CUDA Memory Architecture

GPGPU class
Week 4
CPU – GPU HW Differences

- **CPU**
  - Most die area used for memory cache
  - Relatively few transistors for ALUs

- **GPU**
  - Most die area used for ALUs
  - Relatively small caches
CPU – GPU HW Differences

- Situation is slowly changing
  - Many-core CPUs
  - More caches on GPU die
CPU – GPU Differences

• What does that mean for SW?

• CPU
  • Hides memory latency via hierarchy of caches
    – L1, L2 and L3 caches
  • Little need for thread programming
    – This is currently changing

• GPU
  • Memory latency not hidden by large cache
    – Only texture cache (roughly specialized L1 cache)
    – Needs many (active) threads to hide latency!
  • Only many-threads applications are useful
    – Extra bonus of CUDA: threads can easily communicate (with limits)
A View on the G80 Architecture

● “Graphics mode:”
A View on the G80 Architecture

“CUDA mode:”
CUDA Memory Types

Each thread can:

- Read/write per-thread **registers**
- Read/write per-thread **local memory**
- Read/write per-block **shared memory**
- Read/write per-grid **global memory**
- Read/only per-grid **constant memory**
CUDA Memory Types & Uses

- **Compute Capability 1.x**
  - Global memory (read and write)
    - Slow & uncached
    - Requires sequential & aligned 16 byte reads and writes to be fast (coalesced read/write)
  - Texture memory (read only)
    - Cache optimized for 2D spatial access pattern
  - Constant memory
    - This is where constants and kernel arguments are stored
    - Slow, but with cache (8 kb)
  - Shared memory (16 kb per MP)
    - Fast, but take care of bank conflicts
    - Exchange data between threads in a block
  - Local memory (used for whatever does not fit into registers)
    - Slow & uncached, but automatic coalesced reads and writes
  - Registers (8192-16384 32-bit registers per MP)
    - Fastest, scope is thread local
CUDA Memory Types & Uses

- **Compute Capability 2.x**
  - Global memory (read and write)
    - Slow, but now with cache
  - Texture memory (read only)
    - Cache optimized for 2D spatial access pattern
  - Constant memory
    - Slow, but with cache (8 kb)
    - Special “Load Uniform” (LDU) instruction
  - Shared memory (48kb per MP)
    - Fast, but slightly different rules for bank conflicts now
  - Local memory
    - Slow, but now with cache
  - Registers (32768 32-bit registers per MP)
CUDA Memory Limitations

- **Global memory**
  - Best if 64 or 128 bytes (16 or 32 words) are read
    - Parallel read/writes from threads in a block
    - Sequential memory locations
    - With appropriate alignment
    - Called “coalesced” read/write
  - Otherwise: a sequence of reads/writes
    - >10x slower!

- **Shared memory**
  - Fastest if
    - All threads read from the same shared memory location
    - All threads index a shared array via permutation
      - E.g. linear reads/writes
  - Otherwise: bank conflicts
    - Not as bad as uncoalesced global memory reads/writes
CUDA Type Qualifiers

- Type Qualifier table

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int LocalArray[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>[<strong>device</strong>] <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td>[<strong>device</strong>] <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- Notes:
  - __device__ not required for __local__, __shared__, or __constant__
  - Automatic variables without any qualifier reside in a register
    - Except arrays that reside in local memory
    - Or not enough registers available for automatic variables
CUDA Type Qualifiers

- Type Qualifier table / performance

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Performance penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>1x</td>
</tr>
<tr>
<td>int LocalArray[10];</td>
<td>local</td>
<td>100x</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>1x</td>
</tr>
<tr>
<td><strong>device</strong>           int GlobalVar;</td>
<td>global</td>
<td>100x</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>1x</td>
</tr>
</tbody>
</table>

- Notes (for G80, somewhat simplified)
  - Scalar vars reside in on-chip registers (fast)
  - Shared vars resides in on-chip memory (fast)
  - Local arrays and global variables reside in off-chip memory (slow)
  - Constants reside in cached off-chip memory
CUDA Type Qualifiers

- Type Qualifier table / performance

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Instances</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>100.000s</td>
<td>1</td>
</tr>
<tr>
<td>int LocalArray[10];</td>
<td>100.000s</td>
<td>1</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>100s</td>
<td>100s</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>1</td>
<td>100.000s</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>1</td>
<td>100.000s</td>
</tr>
</tbody>
</table>

- 100.000s per-thread variables, but only accessed per thread
- 100s of shared variables, accessed by ~100 threads (a block)
- Global memory and constants are accessed by many threads
CUDA Type Qualifiers

- Where is a variable accessed?

