GPU Computing with Nvidia CUDA

David Kaeli, Perhaad Mistry, Rodrigo Dominguez, Dana Schaa, Matthew Sellitto,
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA

GPU Computing Course – Lecture 2
Analogic Corp. 4/14/2011
Please make sure you join

https://groups.google.com/group/analogic-gpu-course

Mail Questions to
analogic-gpu-course@googlegroups.com
Topics – Lecture 2

- Review of Lecture 1 and introduction to GPU Computing
- Overview of GPU Architecture
- Nvidia CUDA Syntax
- Basic CUDA optimization steps
- Nvidia Fermi
- Kernel optimizations and host – device IO
- Pointers to useful CUDA tools
- Conclusions and Discussion
Motivation to study CUDA
Motivation to study CUDA

Theoretical Peaks Don’t matter Much
How do you write an application that performs well??
CPU vs GPU Architectures

Irregular data accesses
More cache + Control
Focus on per thread performance

Regular data accesses
More ALUs and massively parallel
Throughput oriented
The System

CPU (host)

GPU w/ local DRAM (device)

MCH: Memory Controller Hub
ICH: I/O Controller Hub
DDR: Double Data Rate

82925X MCH

Intel Pentium 4 Processor Extreme Edition

DDR2

DDR2

PCI Express x16 Graphics

8.0 GB/s

MDI

Intel High Definition Audio

4 PCI Express x1

150 MB/s

4 Serial ATA Ports

Intel Matrix Storage Technology

BIOS Supports HT Technology

Intel Wireless Connect Technology

NUCAR
Nvidia GPU Compute Architecture

- Compute Unified Device Architecture
- Hierarchical architecture
  - A device contains many multiprocessors
  - Many scalar “cuda cores” per multiprocessor (32 for Fermi)
  - Single instruction issue unit
- Many memory spaces
GPU Memory Architecture

- Device Memory (GDDR):
  - Large memory with a high bandwidth link to multiprocessor
- Registers on chip (~16k)
- Shared memory (on chip)
  - Shared between scalar cores
  - Low latency and banked
- Constant and texture memory
  - Read only and cached
A “Transparently” Scalable Architecture

The CUDA programming model maps easily to underlying architecture.

Same program will be scalable across devices.
Array Addition (CPU)

```c
void arrayAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}

int main() {
    int N = 4096;
    float *A = (float *)malloc(sizeof(float)*N);
    float *B = (float *)malloc(sizeof(float)*N);
    float *C = (float *)malloc(sizeof(float)*N);

    init(A); init(B);

    arrayAdd(A, B, C, N);

    free(A); free(B); free(C);
}
```
CUDA Programming – High Level View

- Initialize the GPU – done implicitly in CUDA
- Allocate Data on GPU
- Transfer data from CPU to GPU
- Decide how many threads and blocks
- Run the GPU program
- Transfer back the results from GPU to CPU
CUDA terminology

- A Kernel is the computation offloaded to GPUs
- The kernel is executed by a grid of threads
- Threads are grouped into blocks which execute independently
  - Each thread has a unique ID within the block
  - Each block has a unique ID
Array Addition (GPU)

```c
__global__
void gpuArrayAdd(float *A, float *B, float *C) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
}
```

**GRID**

**BLOCK**

<table>
<thead>
<tr>
<th>(0,0)</th>
<th>(1,0)</th>
<th>(2,0)</th>
<th>...</th>
<th>(31,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(1,0)</td>
<td>(2,0)</td>
<td>...</td>
<td>(31,0)</td>
</tr>
</tbody>
</table>

**blockDim.x = 32**

```
tid = blockIdx.x * blockDim.x + threadIdx.x
```
Vector Addition Example

CUDA allocates space in the global memory:

```c
float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, sizeof(float)*N);
cudaMalloc(&d_B, sizeof(float)*N);
cudaMalloc(&d_C, sizeof(float)*N);
```

CUDA copies from host to global memory over PCI:

```c
cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
```
Vector Addition Example

- dim3 – A 3D Vector data type which is used to pass thread and block configuration
  - Natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.
    - `dim3 dimBlock(32,1);
      dim3 dimGrid(N/32,1);
    `
- Launch Kernel Call
  - `gpuArrayAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);`
Vector Addition Example

- Initialize CUDA
- Allocate Buffers
- Copy Data
- Set Block, Grid Size
- Start Kernel
- Copy Results

