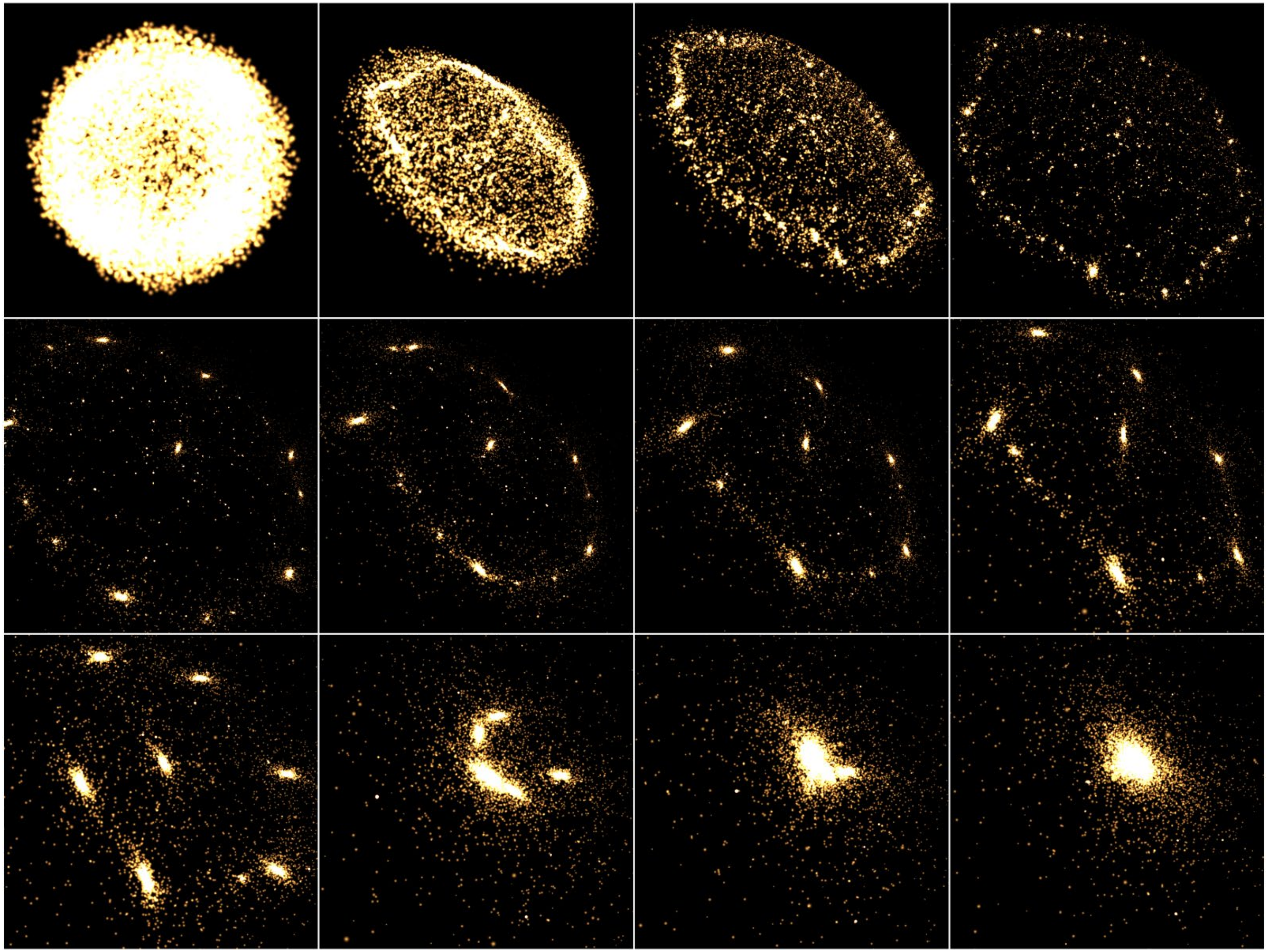
Barrier Synchronization

- Synchronize all threads in a block
  - __syncthreads()

- All threads in the same block must reach the __syncthreads() before any of the them can move on

- Best used to coordinate the phased execution tiled algorithms
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase

Example 3: all-pairs n-body computation (3D)



Force calculation

- Recall simple force calculation

$$\mathbf{F}_i = \sum_{\substack{1 \le j \le N \\ j \ne i}} \mathbf{f}_{ij} = G m_i \cdot \sum_{\substack{1 \le j \le N \\ j \ne i}} \frac{m_j \mathbf{r}_{ij}}{\left\| \mathbf{r}_{ij} \right\|^3}$$