  - Can host access it? (e.g. via cudaMemcpy)
    - yes
    - no

  - __global__
  - __constant__
  - register (automatic)
    - __shared__
    - __local__

Declared outside of any Function

Declared in the kernel
Pointers & CUDA

- Pointers can only point to global memory
  - Typical usage: as array argument to kernels
    - `__global__ void kernel(float * d_ptr);`
  - Alternative: explicit pointer assignment
    - `float * ptr = &globalVar;`
  - Use pointers only to access global memory
    - Simple, regular read/write patterns
    - No pointer chains (linked lists)
    - No C wizard pointer magic
      - But index magic is fine
A Common Programming Scenario 1

- **Task:**
  - Load data from global memory
  - Do **thread-local** computations
  - Store result to global memory

- **Solution (statements in kernel)**
  - Load data to registers (coalesced)
    - float a = d_ptr[blockIdx.x*blockDim.x + threadIdx.x];
  - Do computation with registers
    - float res = f(a);
  - Store back result (coalesced)
    - d_ptr[blockIdx.x*blockDim.x + threadIdx.x] = res;
A Common Programming Scenario 1

- Full kernel code

```c
__global__ void kernel(float * d_ptr)
{
    // Coalesced read if blockDim.x is a multiple of 16
    float a = d_ptr[blockIdx.x*blockDim.x + threadIdx.x];

    float res = a*a;

    // Coalesced write if blockDim.x is a multiple of 16
    d_ptr[blockIdx.x*blockDim.x + threadIdx.x] = res;
}
```
A Common Programming Scenario 2

- **Task:**
  - Load data from global memory
  - Do **block-local** computations
  - Store result to global memory
- **Solution (statements in kernel)**
  - Load data to shared memory (coalesced)
    - __shared__ float a_sh[BLOCK_SIZE]; // blockDim.x == BLOCK_SIZE
    - a_sh[threadIdx.x] = d_ptr[blockIdx.x*blockDim.x + threadIdx.x];
    - __syncthreads(); // !!!
  - Do computation
    - float res = f(a_sh[threadIdx.x], a_sh[threadIdx.x+1]);
  - Store back result (coalesced)
    - d_ptr[blockIdx.x*blockDim.x + threadIdx.x] = res;
A Common Programming Scenario 2

• Full kernel code

```c
__global__ void kernel(float * d_ptr)
{
    // Note: BLOCK_SIZE == blockDim.x
    int tx = threadIdx.x, bx = blockIdx.x;

    __shared__ float a_sh[BLOCK_SIZE];
    a_sh[tx] = d_ptr[bx*blockDim.x + tx];
    __syncthreads();

    // Ignore out-of-bounds access for now
    float res = a_sh[tx+1] - a_sh[tx];
    d_ptr[bx*blockDim.x + tx] = res;
}
```
General CUDA Scenario

- Partition data into subsets fitting into shared memory
- Copy constants to `__constant__` variables
  - But not the input of the problem!
  - Limited size of constant memory and its cache
- One thread block per subset
  - Load data from global memory to `__shared__` memory
    - Exploit coalescing
  - Perform computation on the subset
    - Exploit communication between threads in a block
      - Not always possible
      - Use `__shared__` variables, pay attention to race conditions!
  - Write result (in register or `__shared__` variable) to global memory
    - Exploit coalescing
Communication via Shared Mem.

• Little question:

    __global__ race_condition()
    {
        __shared__ int shared_var = threadIdx.x;
        // What is the value of shared_var here???
    }
Communication via Shared Mem.

• Answer:
  • Value of shared_var is undefined
  • This is a race condition
    – Multiple threads writing to one variable w/o explicit synchronization
    – Variable will have arbitrary (i.e. undefined) value
  • Need for synchronization/barriers
    – __syncthreads()
    – Atomic operations
Communication via Shared Mem.

- **__syncthreads()**
  - Point of synchronization for all threads in a block
  - Not always necessary
    - Half-warps are lock-stepped
- Common usage: make sure data is ready

```c
__global__ void kernel(float * d_src)
{
    __shared__ float a_sh[BLOCK_SIZE];
    a_sh[threadIdx.x] = d_src[threadIdx.x];
    __syncthreads();
    // a_sh is now correctly filled by all
    // threads in the block
}```
Communication via Shared Mem.

• Atomic operations
  • atomicAdd(), atomicSub(), atomicExch(), atomicMax(), …

• Example

```c
__global__ void sum(float * src, float * dst) {
  atomicAdd(dst, src[threadIdx.x]);
}
```
Communication via Shared Mem.

• But: atomic operations are not cheap
• Serialized write access to a memory cell
• Better solution:
  • Partial sums within thread block
    – atomicAdd() on a __shared__ variable
  • Global sum
    – atomicAdd() on global memory
Communication via Shared Mem.

Better version of sum()

```c
__global__ void sum(float * src, float * dst)
{
    int pos = blockDim.x*blockIdx.x + threadIdx.x;

    __shared__ float partial_sum;
    if (threadIdx.x == 0) partial_sum = 0.0f;
    __syncthreads();

    atomicAdd(&partial_sum, src[pos]);

    if (threadIdx.x == 0) atomicAdd(dst, partial_sum)
}
```
Communication via Shared Mem.

• General guidelines:
  • Do not synchronize or serialize if not necessary
  • Use __syncthreads() to wait until __shared__ data is filled
  • Data access pattern is regular or predictable
    → __syncthreads()
  • Data access pattern is sparse or not predictable
    → atomic operations
  • Atomic operations are much faster for shared variables than for global ones
Acknowledgements

- UIUC parallel computing course
  - http://courses.engr.illinois.edu/ece498/al/Syllabus.html
- Stanford GPU lecture
- General CUDA training resources