GPU Computing: A Quick Start

Orest Shardt

Department of Chemical and Materials Engineering
University of Alberta

August 25, 2011

UNIVERSITY OF ALBERTA

Session Goals

- Get you started with highly parallel LBM
- Take a practical approach
  - Less about hardware/architecture details
  - More about how to use it
  - Show performance impact of programming decisions
- Focus on what is possible with GPUs
- Detailed reference information is available on manufacturers' websites

CPUs

- Sequential operation (on each core)
  - Read memory
  - Compute
  - Store to memory

- Sometimes, need to perform exactly the **same operation** but on **different data**
- Order of execution does not matter

Pseudo-code for applying collision operator

```
for(int i = 0; i < NX; ++i)
 for(int j = 0; j < NY; ++j)
   collide(distribution[i][j]);
```

```
collide(distribution[0][0]);
collide(distribution[0][1]);
            …

collide(distribution[NX-1][NY-2]);
collide(distribution[NX-1][NY-1]);
```

GPUs for Graphics

- For 3D graphics, need to process large meshes very quickly
- You perform the **same operation** (determine visibility, compute normal vector, shade) on a lot of **different data** (triangles)
- Led to development of highly specialized “stream” processors
- Architecture was adapted for general purpose use: GPGPU

GPUs for Computing

- Parallel operation
  - Read data set from memory
  - Perform the **same computation** on many data sets **simultaneously**
  - Store results to memory

Pseudo-code for collision operator

```
for(int i = 0; i < NX; ++i)
 for(int j = 0; j < NY; ++j)
   collide(distribution[i][j]);
```

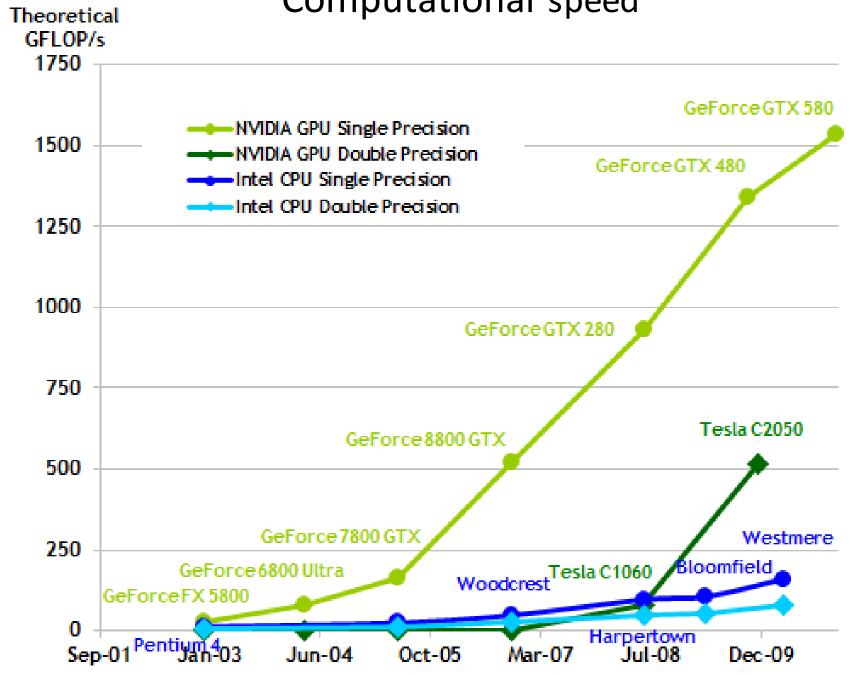
N locations processed simultaneously

```
collide(distribution[0][0]);       ——   collide(distribution[0][1]);         —//—   collide(distribution[0][N-1]);
collide(distribution[0][N]);       ——   collide(distribution[0][N+1]);       —//—   collide(distribution[0][2N-1]);
             …                                       …                         …                  …
collide(distribution[1][0]);       ——   collide(distribution[1][1]);         —//—   collide(distribution[1][N-1]);
collide(distribution[1][N]);       ——   collide(distribution[1][N+1]);       —//—   collide(distribution[1][2N-1]);
             …                                       …                                            …
collide(distribution[NX-1][0]);     —   collide(distribution[NX-1][1]);      —//—   collide(distribution[NX-1][N-1]);
             …                                       …                                            …
collide(distribution[NX-1][NY-N]); —collide(distribution[NX-1][NY-N+1]);—//—   collide(distribution[NX-1][NY-1]);
```

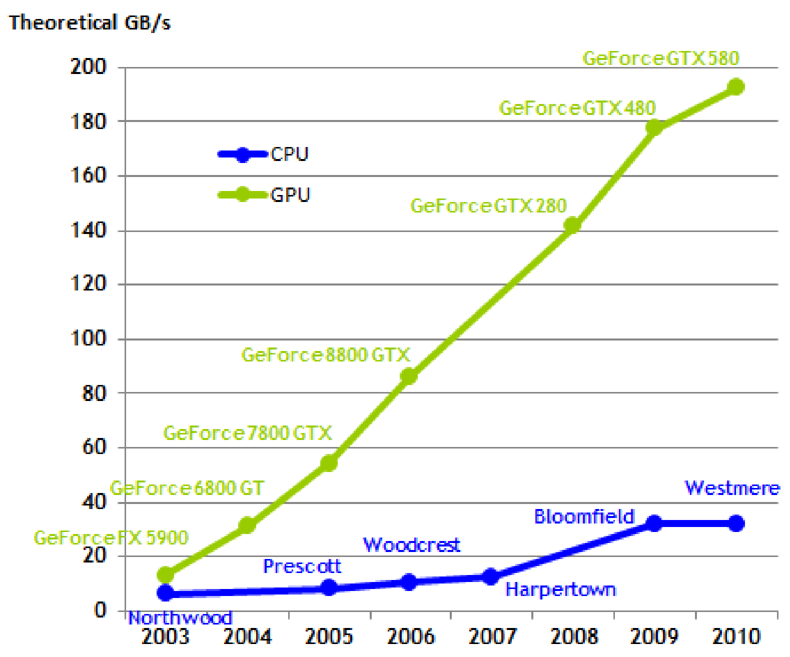
GPU Performance

- When a problem can be parallelized, the execution speed can increase dramatically
- High speed at low cost
  - GTX 580 ~$500, Intel i7 Quad Core ~$300

Computational speed



Memory transfer rate



From NVIDIA CUDA C Programming Guide

Available Architectures

- AMD/ATI
  - Radeon

RADEON GRAPHICS AMD

- NVIDIA
  - GeForce, Tesla, Quadro

NVIDIA GEFORCE WITH CUDA

UNIVERSITY OF ALBERTA

Development Environments

- Architecture Specific
  - Make use of specific hardware capabilities
  - NVIDIA: Compute Unified Device Architecture (CUDA)
  - AMD/ATI: AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream)
- General Purpose
  - Work with NVIDIA and AMD GPUs
  - OpenCL
    - standard for heterogeneous computing: CPU, FPU, GPU, embedded, etc
  - DirectCompute
    - part of Direct X for Windows