- Softening factor $\varepsilon^2 > 0$ to limit forces

$$\mathbf{F}_i \approx G m_i \cdot \sum_{1 \le j \le N} \frac{m_j \mathbf{r}_{ij}}{\left(\left\| \mathbf{r}_{ij} \right\|^2 + \varepsilon^2 \right)^{3/2}}$$

$$\mathbf{A}_i = \frac{\mathbf{F}_i}{m_i} \approx G \cdot \sum_{1 \le j \le N} \frac{m_j \mathbf{r}_{ij}}{\left(\left\| \mathbf{r}_{ij} \right\|^2 + \varepsilon^2 \right)^{3/2}}$$

Body-body interaction

Listing 31-1. Updating Acceleration of One Body as a Result of Its Interaction with Another Body

```
__device__ float3
body_body_interaction(float4 bi, float4 bj, float3 ai)
{
  float3 r;

  // r_ij  [3 FLOPS]
  r.x = bj.x - bi.x;
  r.y = bj.y - bi.y;
  r.z = bj.z - bi.z;

  // distSqr = dot(r_ij, r_ij) + EPS^2  [6 FLOPS]
  float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;

  // invDistCube =1/distSqr^(3/2)  [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
  float distSixth = distSqr * distSqr * distSqr;
  float invDistCube = 1.0f/sqrtf(distSixth);

  // s = m_j * invDistCube [1 FLOP]
  float s = bj.w * invDistCube;

  // a_i =  a_i + s * r_ij [6 FLOPS]
  ai.x += r.x * s;
  ai.y += r.y * s;
  ai.z += r.z * s;

  return ai;
}
```

use reciprocal square root rsqrt()

20 FLOPS per interaction

Computational Tile

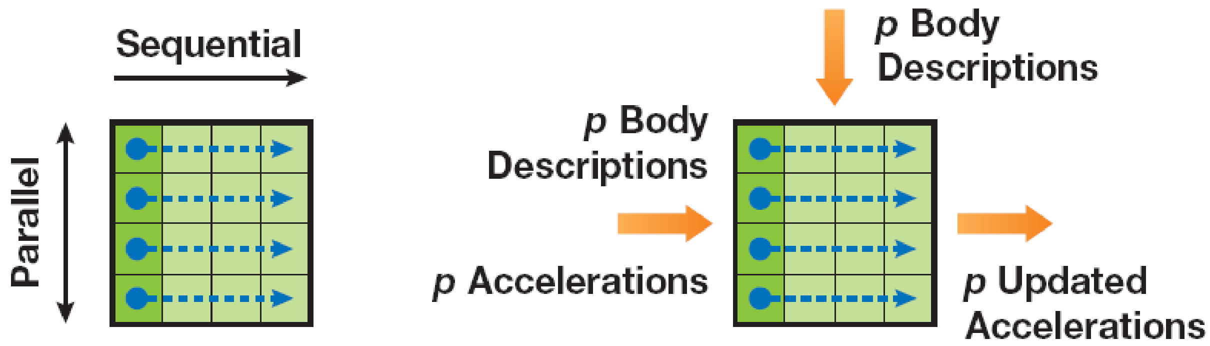
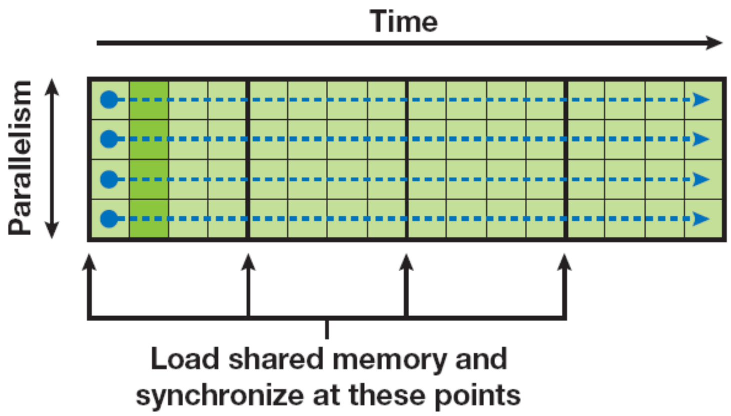


Figure 31-2. A Schematic Figure of a Computational Tile
Left: Evaluation order. Right: Inputs needed and outputs produced for the p^2 interactions calculated in the tile.