☐ Read results back to host

```c
cudaMemcpy(C, d_C, sizeof(float)*N,DtoH);
```

☐ Cleanup memory and end program

☐ Our first CUDA program is finished 😊
Summary of Relevant Identifiers

Philosophy: Minimal set of extensions necessary to expose architecture

**Function qualifiers:**
- `__global__` void MyKernel() {}
- `__device__` float MyDeviceFunc() {}

**Variable qualifiers:**
- `__constant__` float MyConstantArray[32];
- `__shared__` float MySharedArray[32];

**Execution configuration:**
- dim3 dimGrid(100, 50); // 5000 thread blocks
- dim3 dimBlock(4, 8, 8); // 256 threads per block

**Kernel Launch**
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
Vector Addition (GPU)

```c
int main() {
    int N = 4096;
    float *A = (float *)malloc(sizeof(float)*N); init(A);
    float *B = (float *)malloc(sizeof(float)*N); init(B);
    float *C = (float *)malloc(sizeof(float)*N);
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, sizeof(float)*N);
    cudaMalloc(&d_B, sizeof(float)*N);
    cudaMalloc(&d_C, sizeof(float)*N);

    cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
    cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
    dim3 dimBlock(32,1);
    dim3 dimGrid(N/32,1);

    gpuArrayAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
    cudaMemcpy(C, d_C, sizeof(float)*N,DtoH);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(A); free(B); free(C);
}
```

Allocate memory on GPU

Initialize memory on GPU

Configure threads

Run kernel (on GPU)

Copy results back to CPU

Deallocate memory on GPU
Global Memory Access in GPUs

- Global memory accessed via 32, 64, or 128-byte transactions
- No of transactions depend on size of data accessed by thread and distribution of the memory addresses across the threads
- **Coalescing**: combining memory requests across threads into a single transaction

```
__global__ void bad_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[1000*tid] = threadIdx.x;
}

__global__ void good_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[tid] = threadIdx.x;
}
```
Coalescing Data Access

- Memory access requirements between threads depend on compute capability of device
- Memory accesses are handled per 16 or 32 threads
- For devices of capability 2.x, memory transactions are cached
- Data locality is exploited to reduce impact on throughput
  - **Temporal locality:** data accessed is likely to be used in future,
  - **Spatial locality:** neighboring data is also likely to be reused
- Distribution of addresses across threads to get coalescing is very inflexible for older devices (Pg 168 Progg. Guide v4.0)
Application 1: Image Rotation

- Rotate an image by a given angle
- A basic feature in image processing applications

Original Input Image

Rotated Output Image
Example 1 - Image Rotation

- A common image processing routine
  - Applications in matching, alignment, etc.
- New coordinates of \((x_1, y_1)\) when rotated by an angle \(\Theta\) around \((x_0, y_0)\)
  \[
  x_2 = \cos(\theta) \cdot (x_1 - x_0) - \sin(\theta) \cdot (y_1 - y_0) + x_0 \\
  y_2 = \sin(\theta) \cdot (x_1 - x_0) + \cos(\theta) \cdot (y_1 - y_0) + y_0
  \]
- By rotating about the origin \((0, 0)\) we get
  \[
  x_2 = \cos(\theta) \cdot (x_1) - \sin(\theta) \cdot (y_1) \\
  y_2 = \sin(\theta) \cdot (x_1) + \cos(\theta) \cdot (y_1)
  \]
Application 1: Image Rotation

- What the application does:
  - Step 1. Compute a new location according to the rotation angle (trigonometric computation)
  - Step 2. Read the pixel value of original location
  - Step 3. Write the pixel value to the new location computed at Step 1
- Create the same number of threads as the number of pixels
- Each thread takes care of moving one pixel
Image Rotation

- Input: To copy to device
  - Image (2D Matrix of floats)
  - Rotation parameters
  - Image dimensions
- Output: From device
  - Rotated Image
__global__ void transformKernel(float* g_odata, float* d_idata, int width, int height) {
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    //! We could use normalized coordinates here if we
    //! were using textures
    float u = x; float v = y; //Just a 90° flip

    int new_y = int(tv);
    int new_x = int(tu);

    g_odata[y*width + x] = d_idata[new_y * width + new_x];
}
# Implementation Steps – Hands on

- Copy image to device by enqueueing a write to a buffer on the device from the host
- Decide the work group dimensions
- Run the Image rotation kernel on input image
- We will use the provided Nvidia utilities for image handling
- Copy output image to host by enqueueing a read from a buffer on the device
- Look at Vector add for help and syntax
- `cp /sg`
Compiling CUDA - C