Languages

- C/C++ is best supported
- CUDA Fortran is available (Portland Group)
- OpenCL bindings are available for many languages
  - Java, Javascript, Python, C#
- GPGPU capabilities are available in other computing packages, e.g. Matlab, Mathematica

We will work with

- NVIDIA CUDA Toolkit 4.0 (May 2011)
- C/C++

NVIDIA® CUDA®

EXAMPLE

Your first kernel

- CUDA C/C++
  - All features of C/C++ are allowed in code that executes on the "host" cpu
  - Some restrictions on "device" (gpu) code
  - *.cu* source files combine host and device code

Regular includes for standard libraries

Identifies this function as device code that is called by host

Special variables available in device code

vectorAdd.cu

```
#include <stdlib.h>
#include <stdio.h>

// the kernel
__global__ void vecAddKernel(float* a, float* b, float* c)
{
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    c[i] = a[i]+b[i];

}
```

UNIVERSITY OF ALBERTA

vectorAdd.cu (continued)

no __global__ means this is host code

CUDA API functions

```
int main(int argc, char** argv)
{
    // use command-line specified CUDA device, otherwise use
    // device with highest Gflops/s
    if(cutCheckCmdLineFlag(argc,(const char**)argv, "device"))
        cutilDeviceInit(argc, argv);
    else
        cudaSetDevice(cutGetMaxGflopsDeviceId());

    int deviceId;
    // get information about the selected device
    cudaGetDevice(&deviceId);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, deviceId);

    // constants for vector size and memory size
    const unsigned int vector_size = 12*1024*1024;
    const unsigned int mem_size = sizeof(float)*vector_size;

    printf("Adding %i MB vectors\n",vector_size/(1024*1024));
    printf("Using device %i: %s\n",deviceId,deviceProp.name);
```

vectorAdd.cu (continued)

```
    // allocate host memory
    float* a_host = (float*) malloc(mem_size);
    float* b_host = (float*) malloc(mem_size);
    float* c_host = (float*) malloc(mem_size);

    // initialize the memory
    for(unsigned int i = 0; i < vector_size; ++i)
    {
        a_host[i] = 1.0f*i;
        b_host[i] = 2.0f*i;
        c_host[i] = 0.0f;
    }

    // allocate device memory
    float* a_dev;
    float* b_dev;
    float* c_dev;
    cudaMalloc((void**) &a_dev, mem_size);
    cudaMalloc((void**) &b_dev, mem_size);
    cudaMalloc((void**) &c_dev, mem_size);
```

vectorAdd.cu (continued)

```
    // create events for timing
    cudaEvent_t start, stop;
    float time,gbps;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // copy from host to device memory
    cudaMemcpy(a_dev, a_host, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b_host, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(c_dev, c_host, mem_size, cudaMemcpyHostToDevice);

    // get ready to execute kernel

    // kernel execution parameters
    unsigned int nThreads = 512;
    unsigned int nBlocks = vector_size/nThreads;

    // blocks in grid
    dim3  grid(nBlocks, 1, 1);
    // threads in block
    dim3  threads(nThreads, 1, 1);
```

vectorAdd.cu (continued)

triple angle bracket syntax for launching a kernel

```
    cudaEventRecord(start,0);              // get time before kernel starts

    // execute the kernel
    vecAddKernel<<< grid, threads >>>(a_dev, b_dev, c_dev);


    cudaEventRecord(stop,0);               // get time after kernel finishes
    cudaEventSynchronize(stop);            // wait for GPU to finish
    cudaEventElapsedTime(&time,start,stop); // compute time difference

    printf("runtime: %.4f (ms)\n", time);

    // copy result from device to host
    cudaMemcpy(c_host, c_dev, mem_size, cudaMemcpyDeviceToHost);
```

vectorAdd.cu (continued)

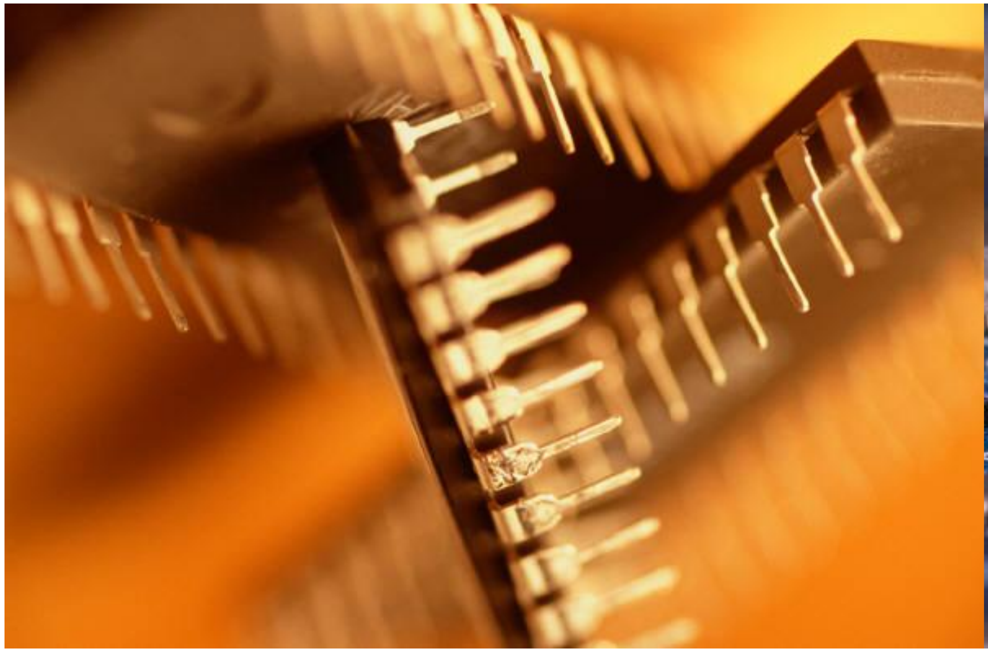
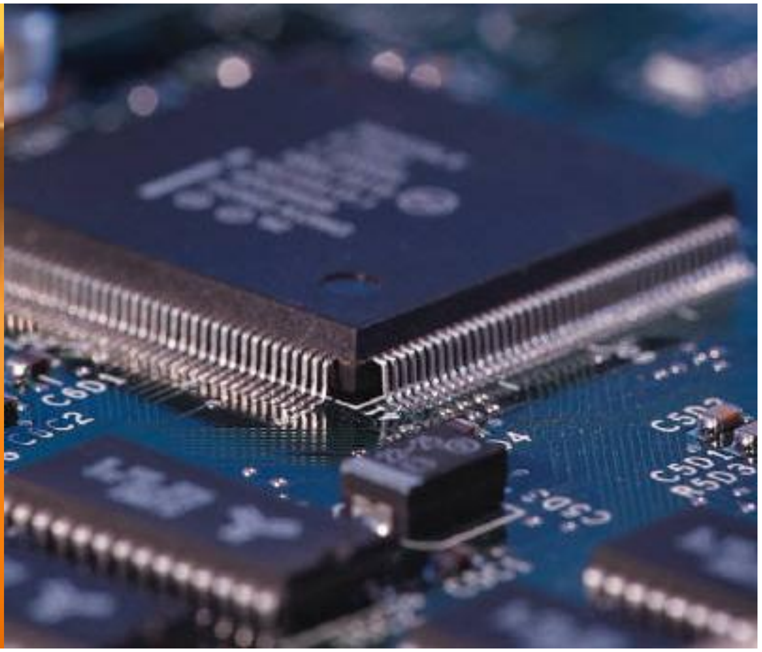
```
    // release event resources
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    // free host memory
    free(a_host);
    free(b_host);
    free(c_host);

    // free device memory
    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    cudaDeviceReset();

} // end of main()
```