Evaluation of a single tile

```
__device__ float3
tile_calculation(float4 myPosition, float3 accel)
{
    int i;
    extern __shared__ float4[] shPosition;
    for (i = 0; i < p; i++) {
        accel = body_body_interaction(myPosition,
                                      shPosition[i], accel);
    }
    return accel;
}
```



Evaluation of all tiles in a thread block

Listing 31-3. The CUDA Kernel Executed by a Thread Block with *p* Threads to Compute the Gravitational Acceleration for *p* Bodies as a Result of All *N* Interactions

```
__global__ void
calculate_forces(void *devX, void *devA)
{
  extern __shared__ float4[] shPosition;

  float4 *globalX = (float4 *)devX;
  float4 *globalA = (float4 *)devA;
  float4 myPosition;
  int i, tile;
  float3 acc = {0.0f, 0.0f, 0.0f};
  int gtid = blockIdx.x * blockDim.x + threadIdx.x;

  myPosition = globalX[gtid];

  for (i = 0, tile = 0; i < N; i += p, tile++) {
    int idx = tile * blockDim.x + threadIdx.x;
    shPosition[threadIdx.x] = globalX[idx];
    __syncthreads();
    acc = tile_calculation(myPosition, acc);
    __syncthreads();
  }
  // Save the result in global memory for the integration step.
  float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
  globalA[gtid] = acc4;
}
```

These *p* float4 values occupy consecutive locations in device memory. The *p* loads are *coalesced* and transfer at full memory bandwidth

Execution configuration

```
// N bodies, N threads
int     p = 256;
dim3    DimBlock(p, 1, 1); // p threads per block
dim3    DimGrid(N/p, 1);   // N/p thread blocks

// p bodies in shared memory per tile evaluation
size_t SharedMemBytes = p * sizeof(Float4);

CalculateForces <<< DimGrid, DimBlock, SharedMemBytes >>>
        (Posns, Accels);
```

Performance (p = 256) GTX 8800 (2007 !)