- c for cuda
- compile-time
- Nvidia CUDA Compiler (nvcc)
- PTX passed as data to host
- make verbose=1 for commands run
- make keep=1 for intermediate files
- execution-time
- runtime
- driver
- binary
- NUCAR
Medusa Cluster – Nvidia Subsystem

1 PCIe / S1070

8 Tesla GPUs

compute-0-8

~ 8TFlops in 3 U
Application 1: Image Rotation

- Replace ??? in the skeleton with your own CUDA code
  - Add the cudaMalloc and the cudaMemcpy calls
  - Compile with Makefile and execute

- Goals are
  - Understand how to use GPU for data parallelism
  - To know how to map threads to data
CUDA Abstractions

- Millions of lightweight threads - Simple decomposition
- Hierarchy of concurrent threads - Simple execution model
- Later we will cover:-
  - Lightweight synchronization primitives
    - Simple synchronization model
  - Shared memory model for cooperating threads
    - Simple communication model
Input vs. Output Decomposition

- Identify the data on which computations are performed
- Partition data into sub-units
  - Partition can be as per the input, output or intermediate dimensions for different computations
  - Data partitioning induces one or more decompositions of the computation into tasks e.g., by using the owner computes
- Input decomposition: Cases where we don’t know size of output (e.g. finding occurrences in a list)
- Output decomposition: Cases where more than one element of the input is required (e.g. matrix multiplication)
Application 2: Matrix Multiplication

\[ A_{HA,WA} \times B_{HB,WB} = C_{HC,WC} \]

\[
\begin{bmatrix}
  a_{0,0} \\
  a_{HA,0}
\end{bmatrix}
\times
\begin{bmatrix}
  b_{0,0} \\
  b_{HB,0}
\end{bmatrix}
=
\begin{bmatrix}
  c_{0,0} \\
  c_{HC,0}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  a_{HA,0} \\
  a_{HA,WA}
\end{bmatrix}
\times
\begin{bmatrix}
  b_{HB,0} \\
  b_{HB,WB}
\end{bmatrix}
=
\begin{bmatrix}
  c_{HC,0} \\
  a_{HC,WC}
\end{bmatrix}
\]

for (int i=0; i < HC; i++)
for (int j=0; i < WC; j++)
for (int k=0; i < WA; k++)
\[ C[i][j] += A[i][k] \times B[k][j]; \]
Application 2: Matrix Multiplication

- An O(n³) computation
- C[i][j] computed in parallel
  - An output decomposition
  - Multiple I/P elements per O/P
  - No of threads = No of elements in C
  - Each thread works independently
Matrix Multiplication Kernel

```c
__global__ void matrixMul ( float * C, float * A, float * B, int wA, int wB) {

    //! matrixMul( float* C, float* A, float* B, int wA, int wB)
    //! Each thread computes one element of C
    //! by accumulating results into Cvalue
    float Cvalue = 0;
    //! Global index of thread calculated
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int wC = wB;

    //! Each thread reads its own data from global memory
    for(int e = 0; e < wA; e++)
        Cvalue += A[row * wA + e] * B[e * wB + col];
    C[row * wC + col] = Cvalue;
}
```
Performance of Matrix Mul

- Previous implementation – Poor Scaling - Why?
  - No of operations
    - Per thread reads = (Row + Col)
    - Per thread computation = 2(Row + Col)
    - 1 Mul and 1 Add per access

- Redundant memory accesses
  - Each thread reads in whole row and whole column
  - How do we improve it? And if its this bad, why discuss it?
Matrix Multiplication Performance

Let's compare the shared memory

Matrix Mul Performance

Kernel Time (ms)

No of Elements * 1k

Using SM
Naive
Example Takeaways

- What have we learned through the two projects?
- Understood a massive parallel computing on GPU
- Experienced what CUDA programming looks like
- Understood how to decompose a simple problem
- Experienced solving problem in massively parallel fashion
Steps Porting to CUDA

- Create standalone C version
- Multi-threaded CPU version (debugging, partitioning)
- Simple CUDA version
- Optimize CUDA version for underlying hardware
- No reason why an application should have only 1 kernel
- Use the right processor for the job
Break

- GPGPU shared memory optimization
- GPGPU Block Synchronization
- Fermi Capabilities
- Page-able and Page-locked memory
- Warps and Occupancy
- Histogram64 Example
Examples have not discussed using shared memory