Sample Results for Full Program

Compiling and running:

```
nvcc --ptxas-options=-v  -arch sm_13 -O3 -c -I $(CUDADIR)/C/common/inc  -o vecAdd.cu.o  vecAdd.cu
g++ -c -O3 vecAdd.cpp  -o vecAdd.cpp.o
g++ vecAdd.cu.o  vecAdd.cpp.o  -o vecAdd -lcudart -L$(CUDALIBDIR)  $(CUDADIR)/C/lib/libcutil_x86_64.a
./vecAdd
```

```
Adding 12 MB vectors
Using device 0: GeForce 310M

Host to device memory copy: 48.0MB in 97.6 (ms)
                 Bandwidth: 1.44 (GB/s)

vecAddKernel
        runtime: 24.6969 (ms)
        bandwidth: 5.69 (GB/s)
        Correct result: YES

Device to host memory copy: 48.0MB in 35.1 (ms)
                 Bandwidth: 1.34 (GB/s)
```

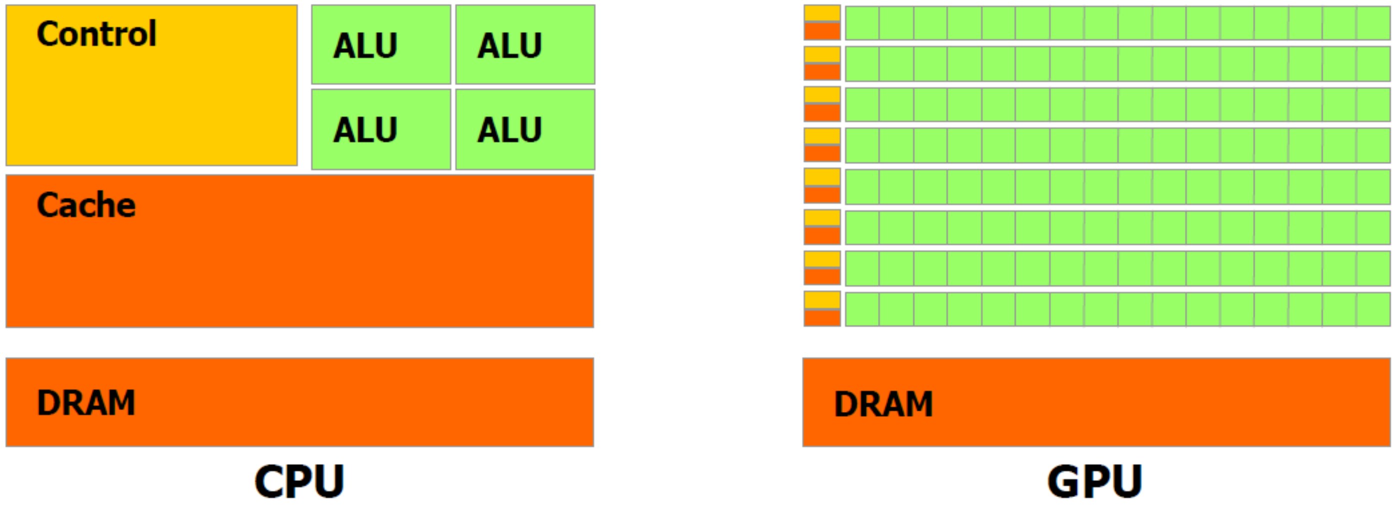



ARCHITECTURE

NVIDIA Architecture

- Stream processors
  - SIMT: Single Instruction, Multiple Thread
  - Similar to SIMD: Single Instruction, Multiple Data