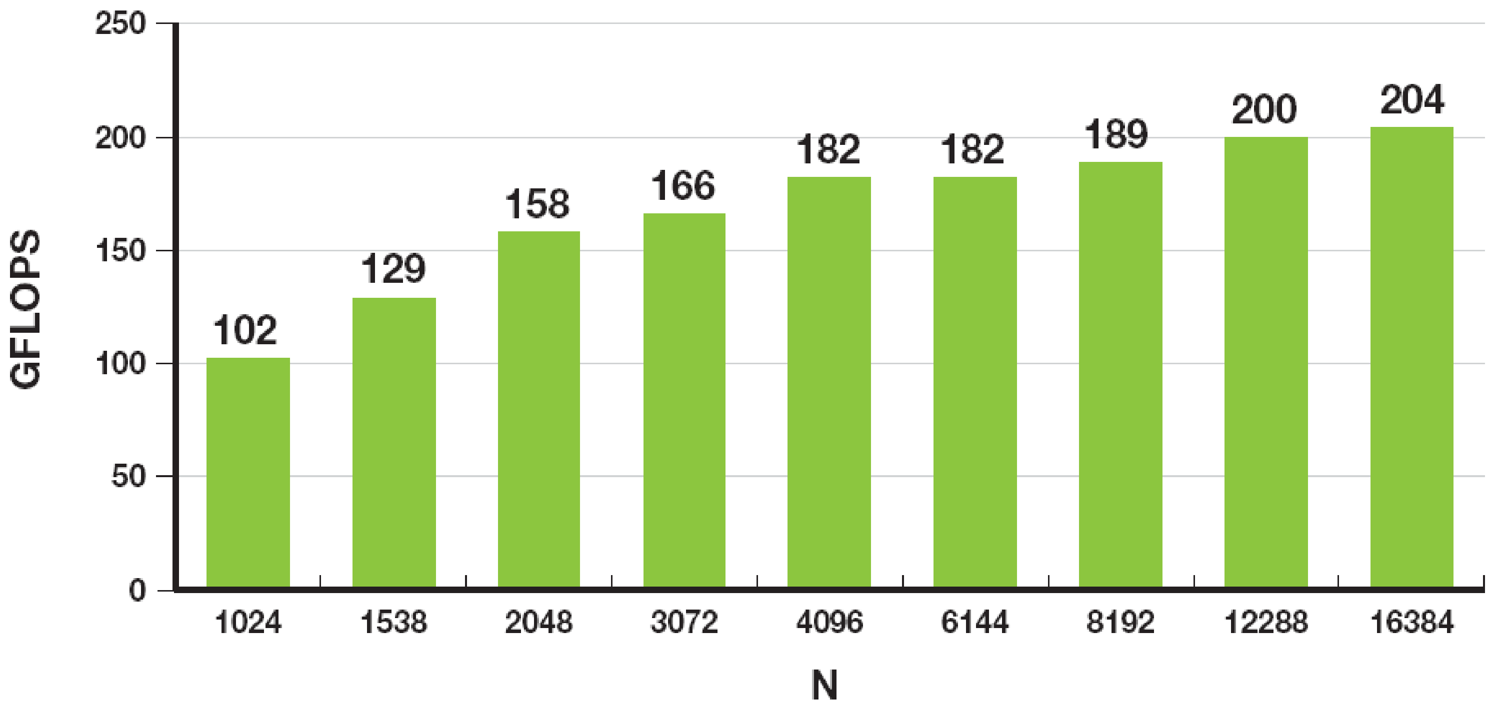


Figure 31-6. Performance Increase as *N* Grows

This is about 10B interactions/sec (single precision)

Performance (p=256) Titan V100 (2018)

- 10 timesteps

		SP		DP	
GPU	n	inter/s	GFLOPS	inter/s	GFLOPS
GTX 8800	16384	10B	200	-	-
V100	16384	314B	6300	100B	3000
V100	65536	370B	7400	135B	3900
V100	1,048,576	463B	9300	161B	4840

Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    `__global__ void KernelFunc(float* ptr)`
  - Obtained as the address of a global variable: `float* ptr = &GlobalVar;`

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

Common Runtime Component

- Provides:
  - Built-in vector types
    - `[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]`
      - Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;
int y = param.y;
```

    - `dim3`
      - Based on `uint3`
      - Used to specify dimensions
  - A subset of the C runtime library supported in both host and device codes

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- (more)
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
    - Initializes the first time a runtime function is called
    - A host thread can invoke device code on only one device
      - Multiple host threads required to run on multiple devices
  - Memory management
    - Device memory allocation
      - `cudaMalloc()`, `cudaFree()`
    - Memory copy* from host to device, device to host, device to device
      - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
    - Memory addressing*
      - `cudaGetSymbolAddress()`
  - Error handling

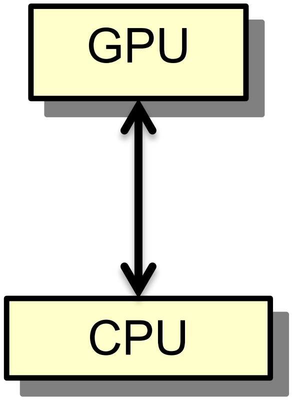
* Not needed when using unified memory model for host/device

Heterogeneous Parallel Computers

- Composed of
  - CPU(s)
    - Low-latency processor optimized for sequential execution
    - large memory size and deep memory hierarchy
  - 1-8 Accelerator(s)
    - high throughput SIMD or MIMD processors optimized for data-parallel execution
    - high-performance local memory with limited size (16-24 GB) and small depth memory hierarchy
- Example
  - Multisocket compute server
    - Host: two-socket 20 – 40 Intel Xeon cores with 128 – 512 GB CC-NUMA shared memory
    - Accelerators: 1-8 accelerators (e.g. Nvidia Cuda cards connected via PCIe x16 interfaces (16GB/s)
      - host controls data to/from accelerator memory

Basic Programming Models

- Offload model
  - idea: offload computational kernels
    - send data
    - call kernel(s)
    - retrieve data
  - accelerator-specific compiler support
    - Cuda compiler (`nvcc`) for Nvidia GPUs
  - accelerator-neutral OpenCL
    - Cuda-like notation
    - OpenCL compiler can target Nvidia or Intel Xeon Phi



Emerging Programming Models

- directive model
  - idea: identify sections of code to be compiled for accelerator(s)
    - data transfer and kernel invocation generated by compiler
  - accelerator-neutral efforts
    - OpenACC
      - ```
        #pragma acc parallel loop
           for (…) { … }
        ```
      - gang, worker, vector (threadblock, warp, warp in SIMT lockstep)
      - gcc 5, PGI, Cray, CAPS, Nvidia compilers
    - OpenMP 4.0
      - similar directives to (but more general than) OpenACC
      - implemented by gcc 4.9 and icc compiler
- accelerator-specific compiler support
  - Intel Cilk Plus and C++ compilers for Intel Xeon Phi
        ```

Scaling accelerators and interconnect

- DGX-2 (2018) 16 GPUs and 300GB/s full bisection-width interconnect