CS 380 - GPU and GPGPU Programming
Lecture 18: CUDA Memory Access 1

Markus Hadwiger, KAUST
Reading Assignment #10 (until Apr. 13)

Read (required):

• **Brook for GPUs: Stream Computing on Graphics Hardware**  
  Ian Buck et al.  

• **Matrix Multiplication example in CUDA book**  
  (Chapter 5, CUDA Memories)

Read (optional):

• **The Imagine Stream Processor**  
  Ujval Kapasi et al.  

• **Merrimac: Supercomputing with Streams**  
  Bill Dally et al.  
Let’s think about our problem in a new way

• Goal: SW programming model that matches today’s VLSI

Streams

• Collection of data records

• All data is expressed in streams

Kernels

• Inputs/outputs are streams

• Perform computation on streams

• Can be chained together

Courtesy John Owens
CUDA Highlights: Scatter

- **CUDA provides generic DRAM memory addressing**
  - Gather:

    - And **scatter**: no longer limited to write one pixel

    ![Diagram showing CUDA memory addressing](image)

  ![Diagram showing CUDA memory addressing](image)

  More programming flexibility
CUDA Highlights: On-Chip Shared Memory

- CUDA enables access to a parallel on-chip shared memory for efficient inter-thread data sharing

Big memory bandwidth savings
Programming Model: Memory Spaces

- **Global Memory**
  - Read-write per-grid
  - Hundreds of MBs
  - Very slow (600 clocks)

- **Texture Memory**
  - Read-only per-grid
  - Hundreds of MBs
  - Slow first access, but cached
  - Built-in filtering, clamping

- **Constant Memory**

- **Shared! Memory**
  - Read-write per-block
  - 16 KB per block
  - Very fast (4 clocks)

- **Registers**
  - Unique per thread
Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a block!
Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a block!

```c
// specify as global variable
__device__ __constant__ float gpuGamma[2];
...
// copy gamma value to constant device memory
cudaMemcpyToSymbol(gpuGamma, &gamma, sizeof(float));
// access as global variable in kernel
res = gpuGamma[0] * threadIdx.x;
```
Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32-bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access
- Not used explicitly for pixel shader programs
  - we dislike pixels talking to each other 😊

On Fermi/Kepler, there is a hardware-managed L1 cache in the same 64KB memory space as shared memory:

either 16KB shared + 48KB L1
or 48KB shared + 16KB L1
or 32KB shared + 32KB L1 (Kepler only)

On Maxwell, shared (cc. 5.0: 64KB; cc. 5.2: 96KB) and L1 are separate! (L1 and texture cache are same memory)
# Compute Capabilities 2.0 – 3.5

<table>
<thead>
<tr>
<th>Feature</th>
<th>FERMI GF100</th>
<th>FERMI GF104</th>
<th>KEPLER GK104</th>
<th>KEPLER GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>48</td>
<td>48</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / Multiprocessor</td>
<td>1536</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Registers / Multiprocessor</td>
<td>32768</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>255</td>
</tr>
<tr>
<td>Max Threads / Thread Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size Configurations (bytes)</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
</tr>
<tr>
<td></td>
<td>48K</td>
<td>48K</td>
<td>32K</td>
<td>32K</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>48K</td>
<td>48K</td>
</tr>
<tr>
<td>Max X Grid Dimension</td>
<td>$2^{16}-1$</td>
<td>$2^{16}-1$</td>
<td>$2^{32}-1$</td>
<td>$2^{32}-1$</td>
</tr>
<tr>
<td>Hyper-Q</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compute Capability of Fermi and Kepler GPUs
### Fermi
- GF100: 2.0
- GF104: 2.1

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65535</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td>16 KB</td>
<td>48 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td></td>
<td></td>
<td></td>
<td>16 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64 KB</td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td>Device dependent, between 6 KB and 8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to a CUDA array</td>
<td></td>
<td></td>
<td></td>
<td>8192</td>
<td>32768</td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to linear memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2^27</td>
</tr>
<tr>
<td>Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array</td>
<td></td>
<td></td>
<td></td>
<td>65536 x 32768</td>
<td>65536 x 65536</td>
</tr>
<tr>
<td>Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array</td>
<td></td>
<td></td>
<td></td>
<td>2048 x 2048 x 2048</td>
<td>4096 x 4096 x 4096</td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 million</td>
</tr>
</tbody>
</table>
## Compute Capabilities 2.x - 5.x

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>2.x</th>
<th>3.0, 3.2</th>
<th>3.5</th>
<th>3.7</th>
<th>5.0</th>
<th>5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>32 K</td>
<td>64 K</td>
<td>128 K</td>
<td>64 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>32 K</td>
<td></td>
<td>64 K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>63</td>
<td></td>
<td>255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>48 KB</td>
<td>112 KB</td>
<td>64 KB</td>
<td>96 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per thread block</td>
<td></td>
<td></td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td></td>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td></td>
<td></td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
<td></td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td>10 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td>12 KB</td>
<td></td>
<td></td>
<td>Between 12 KB and 48 KB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fermi
- GF100: 2.0
- GF104: 2.1

### Kepler
- GK104: 3.0
- GK110: 3.5
- GK210: 3.7

### Maxwell
- GM107: 5.0
- GM204: 5.2
Shared Memory Allocation

- 2 modes
- Static size within kernel
  ```c
  __shared__ float vec[256];
  ```

- Dynamic size when calling the kernel
  ```c
  // in main
  int VecSize = MAX_THREADS * sizeof(float4);
  vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, ...);

  // declare as extern within kernel
  extern __shared__ float vec[];
  ```
L1 Cache vs. Shared Memory

Two different configs (on Fermi and Kepler; NOT on Maxwell!)