Schematics of CPU and GPU organization

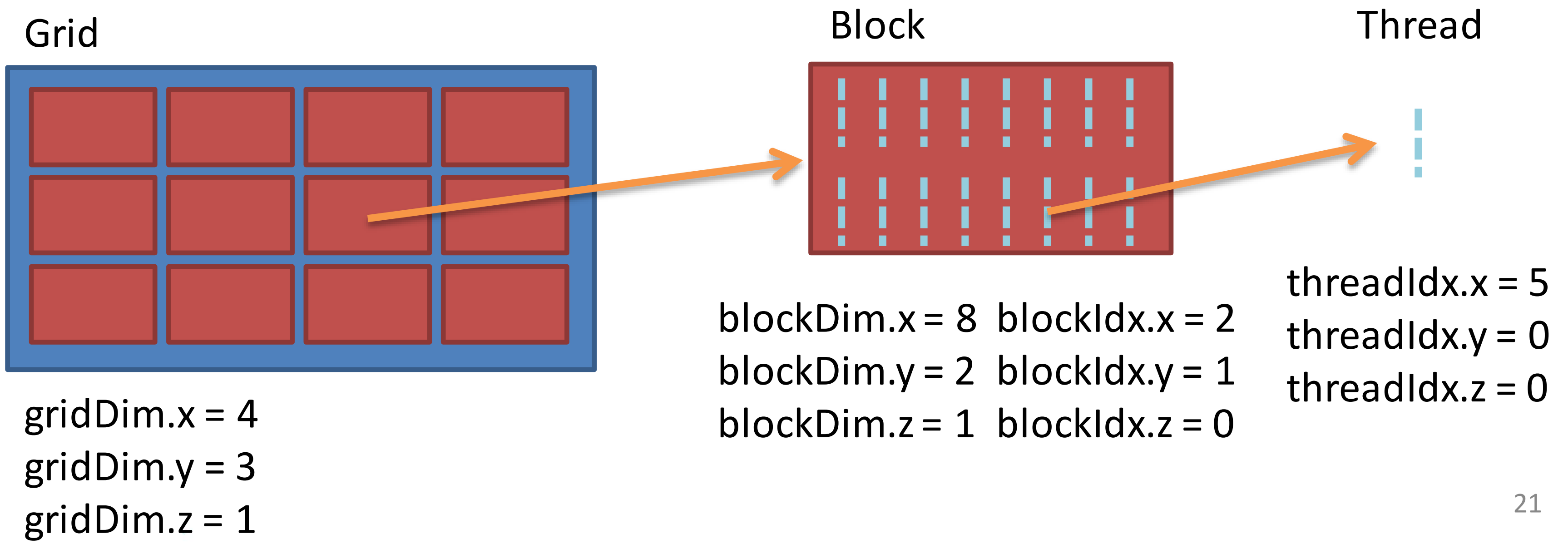


From NVIDIA CUDA C Programming Guide

Threads, Blocks, and Grids

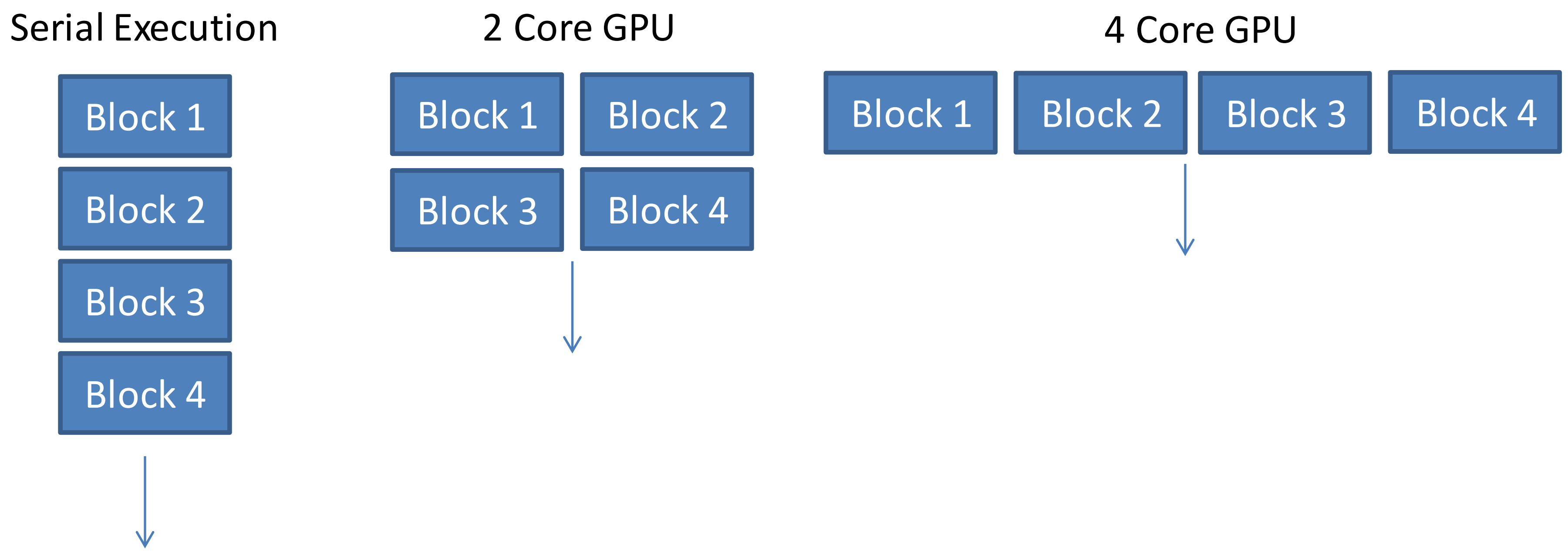
- ***Kernels*** are special functions that are executed simultaneously by many threads
- ***Threads*** are organized in a 3D arrangement called a ***block***
- ***Blocks*** are organized in a 3D ***grid***
- Blocks must be able to execute independently and in any order
- Blocks are split into ***warps*** for execution
- Special variables in device code
  - threadIdx, blockIdx, blockDim, gridDim

2D example of grid and block hierarchy



Block Execution

- Automatic parallelization
- Don’t need to think about device details
- But, still need to know resource limits…



Block and Grid Sizes

- Choosing block and grid sizes
  - Try to match problem
  - Use many threads to hide memory access delays
  - Device limits on block and grid dimensions
    - Check reference or run deviceQuery example
  - Memory limits
    - Registers: *threadsPerBlock*registersPerThread* < *maxRegistersPerBlock*
    - Shared and constant memory
    - compiler tells you: *nvcc --ptxas-options=-v*

Sample deviceQuery output

```
Total number of registers available per block: 16384
Maximum number of threads per block:           512
Maximum sizes of each dimension of a block:    512 x 512 x 64
Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
```

UNIVERSITY OF ALBERTA

Memory

- Local to a thread
  - registers (fastest)
  - managed by compiler
- Local to all threads in block
  - shared
    - used for cooperation between threads
    - managed by user
    - `__shared__`
- Visible to host and device
  - global (slowest)
  - managed by user

```
cudaError_t cudaMalloc(void **devPtr, size_t size)
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
cudaError_t cudaFree(void *devPtr)
```

