MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

CUDA Computing

Klaus Mueller, Wei Xu, Ziyi Zheng, Fang Xu

Computer Science
Center for Visual Computing
Stony Brook University, NY

Siemens USA
Research
Princeton, NJ

Outline

Goal: To develop medical imaging applications in CUDA environment

- CUDA Hardware
- CUDA Programming API
- CUDA Graphics API
- CUDA Performance

Setup CUDA

Compute Unified Device Architecture
- Driver, Toolkit and SDK  http://www.nvidia.com/object/cuda_get.html

Other resource
- Cuda-dgb beta 2.1
- CUDA occupancy calculator
- Visual studio syntax highlighting
- Template wizard

CUDA Hardware

CUDA Hardware
- Host & Device
- CG Model & CUDA Model
- Thread Hierarchy
- Memory Hierarchy

--Know your weapon.
**Hardware Architecture**

**GPU**
- Compute-intensive
- Highly data parallel

**CUDA**
- Expose the parallel capabilities of GPUs.

---

**Host & Device**

**Host (CPU)**
- Program flow
- Thread management
- Load GPU programs (kernels)

**Device (GPU)**
- Load data
- Perform computations

Heterogeneous Programming

---

**CG Model**

Collect

<table>
<thead>
<tr>
<th>Control</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>...</th>
<th>Control</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>...</td>
</tr>
</tbody>
</table>

Can not scatter!

Threads can not cooperate! \(\rightarrow\) Multi-pass render

---

**CUDA Model**

Collect

<table>
<thead>
<tr>
<th>Control</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>...</th>
<th>Control</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>...</td>
</tr>
</tbody>
</table>

Scatter

<table>
<thead>
<tr>
<th>Control</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>...</th>
<th>Control</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>...</td>
</tr>
</tbody>
</table>

Threads can cooperate!
Decomposition

Can all threads cooperate?
- NO

Cooperate with a smaller batch of threads (block)
- Same multiprocessor
- Shared memory
- Highlight of CUDA computing

CUDA Tips From NVidia
1. Decompose program into a sequence of steps (Grids)
2. Decompose grid into independent parallel blocks (Thread blocks)
3. Decompose block into cooperating parallel elements (Threads)

Examples
- Adding vector
- Sobel filter

Memory Model

Memory Hierarchy
- Simply start by using just global memory
- Then optimize
- More about this later
### CUDA vs. CG

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>General purpose</td>
<td>Graphics</td>
</tr>
<tr>
<td>Program</td>
<td>Kernel</td>
<td>Shader</td>
</tr>
<tr>
<td>Collect</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Scatter</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Thread Synchronization</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Memory</td>
<td>Local, Shared, Constant, Global, Texture</td>
<td>Texture</td>
</tr>
</tbody>
</table>

### CUDA Programming API

#### Function Qualifiers

Device Global, & Host
- To specify whether a function executes on the host or device
- __device__ must return void

<table>
<thead>
<tr>
<th>Function</th>
<th>Exe on</th>
<th>Call from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>GPU</td>
<td>GPU</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>CPU</td>
<td>CPU</td>
</tr>
</tbody>
</table>

#### Variable Qualifiers

Shared, Device & Constant
- To specify the memory location on the device of a variable
- __shared__ and __constant__ are optionally used together with __device__

```
__global__ void Func(float* parameter);
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shared</strong></td>
<td>Shared</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>

```
__constant__ float ConstantArray[16];
__shared__ float SharedArray[16];
__device__ ....
```
Execution Configuration

<<< Grids, Blocks>>>
- Kernel function must specify the number of threads for each call (dim3)

<<< Grids, Blocks, Shared>>>
- It can specify the number of bytes in shared memory that is dynamically allocated per block (size_t)

```cpp
dim3 dimBlock(8, 8, 2);
dim3 dimGrid(10, 10, 1);
KernelFunc<<<dimGrid, dimBlock>>>(...);
KernelFunc<<<100, 128>>>(...);
```

Device Side Parameters

Threads get parameters from execution configuration
- Dimensions of the grid in blocks
  ```cpp
dim3 gridDim;  
```
- Dimensions of the block in threads
  ```cpp
dim3 blockDim;  
```
- Block index within the grid
  ```cpp
dim3 blockIdx;  
```
- Thread index within the block
  ```cpp
dim3 threadIdx;  
```

Example 1: Adding Matrix

void AddMatrix(int *A, int *B, int *C, int w, int d) {
    for (int j = 0; j < d; j++)
        for (int i = 0; i < w; i++)
            C[j * w + i] = A[j * w + i] + B[j * w + i];
}

CUDA C++