Critical for hiding high latency of global memory accesses

Shared memory provides almost single cycle access to data to each scalar core

- Shared memory is banked

Usage rule of thumb: coalesce frequently accessed data
Heterogeneous Apple Picking – Recap…

Trees have a very different number of apples on them?  Different pickers?
Extending Apple Picking – Again…

- Lets sell the apples in the market
- Pickers can't start pushing cart till ALL pickers have loaded their apples
  - Synchronization required within groups

Bulk-Synchronous programming models

Each cart can go to the market independently

cart ~ shared memory/ block

"It's just there as a reminder!"
Synchronization in CUDA

- Threads within block may synchronize with barriers
  ```
  ...
  __syncthreads();
  ...
  ```

- Blocks `coordinate` via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between dependent kernels (making apple juice)
  ```
  vec_minus<<<nblocs, blksize>>>(a, b, c);
  vec_dot<<<nblocs, blksize>>>(c, c);
  ```
Matrix Multiplication - Blocked

- Why look at matrix mul again?
  - Gets annoying
- Previous implementation was bad - Repetitive reads
  - Each thread worked independently
- Reuse data read by each thread
  - Inter thread-locality in access of both A and B
- Blocking is known in linear algebra for 20+ years
Matrix Multiplication - Blocked

- Shared memory optimization
- Store per-block matrices (As and Bs)
  - Shared memory is faster
- Synchronization in CUDA - Selling apple analogy
- Each thread reads in a piece of data
Matrix Multiplication - Blocked

```c
__global__ void matrixMul( float* C, float* A, float* B,
int wA, int wB)
{
int bx = blockIdx.x;   int by = blockIdx.y;
int tx = threadIdx.x;  int ty = threadIdx.y;

// Index of the first sub-matrix of A processed by the block
int aBegin = wA * BLOCK_SIZE * by;
int aEnd   = aBegin + wA - 1;
int aStep  = BLOCK_SIZE;

// Index of the first sub-matrix of B processed by the block
int bBegin = BLOCK_SIZE * bx;
int bStep  = BLOCK_SIZE * wB;

float Csub = 0;
```

Step size used to iterate through the sub-matrices of A
Step size used to iterate through the sub-matrices of B
Running Sum of result of each thread
Matrix Multiplication - Blocked

Loop over sub-matrices of A & B for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {

Declaration of the shared memory array used to store submatrix:

__shared__ float As [BLOCK_SIZE] [BLOCK_SIZE];
__shared__ float Bs [BLOCK_SIZE] [BLOCK_SIZE];

Load matrices from device to shared memory; thread loads one element:

AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

Multiply the two matrices together; each thread computes one element of the block sub-matrix:

// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}

Multiply the two matrices together; each thread computes one element of the block sub-matrix

Declaration of the shared memory array used to store submatrix

Load matrices from device to shared memory; thread loads one element

for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {

__shared__ float As [BLOCK_SIZE] [BLOCK_SIZE];
__shared__ float Bs [BLOCK_SIZE] [BLOCK_SIZE];

AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}
Matrix Multiplication - Blocked

for (int a = aBegin, b = bBegin; a <= aEnd; 
a += aStep, b += bStep) {

__shared__ float As [BLOCK_SIZE] [BLOCK_SIZE];
__shared__ float Bs [BLOCK_SIZE] [BLOCK_SIZE];

AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

__syncthreads();

for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

// Write the block sub-matrix to device memory;
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
__syncthreads();
}
Application 2: Matrix Multiplication

- Hands-on performance comparison
- For a MxN matrix
  - Count no of global reads / thread
  - Count no of global writes / thread
- Compare blocking vs non blocking performance
- You can use the CUDA visual profiler later to count the number of memory accesses.
  - Note: they may not be the same because of coalescing
Matrix Multiplication Performance

- Lets compare the shared memory

Matrix Mul Performance

![Graph showing Matrix Mul Performance](image)

- Using SM
- Naive
Textures and Images

- Textures are allocated in global memory and cached.
  - Cache size ~6-8KB per mp,
  - Optimized for 2D locality in accesses

- Constant memory is also cached

- Use to optimize the image rotation example
  - Uncoalesced reads from global memory
Hands On – Try simpletexture

- Defined at file scope as a type texture:
  
  ```
  texture<Type, Dim, ReadMode> mytex;
  ```

- Textures are referenced using floating-point coordinates in the range [0, N) or if normalized [0,1.0).