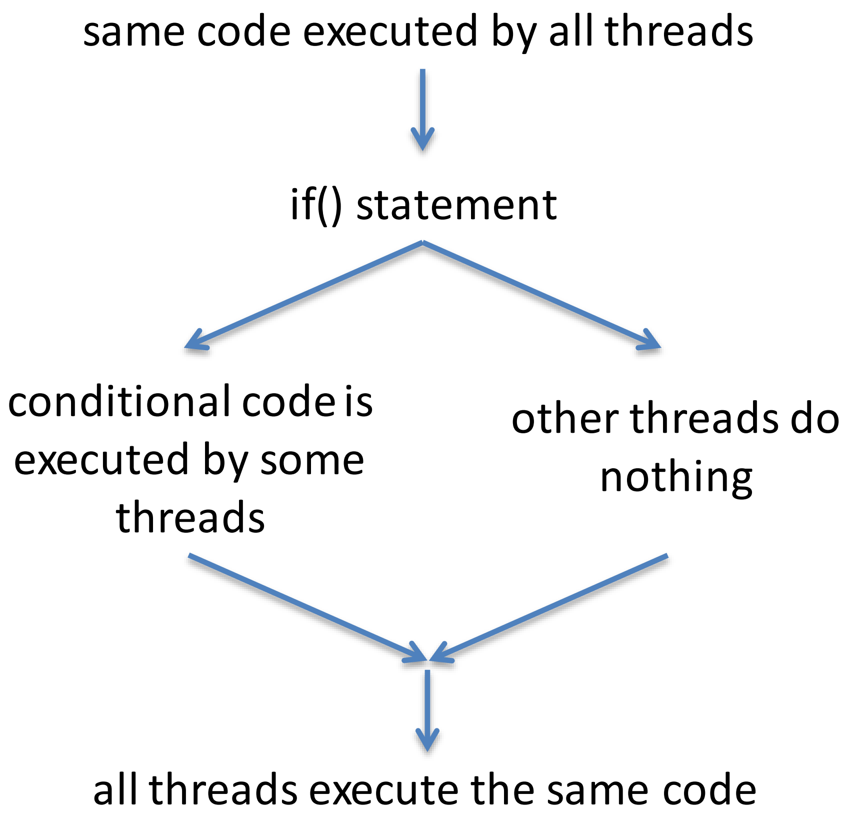
PERFORMANCE

Timing

```
// create events for timing
cudaEvent_t start, stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start,0);
// do something interesting
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
float time;

// compute time difference
cudaEventElapsedTime(&time,start,stop);
// release event resources
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

- Don't use CPU time
- If using wall time, `cudaDeviceSynchronize()`

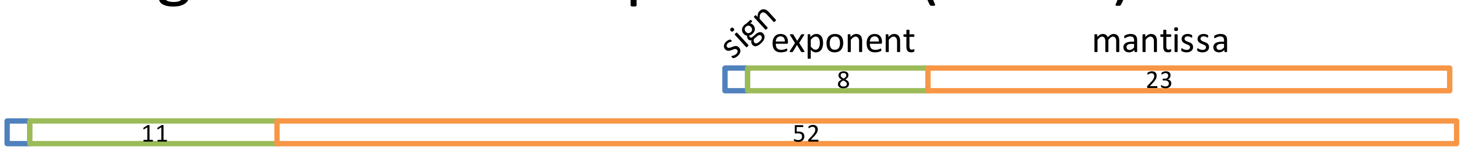
Branching


same code executed by all threads
if() statement
conditional code is executed by some threads
other threads do nothing
all threads execute the same code

- Lesson: structure code to avoid divergent execution paths in threads

Floating Point

- <1.3: only single precision (32 bit) floating point
- ≥1.3: single and double precision (64 bit)



- IEEE 754 standard defines rules for floating point computation
- Due to rounding, (A+B)+C ≠ A+(B+C)
- GPU provides a fused multiply add (FMA) instruction
  - round only once after A*B+C

Floating Point

- Compliance with IEEE 754 depends on device compute capability
- On early devices, some single precision operations do not have full single precision accuracy
- Now (capability >2.0), both single and double precision computations are IEEE compliant

- Lesson: Do not expect identical results on GPU and CPU
- Question: Which is better?
- For more information, see Whitehead and Fit-Florea, “Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs”

Memory

- **Most important optimization** for memory intensive algorithm like LBM
- This is also important in CPUs
  - Lab: we will see that poor memory use causes ~10× slow down
- Don't compare poor CPU code with good GPU code or vice versa
- General guide:
  - **CPU**: ***consecutive*** memory accesses should be nearby
  - **GPU**: ***simultaneous*** memory accesses should be nearby
- Lesson: watch memory access patterns in both CPUs and GPUs

Bandwidth

$$Theoretical\ Bandwidth = Memory\ Clock \times Bus\ width\,(bits) \times \frac{1\ byte}{8\ bits} \times 2$$

Divide by 10^9 or 1024^3 to get GB (use consistent definition)

Double Data Rate (DDR) memory:
2 transactions per clock cycle