```cpp
void AddMatrix(int *A, int *B, int *C, int w, int d) {
    for (int j = 0; j < d; j++)
        for (int i = 0; i < w; i++)
            C[j * w + i] = A[j * w + i] + B[j * w + i];
}
```

Memory Management

Device memory allocation
- cudaMalloc(), cudaFree()

Memory copy
- cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()

Memory addressing
- cudaGetSymbolAddress()
Example 1: Adding Matrix

```c
void * a,*b,*c;
cudaMalloc((void**)&a, w*d*sizeof(int));
cudaMalloc((void**)&b, w*d*sizeof(int));
cudaMalloc((void**)&c, w*d*sizeof(int));
...//load data into a, b;
cudaMemcpy(a, w*d*sizeof(int),cudaMemcpyHostToDevice);
Launch kernel
AddMatrix <<<dimGrid, dimBlock>>> (a,b,c,w );
Get threadIdx, blockIdx here
__global__ void AddMatrix (int *A, int *B, int *C, int w)
{
    int i = blockIdx.x *blockDim.x + threadIdx.x;
    int j = blockIdx.y *blockDim.y + threadIdx.y;
    int id = j*w +i;
    C[id]=A[id]+B[id];
}
Copy result
cudaMemcpy(c, w*d*sizeof(int),cudaMemcpyDeviceToHost);
Free memory
cudaFree(a); cudaFree(b); cudaFree(c);

void __syncthreads();
```

Parallel and Synchronize

Threads execute in asynchronous manner in general
- Threads with one block share memory and can synchronize

```
void __syncthreads();
```
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- No such function in CG. CG can do this by multi-pass render

CUDA Graphics API

- Texture (1D 2D 3D)
- PBO (Pixel Buffer Object)
- FBO (Frame Buffer Object)

--Go back to our goal

Texture Memory Advantage

Texture fetch versus global or constant memory read
- Cached, better performance if fetch with locality
- Not subject to the memory coalescing constraint for global and constant memory
- 2D address
- Filtering
- Normalized coordinates
- Handling boundary address
Texture

1. Declaring texture reference, format and cudaArray

   texture<Type, Dim, ReadMode> texRef; cudaArray* cu_array;
   cudaChannelFormatDesc cudaCreateChannelDesc<T>();

<table>
<thead>
<tr>
<th>Feature</th>
<th>Use</th>
<th>Caveat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filtering</td>
<td>Fast, low-precision interpolation</td>
<td>Only valid if the texture reference returns float-point data</td>
</tr>
<tr>
<td>Normalized coordinates</td>
<td>Resolution-independent</td>
<td></td>
</tr>
<tr>
<td>Addressing modes</td>
<td>Handling boundary</td>
<td>Normalized texture only</td>
</tr>
</tbody>
</table>

2. Memory management

cudaMallocArray(), cudaFreeArray(), cudaMemcpyToArray()

3. Bind/Unbind texture before/after texture fetching

cudaBindTextureToArray(), cudaUnbindTexture()

Example 2: Texture Example

texture<unsigned char, 2, cudaReadModeElementType> texr;
...

cudaChannelFormatDesc chDesc = cudaCreateChannelDesc<unsigned char>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, chDesc, w, h);
cudaMemcpyToArray(cuArray, 0, 0, input, data_size, cudaMemcpyHostToDevice);
cudaBindTextureToArray(texr, cuArray);
...

//launch kernel. Inside kernel, use tex2d(texr, idxX, idxY);
...//read result from device
cudaUnbindTexture(texr);
cudaFreeArray(cuArray);
cudaFree(data);

Example 3: PBO in OPENGL

GLuint pbo;
glGenBuffersARB(1, &pbo);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB, width*height*sizeof(GLubyte) *4, 0, GL_STREAM_DRAW_ARB);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
cudaGLRegisterBufferObject(pbo);
...

cudaGLMapBufferObject((void**)&d_output, pbo);
//launch kernel
..
cudaGLUnmapBufferObject(pbo);
cudaGLUnregisterBufferObject(pbo);
Example 4: X Ray Rendering

- Direct volume rendering
- 3D texture only
- Pixel Buffer Object (PBO)
- Ray-casting
- Bounding volume
- No shared memory. Similar to CG.