- Addressing mode can be
  - Clamped, 1.25 -> 1.0 in [0,1.0) or
  - Wrapped, eg 1.25 -> 0.25

- Value returned can be a single element or a interpolated value
Warp and Occupancy

- Multiprocessor creates and executes threads in groups of 32 parallel threads called warps.
- Threads in a warp start at the same program address
  - Have individual instruction and register state
  - Free to branch and execute independently
- Enables more applications (See Histogram256)
Using the Occupancy Calculator

- The fact that all instructions in a warp execute together in lock step can be used to our advantage
- **NOTE:** Warps are not part of the CUDA language definition
- Cost of warp divergence = sum of if + sum of else block
- Occupancy is the ratio of active warps to the maximum number supported on a multiprocessor of the GPU
- Determines how efficient the kernel will be on the GPU.
- Get statistics for occupancy calculator with `make keep=1`
Using the Occupancy Calculator

CUDA GPU Occupancy Calculator

1. Select a GPU from the list (click)
2. Enter your resource usage:
   - Threads Per Block
   - Warps Per Block
   - Shared Memory Per Block (in bytes)

3. GPU Occupancy data is displayed here and in the graphs:
   - Active Threads per Multiprocessor
   - Active Warps per Multiprocessor
   - Occupancy of each Multiprocessor
   - Maximum Simultaneous Blocks per GPU

(Your chosen resource usage is indicated by the red triangle in the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.)

---

NUCAR
Occupancy Tradeoffs

- Occupancy is an empirical measure
  - A last order optimization step and device dependent
- More threads / block
  - Benefits – Helps compute bound workloads (rare for GPUs)
  - Drawbacks – Reduces number of registers per thread and shared memory per block, less blocks to hide latency
- Optimum threads / block
  - IO bound workload has just enough warps to switch with
Experiment with Occupancy

- Download excel file from course web page
- Occupancy is not a performance counter, it is simply a ratio
- Try with non blocking and blocking matrix multiplication
  - Choose one data set
    - Note: press ‘0’ when verification is not needed
  - Vary number of threads per block
End – Class II
Note: The Next lecture should be covering material below
Nvidia Fermi

- Compute 2.0 / 2.1 devices
- Better double precision
- ECC support
- Configurable cache hierarchy
- Faster context switching
- Faster atomic operations
- Concurrent kernel execution
- Dual DMA Engines
Nvidia Fermi Features

- Everything discussed till now is still relevant 😊
- ECC support - Data-sensitive applications
- Configurable Cache Hierarchy
  - Implementations unable to use shared memory
- Faster Context Switching
  - Application graphics and compute interoperation
Concurrent Kernel Execution

- Concurrent Kernel Operation - Enables smaller data sets

Requires knowledge of CUDA Stream API
More than enough rope provided to hang yourself
Eowyn – Fermi System

- My personal system at NEU
  - Dell XPS Gaming Platform
  - GTX-480

PCI Bus
Host – Device Interaction

- An application dependent optimization space
- Page-locked Memory
- Asynchronous host – device Application IO
  - Used commonly in medical imaging where data is continuously fed to device
  - Use CUDA stream’s asynchronous API
- Divide application into multiple kernels and keep data on device
  - This often means coding non data parallel or inefficient kernels to avoid IO
Pinned Memory Optimization

- Page-able vs. Page-locked memory
- Locked pages will not be swapped out to disk by the OS
- Allocate using cudamallochost
- Fermi + CUDA 4.0 provides non-copy pinning

Note: excess page locking affects system performance
Performance of Page-locked Memory

Tested using CUDA SDK
e example bandwidth test
Performance of Page-locked Memory

Host - Device IO (Fermi)

Tested using CUDA SDK
example bandwidth test
Device Query & Bandwidth Test

Useful tools to check your setup configuration and learn about device
Application: Histogram64

- 64 bin histogram of data
  - Build per thread subhistogram
  - Build per block sub histogram

- Homework: Try Histogram256 using local memory atomics

```c
for (int i = 0; i < BIN_COUNT; i++)
    result[i] = 0;
for (int i = 0; i < dataN; i++)
    result[data[i]]++;
```
Implementation of Histogram

- **Kernel 1:** Build per block histogram from per thread histogram
  - Per thread histogram in shared memory
  - Reduce to block histogram

- **Kernel 2:** Combine block histograms into final histogram
Histogram64 Kernel1