```
Device 0: "GeForce GTX 460"
  Memory Clock rate:                                 1800.00 Mhz
  Memory Bus Width:                                  256-bit
```

$$Theoretical\ Bandwidth = 1.8 \times 10^9 s^{-1} \times 256\ bits \times \frac{1\ byte}{8\ bits} \times \frac{1\ GB}{1024^3\ bytes} \times 2$$
$$= 107.3\ GB/s$$

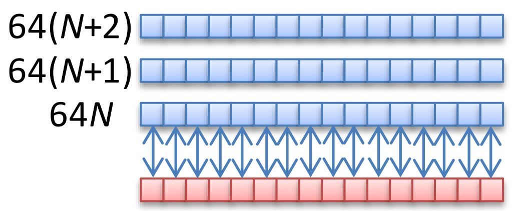
$$Effective\ Bandwidth = \frac{bytes_{read} + bytes_{written}}{time\ interval}$$

Coalesced Access

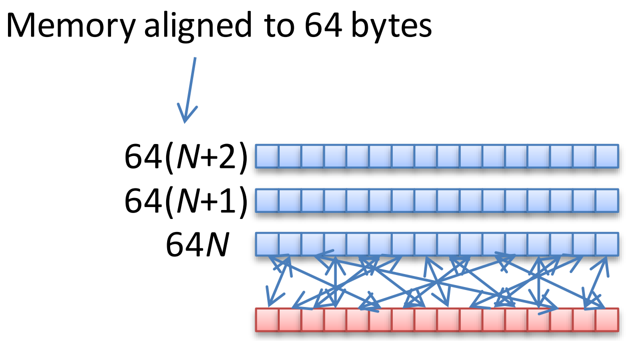
- 1.x: memory accesses in a half-warp (16 threads) are combined into one transaction if:
  - 1.0, 1.1: the *i*th thread accesses the *i*th word (32 bits) of an aligned 16 word (64 byte) segment
    - Otherwise, one transaction per thread
  - 1.2, 1.3: all threads access words (in any order) in an aligned 16 word segment
    - Threads may be combined to reduce number of transactions and issue smaller transactions
- 2.x: global memory access is cached
  - alignment and access patterns still matter, like on CPU

Memory Access Patterns

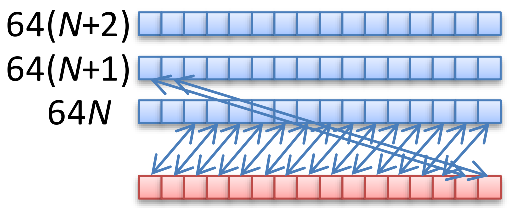
🟦 = one 32-bit word
🟥 = one thread



Result: one 64 byte transaction

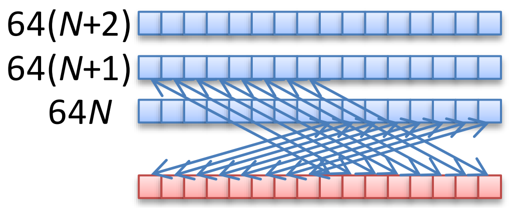


Result
- 1.0, 1.1: 16 transactions
- >1.1: one 64 byte transaction



Result
- 1.0, 1.1: 16 transactions
- >1.1: if aligned, one 128 byte transaction

otherwise, one 64 byte and one 32 byte

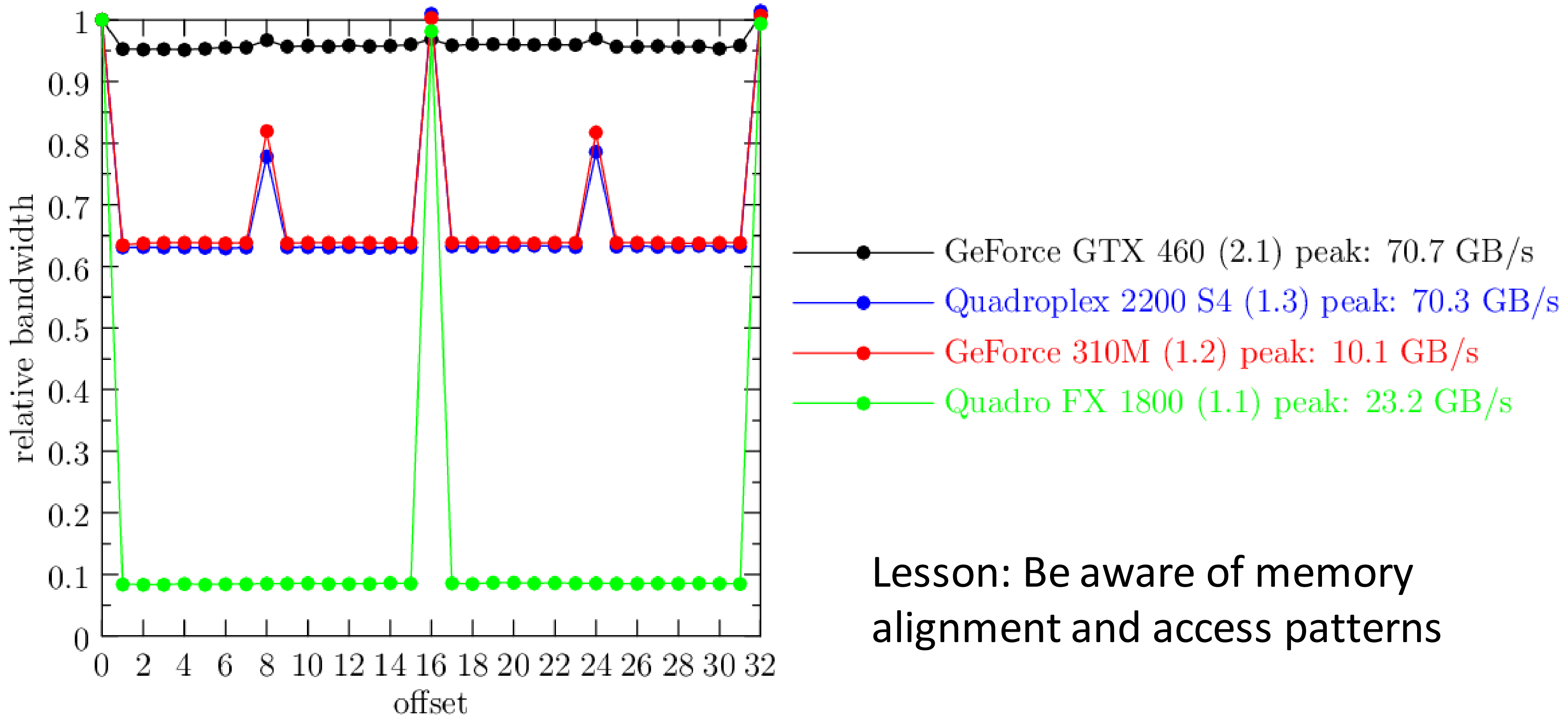


Result
- 1.0, 1.1: 16 transactions
- >1.1: if aligned, one 128 byte transaction

otherwise, two 32 byte transactions

UNIVERSITY OF ALBERTA

Non-Coalesced Access

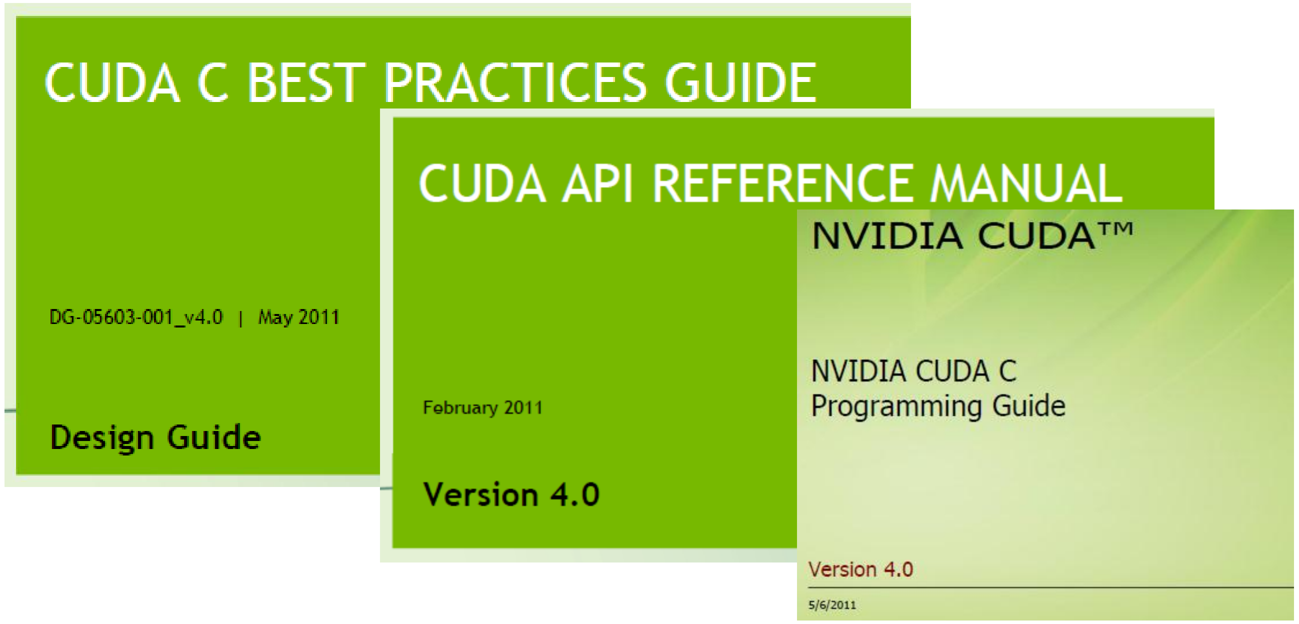


Lesson: Be aware of memory alignment and access patterns