```
// calculate eye ray in world space
// find intersection with box
...
float4 sum = make_float4(0.0f);
float t = far;
for(int i=0; i<maxSteps; i++) {
    float3 pos = eyeRay.o + eyeRay.d*t;
pos = pos*0.5f+0.5f;
    float sample = tex3D(tex, pos.x, pos.y, pos.z);
sample *= transferScale;
    float4 col = make_float4(sample,sample,sample,1.0);
    sum+=col;
t -= tstep;
if (t < near) break;
}
```

Accumulating color

CUDA Performance

- Instruction Optimization
- Global Memory Coalescing
- Shared Memory Bank Conflicts

--Save time, save lives

Instruction Optimization

Compiling with "-usefastmath"

Single or double precision

Unrolling loops

- Overhead by loop/branching is relatively high

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++) {
    sum += data[sharMemPos + k] * d_Kernel[KERNEL_RADIUS - k];
}
```

- Results in 2-fold performance increase

```
    + data[sharMemPos + 0] * d_Kernel[1]
    + data[sharMemPos + 1] * d_Kernel[0];
```
Optimizing Memory Usage

Minimizing data transfers with low bandwidth
- Minimizing host & device transfer
- Maximizing usage of shared memory
- Re-computing can sometimes be cheaper than transfer
- Low-parallelism computation can sometimes be faster

Organizing memory accesses based on the optimal memory access patterns
- Important for global memory access (low bandwidth)
- Shared memory accesses are usually worth optimizing only in case they have a high degree of bank conflicts

Global Memory Coalescing

Warp & global memory
- Threads execute by warp (32)
- Memory read/write by half warp (16)
- Global memory is considered to be partitioned into segments of size equal to 32, 64, or 128 bytes and aligned to this sizes.
- Block width must be divisible by 16 for coalescing
- Check your hardware (Compute Capability 1.x)
- Great improve throughput (Can yield speedups of >10)

Example 5: Sobel Filter
- Discrete convolution with Sobel mask
- Ideally each thread will compute one output pixel
Bad access pattern
- Global memory only. No texture memory or shared memory. Hundreds of clock cycles, compared to 1 or 2 for reading from shared memory
- Unstructured read (non sequential) mostly
- No cache, up to 12 global memory reads per thread

Reading Texture Memory
- Texture memory only. No shared memory
- Almost the same as collecting in CG (a little different)

\[
\begin{align*}
\text{float Horz} &= \text{Input}[ur] + \text{Input}[lr] + 2.0 \times \text{Input}[mr] - 2.0 \times \text{Input}[ml] - \text{Input}[ul] - \text{Input}[ll]; \\
\text{float Vert} &= \text{Input}[ur] + \text{Input}[ul] + 2.0 \times \text{Input}[um] - 2.0 \times \text{Input}[lm] - \text{Input}[ll] - \text{Input}[lr]; \\
\text{short Horz} &= \text{ul} + 2 \times \text{um} + \text{ur} - \text{ll} - 2 \times \text{lm} - \text{lr}; \\
\text{short Vert} &= \text{ul} + 2 \times \text{um} + \text{ur} - \text{ll} - 2 \times \text{lm} - \text{lr}; \\
\text{short Sum} &= (\text{short}) \left( \text{fScale} \times (\text{abs}(\text{Horz}) + \text{abs}(\text{Vert})) \right);
\end{align*}
\]
**Improve Caching?**

**Advantage**
- Texture memory read is better than global or constant memory

**Disadvantage**
- Only using hardware cache to handle spatial locality
- A pixel may be still loaded 9 times in total due to cache miss

**Take advantage of CUDA Shared Memory**
- Shared memory can be as fast as register! As a user-controlled cache.

1. Together with texture memory, load a block of the image into shared memory
2. Each thread compute a consecutive rows of pixels (sliding window)
3. Writing result to global memory

**Example 5: Sobel Filter**

Applying vertical and horizontal masks

Each thread will compute a number of consecutive rows of pixels

Computing all pixels inside one block (without apron)

**Reading Shared Memory**

- Shared memory + texture memory.

```c
__shared__ unsigned char shared[];

kernel<<<blocks, threads, sharedMem>>>(...);

......// copy a large tile of pixels into shared memory
__syncthreads();
......// read 9 pixels from shared memory
out.x = ComputeSobel(pix00, pix01, pix02, pix10, pix11, pix12, pix20, pix21, pix22, fScale );
......// read 9 pixels from shared memory
out.y = ComputeSobel(pix01, pix02, pix00, pix11, pix12, pix10, pix21, pix22, pix20, fScale );
......// read 9 pixels from shared memory
out.z = ComputeSobel(pix02, pix00, pix01, pix12, pix10, pix11, pix22, pix20, pix21, fScale );
......// read 9 pixels from shared memory
out.w = ComputeSobel(pix00, pix01, pix02, pix10, pix11, pix12, pix20, pix21, pix22, fScale );
__syncthreads();
```