- Main Implementation Steps:
  - Initialization of shared memory to 0 is important
  - Make per thread histogram
  - Use 64 threads per block to aggregate per thread into a per-block histogram

- Note: Synchronization after per thread histograms is made
- Also use short data types for the thread histograms
- Later optimization step done in CUDA SDK to remove bank conflicts is left for future discussion
Optimizations in Histogram64

- A simplified version of the Histogram64 kernel is provided
- Optimizations Include
  - Using shared memory
    - Build per block histogram using data gathered by each thread
  - Group 8 bit reads into a 32 bit read
    - As discussed coalescing: needs 32 bit transactions atleast
  - Provided implementation includes bank conflicts in shared memory
Summary

- We have studied the architecture of CUDA capable Nvidia GPUs
- We have seen the basics of CUDA and the relationship between the architecture and the programming model
- We have decomposed a data parallel algorithm
- We have used different architectural features of the GPU like shared and texture memory
Summary

- We have optimized host-device interaction using pinned memory.
- CUDA is a powerful parallel programming model:
  - Heterogeneous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible - minimal but expressive changes to C
  - Interoperable - simple graphics interop mechanisms
Summarizing Today’s Programming

- Array addition, Devicequery and BandwidthTest: Basic CUDA programming, host - device code

- Image Rotation:
  - Flipping: 2D Data Mapping
  - Image rotation extension: using texture memory

- Matrix Multiplication:
  - Naïve: Blocks and threads, coalescing data reads
  - Blocking: Using Shared memory and synchronization in blocks

- Histogram64: Using shared memory to buffer data
Nvidia - CUDA Ecosystem - Today

- Libraries: FFT, BLAS, ...
  Example Source Code
- Integrated CPU
  and GPU C Source Code
- NVIDIA C Compiler
- NVIDIA Assembly
  for Computing
- CPU Host Code
- CUDA Driver
- Debugger Profiler
- Standard C Compiler
- GPU
- CPU

Beyond Programmable Shading: In Action
Productivity Tools Based on CUDA

- **Thrust** - A STL – like library for CUDA
- Linear Algebra and Mathematical Routines
  - CUBLAS and CURAND
  - MAGMA and CULA-Tools provide LAPACK
  - CUSP – CUDA Sparse Algebra
  - CUFFT – FFTW for GPUs
  - NPP: Performance Primitives – Video processing
  - Sections of **OpenCV**

**green = Nvidia product**
**bold = open source**
# Programming Tools for CUDA

<table>
<thead>
<tr>
<th>Solution</th>
<th>Approach</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA C Runtime</td>
<td>Language Integration</td>
<td>NVIDIA CUDA Toolkit</td>
</tr>
<tr>
<td>Fortran</td>
<td>Auto Parallelization</td>
<td>PGI Accelerator</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Device-Level API</td>
<td>Khronos standard</td>
</tr>
<tr>
<td>DirectCompute</td>
<td>Device-Level API</td>
<td>Microsoft</td>
</tr>
<tr>
<td>PyCUDA</td>
<td>API Bindings</td>
<td>Open source</td>
</tr>
<tr>
<td>jCUDA</td>
<td>API Bindings</td>
<td>Freely Available</td>
</tr>
<tr>
<td>CUDA.NET</td>
<td>API Bindings</td>
<td>Freely Available</td>
</tr>
<tr>
<td>OpenCL.NET</td>
<td>API Bindings</td>
<td>Freely Available</td>
</tr>
</tbody>
</table>
Next Class (4/28)

- More advanced CUDA
  - Performance Tools – Using the CUDA Visual Profiler
  - Debugging Techniques – Using cuda-gdb

- Let us know any particular areas of focus you would like
  - Look at the SDK examples for topics you are interested in
More information and References

- NVIDIA GPU Computing Developer Home Page

- CUDA Download

- Programming Massively Parallel Processors: A Hands-on Approach, David B. Kirk and Wen-mei W. Hwu

- Other resources
  - http://courses.engr.illinois.edu/ece498/al/
More information and References

- Beyond Programmable Shading – David Leubke
- Decomposition Techniques for Parallel Programming – Vivek Sarkar
- CUDA Textures & Image Registration - Richard Ansorge
- Setting up CUDA within Windows Visual Studio
- SDK examples: Histogram64, Matmul, SimpleTextures
Thank You!
Questions, Comments?

Perhaad Mistry
pmistry@ece.neu.edu