```
__global__ void readWriteKernel(float* a, int offset)
{
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x + offset;
    a[i] = 2.0f*a[i]+1.5f;
}
```

UNIVERSITY OF ALBERTA

More Information


CUDA C BEST PRACTICES GUIDE

DG-05603-001_v4.0 | May 2011

Design Guide

CUDA API REFERENCE MANUAL

February 2011

Version 4.0

NVIDIA CUDA™

NVIDIA CUDA C Programming Guide

Version 4.0

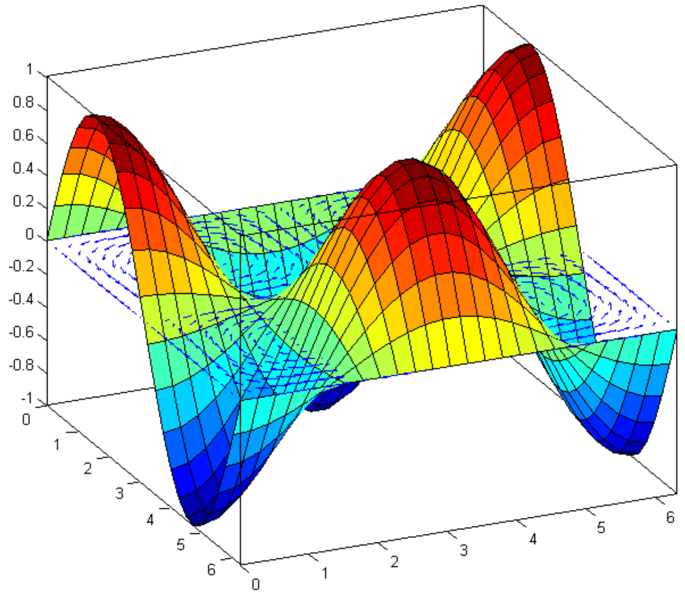
5/6/2011

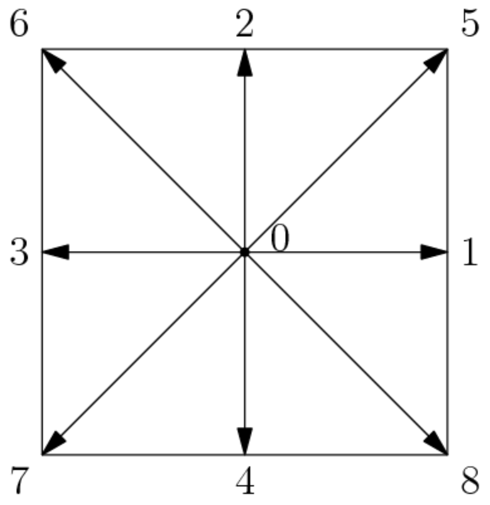
- Online

  http://developer.nvidia.com/nvidia-gpu-computing-documentation

  http://developer.nvidia.com/cuda-education-training

- Books and online tutorials are also available





LBM ON GPU

This afternoon

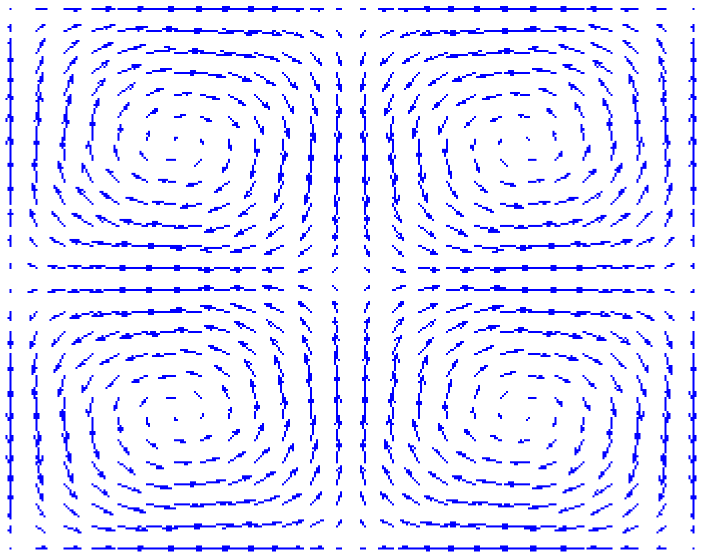
- Taylor-Green vortex decay using BGK
- Time dependent solution

$$u = u_{max} \sin(x) \cos(y)\, e^{-2\nu t}$$

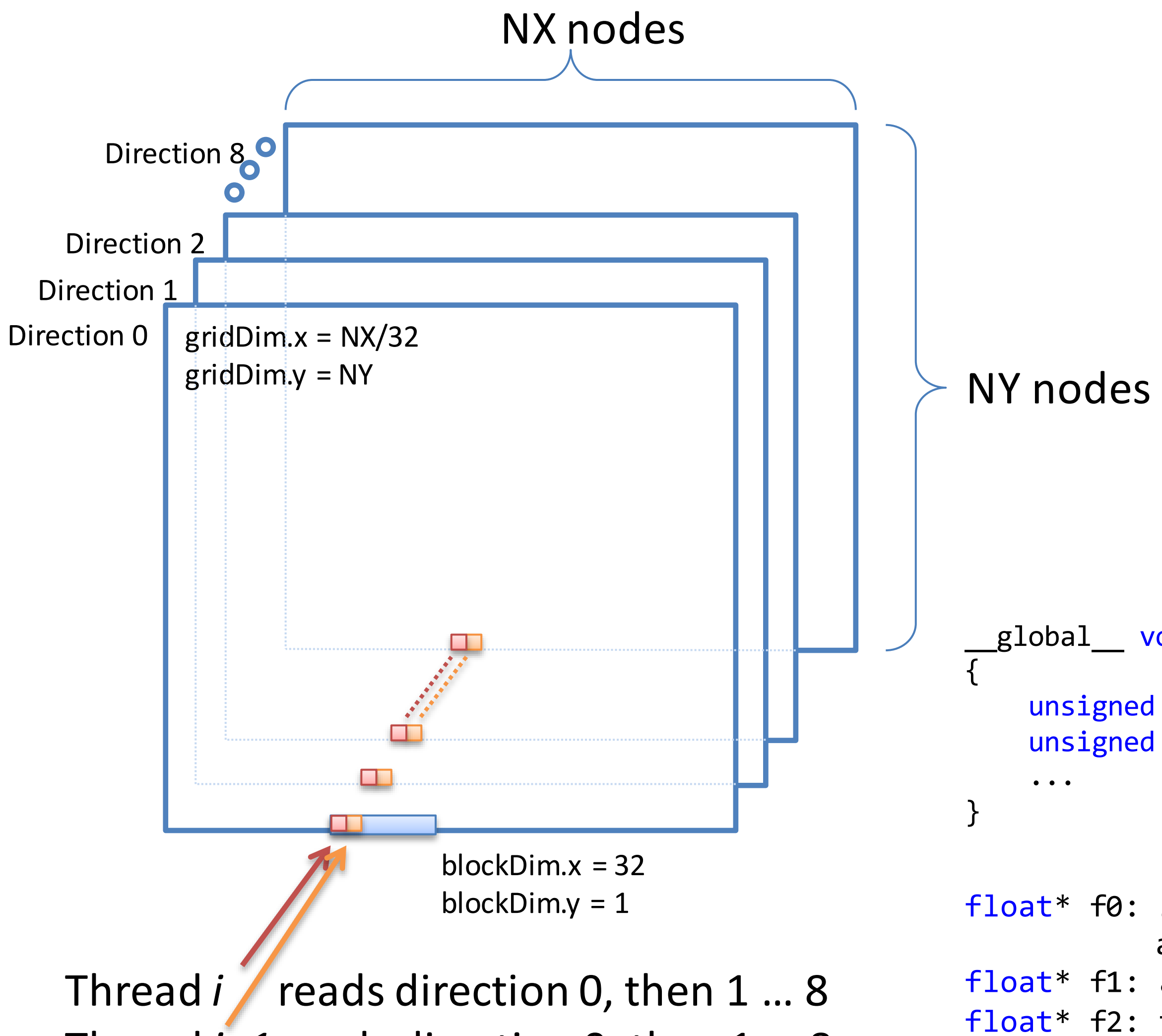
$$v = -u_{max} \cos(x) \sin(y)\, e^{-2\nu t}$$

$$P = \frac{\rho}{4} (\cos(2x) + \cos(2y)) e^{-4\nu t}$$

- Domain: $0 \leq x, y \leq 2\pi$
- Implemented as a fully periodic domain
- 3 kernels
  - Initialization
  - Collision
  - Streaming



GPU Memory Layout

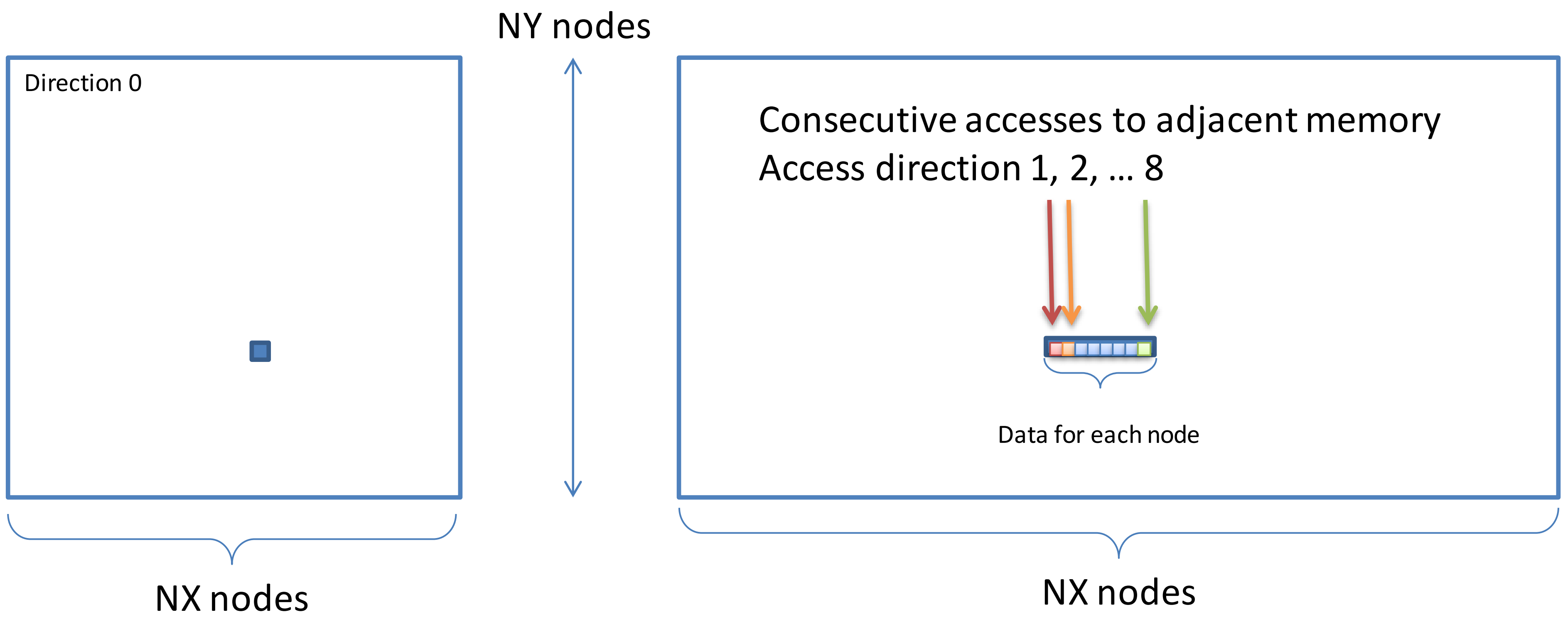