**Shared Memory Bank Conflicts**

- Shared memory are divided into 16 banks to reduce the conflicts
- In a half-warp, each thread can access 32-bit from different banks simultaneously to achieve high memory bandwidth
- Conflict-free shared memory as fast as registers
- Linear
  ```c
  shared__float shared[32];
  float data = shared[BaseIndex + 1* tid];
  ``
- Random
  ```c
  1, 3, 5, 7 ...... (Any odd number)
  ```
**Shared Memory Bank Conflicts**

- **4-way bank conflicts**
  - Shared memory
  - **No bank conflicts**
    - ```
      __shared__ char shared[32];
      char data = shared[BasIndex + tid];
    ```
    - ```
      char data = shared[BasIndex + 4 * tid];
    ```
  - No bank conflicts
    - ```
      __shared__ char shared[32];
      char data = shared[BasIndex + tid];
    ```
    - ```
      char data = shared[BasIndex + 4 * tid];
    ```

- In shared memory edge detection example, 4 pixels are chosen as a group (4 unsigned char = 32 bit)
- If data is larger than 32 bits, one way to avoid bank conflicts in this case is to split data. It might not always improve and will perform worse in future architectures
- Structure assignment can be used

---

**Reading Shared Memory**

- **Shared memory 9 reads**
  - ```
    unsigned char pix00 = shared[BasIndex+4*tid+0*Pitch+0];
    unsigned char pix01 = shared[BasIndex+4*tid+0*Pitch+1];
    unsigned char pix02 = shared[BasIndex+4*tid+0*Pitch+2];
    unsigned char pix03 = shared[BasIndex+4*tid+0*Pitch+3];
    ```
  - ```
    unsigned char pix10 = shared[BasIndex+4*tid+1*Pitch+0];
    unsigned char pix11 = shared[BasIndex+4*tid+1*Pitch+1];
    unsigned char pix12 = shared[BasIndex+4*tid+1*Pitch+2];
    unsigned char pix13 = shared[BasIndex+4*tid+1*Pitch+3];
    ```
  - ```
    unsigned char pix20 = shared[BasIndex+4*tid+2*Pitch+0];
    unsigned char pix21 = shared[BasIndex+4*tid+2*Pitch+1];
    unsigned char pix22 = shared[BasIndex+4*tid+2*Pitch+2];
    unsigned char pix23 = shared[BasIndex+4*tid+2*Pitch+3];
    ```

- **Shared memory update 3 reads**
  - ```
    pix00 = shared[BasIndex+4*tid+0*Pitch+3];
    pix10 = shared[BasIndex+4*tid+1*Pitch+3];
    pix20 = shared[BasIndex+4*tid+2*Pitch+3];
    ```

---

**Shared Memory Broadcasting**

- Shared memory read a 32-bit word and broadcast to several threads simultaneously
  - Read
  - Reduce or resolve bank conflicts if set to broadcasting
  - Which word is selected as the broadcast word and which address is picked up for each bank at each cycle are unspecified

---

**Sample From Nvidia CUDA SDK**

- Image processing samples
  - Histogram
  - Bicubic filter
  - Sobel filter (FBO/PBO)
  - Boxfilter
  - Volume-render (3D PBO)
  - ...

- Code examples in this lecture reference above Nvidia SDK Sample
To Probe Further

NVIDIA CUDA Zone:
- Lots of information and code examples
- NVIDIA CUDA Programming Guide

GPGPU community:
- [http://www.gpgpu.org](http://www.gpgpu.org)
- User forums, tutorials, papers
- Good source: conference tutorials

Conclusion

- **CUDA Hardware**
  - Threads cooperate using shared memory
- **CUDA Programming API**
  - Launch parallel kernels
- **CUDA Graphics API**
  - Visualize the result
- **CUDA Performance**
  - Memory is complex but important

Course Schedule

1:30 – 2:00: Introduction (Klaus Mueller)
2:00 – 2:45: Graphics-style GPU programming with CG (Wei Xu)
2:45 – 3:00: GPGPU-style GPU programming with CUDA (Ziyi Zheng)
   - Coffee Break
3:30 – 4:00: GPGPU-style GPU programming with CUDA (Ziyi Zheng)
4:00 – 4:20: CT reconstruction pipeline components (Klaus Mueller)
4:20 – 5:20: GPU-accelerated CT reconstruction (Fang Xu)
5:20 – 5:30: Extensions and final remarks (all)