• 64KB total
  – 16KB shared, 48KB L1 cache
  – 48KB shared, 16KB L1 cache
  – 32KB shared, 32KB L1 cache (Kepler only)

• Set per kernel

```c
// Device code
__global__ void MyKernel()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```
Vector-Matrix Multiplication - data parallelism -

\[ y = Mx \]
Vector-Matrix Multiplication V1

- Every thread computes a single output value in $y$
- Every thread computes the dot product between one line of $M$ and $x$
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of $M$ and $x$
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of $M$ and $x$
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of $M$ and $x$
- Computations totally independent
Setup

... // allocate memory

float* gpuMat, gpuVec, gpuResVec;
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuMat, w*h * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuVec, w * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuResVec, h * sizeof(float) ) );
CUT_CHECK_ERROR("allocation failed\n");

// upload M and x

CUDA_SAFE_CALL( cudaMemcpy( gpuMat, hostMat, w*h * sizeof(float),
    cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL( cudaMemcpy( gpuVec, hostVec, w * sizeof(float),
    cudaMemcpyHostToDevice) );

// compute the block and grid dimensions

dim3 threadBlock( MAX_THREADS, 1 );
dim3 blockGrid( h / MAX_THREADS + 1, 1, 1);
vecMat1<<< blockGrid, threadBlock >>>( gpuResVec, gpuMat, gpuVec,
    w, h);
CUT_CHECK_ERROR("vecMat filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );

// download result y

CUDA_SAFE_CALL( cudaMemcpy( hostResVec, gpuResVec, h * sizeof(float),
    cudaMemcpyDeviceToHost) );
cudaFree( gpuMat ); cudaFree( gpuVec ); cudaFree( gpuResVec );
__global__ void vecMat1(float *dst, const float* _mat,
const float* _v, int _w, int _h) {

// row index the thread is operating on
int i = blockIdx.x * blockDim.x + threadIdx.x;

if (i < _h) {
    float res = 0.;

    // dot product of one line
    for (int j = 0; j < _w; ++j) {
        res += _mat[i*_w + j] * _v[j];
    }

    // write result to global memory
    _dst[i] = res;
}
}
Why is this slow?

• Problem is bandwidth limited (read)
• Each thread is accessing
  – w elements of M
  – w elements of x

  from global memory

• Total bandwidth: $2 \times w \times h$

• But all threads are accessing the same elements of x
• Load x into shared memory and reuse!
Vector-Matrix Multiplication
- using shared memory -

\[ y = Mx \]
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x

shared memory

Parallel08 – Memory Access
Hendrik Lensch and Robert Strzodka
// allocate memory
float* gpuMat, gpuVec, gpuResVec;
CUDA_SAFE_CALL( cudaMemcpy( (void**)&gpuMat, w*h * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMemcpy( (void**)&gpuVec, w * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMemcpy( (void**)&gpuResVec, h * sizeof(float) ) );
CUT_CHECK_ERROR("allocation failed\n");

// upload M and x
CUDA_SAFE_CALL( cudaMemcpy( gpuMat, hostMat, w*h * sizeof(float), cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL( cudaMemcpy( gpuVec, hostVec, w * sizeof(float), cudaMemcpyHostToDevice) );

// compute the block and grid dimensions
dim3 threadBlock( MAX_THREADS, 1 );
dim3 blockGrid( h / MAX_THREADS + 1, 1, 1 );
vecMat2<<< blockGrid, threadBlock, w * sizeof(float) >>>( gpuResVec, gpuMat, gpuVec, w,h, w / MAX_THREADS);
CUT_CHECK_ERROR("vecMat filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );

// download result y
CUDA_SAFE_CALL( cudaMemcpy( hostResVec, gpuResVec, h * sizeof(float), cudaMemcpyDeviceToHost) );
cudaFree( gpuMat ); cudaFree( gpuVec ); cudaFree( gpuResVec );
VecMat Kernel – Version 2

__global__ void vecMat2(float *dst, const float* mat, const float* __v, int _w, int _h, int nIter) {
    extern __shared__ float vec[];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float res = 0.; int vOffs = 0;

    // load x into shared memory
    for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x) {
        vec[vOffs + threadIdx.x] = __v[vOffs + threadIdx.x];
    }
    // make sure all threads have written their parts
    __syncthreads();

    // now compute the dot product again
    // use elements of x loaded by other threads!
    if (i < _h) {
        for (int j = 0; j < _w; ++j) {
            res += mat[offs + j] * vec[j];
        }
        _dst[i] = res;
    }
}
__global__ void vecMat2(float *dst, const float* mat, const float* v, int w, int h, int nIter) {
    extern __shared__ float vec[];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float res = 0.; int vOffs = 0;

    // load x into shared memory
    for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x) {
        vec[vOffs + threadIdx.x] = _v[vOffs + threadIdx.x];
    }

    // make sure all threads have written their parts
    __syncthreads();

    // now compute the dot product again
    // use elements of x loaded by other threads!
    if (i < h) {
        for (int j = 0; j < w; ++j) {
            res += mat[offs + j] * vec[j];
        }
        _dst[i] = res;
    }
}
Thank you.

- Hendrik Lensch, Robert Strzodka
- NVIDIA