```
__global__ void gpu_collide(float* f0, float* f1)
{
    unsigned int y = blockIdx.y;
    unsigned int x = blockIdx.x*blockDim.x+threadIdx.x;
    ...
}
```

```
float* f0: separate array for direction 0
           avoids need for copying during streaming
float* f1: array for directions 1-8
float* f2: temporary array for streaming
           swapped with f1
```

Thread *i* reads direction 0, then 1 … 8
Thread *i+1* reads direction 0, then 1 … 8
Simultaneous accesses to adjacent memory

CPU Memory Layout


NY nodes
Direction 0
Consecutive accesses to adjacent memory
Access direction 1, 2, … 8
Data for each node
NX nodes
NX nodes

Separate array for direction 0 to avoid need to copy when streaming

```
extern "C" void cpu_collide(float* f0, float *f1)
{
    for(int y = 0; y < NY; ++y)
    for(int x = 0; x < NX; ++x)
    {
        ...
    }
}
```

LBM on GPU Performance

```
Simulating Taylor-Green Vortex Decay in a 1024x1024 2D domain for 8000
iterations with u_max=0.01
Using device 0: GeForce GTX 460
        Grid dimensions (32,1024,1)
        Block dimensions (32,1,1)

GPU timing information
        runtime: 20.6881 (s)
        bandwidth: 51.36 (GB/s)
        LBM speed: 405.48 (Mlups)

Device to host memory copy: 36.0MB in 32.8 (ms)
                 Bandwidth: 1.07 (GB/s)

CPU timing information
        runtime: 128.8069 (s)
        bandwidth: 8.25 (GB/s)
        LBM speed: 65.13 (Mlups)
Maximum absolute error between GPU and CPU solutions: 6.85453e-06
Maximum relative error between GPU and CPU solutions: 3.87587e-05
GPU L2 error: 2.4645e-07
CPU L2 error: 1.04671e-07
```

Compute capability 2.1

6.2 times faster than OpenMP parallelization for 6 core AMD Phenom II X6 1100T (3.3 GHz)
~**37 times faster** than a single core

Compare with 70.1 GB/s for memory access example

10.9 Mlups per core