L3: Memory Hierarchy Optimization I, Locality and Data Placement

Overview of Lecture

- Where data can be stored
  - And how to get it there
- Some guidelines for where to store data
  - Who needs to access it?
  - Read only vs. Read/Write
  - Footprint of data
- High level description of how to write code to optimize for memory hierarchy
  - More details Wednesday and next week
- Reading:
  - Chapter 5, Kirk and Hwu book
  - Or, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

Targets of Memory Hierarchy Optimizations

- Reduce memory latency
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize memory bandwidth
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Administrative

- Assignment due Friday, Jan. 18, 5 PM
  - Use handin program on CADE machines
    - "handin CS6235 lab1 <probfile>"
- Mailing list
  - CS6235@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!
Optimizing the Memory Hierarchy on GPUs, Overview

- Device memory access times non-uniform so **data placement** significantly affects performance.
  - But controlling data placement may require additional copying, so consider overhead.
- Optimizations to increase memory bandwidth. Idea: maximize utility of each memory access.
  - **Coalesce** global memory accesses
  - Avoid memory bank conflicts to increase memory access parallelism
  - **Align** data structures to address boundaries

---

**Memory Hierarchy Terminology**

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>Yes</td>
<td>Read/write</td>
<td>One thread</td>
<td>O(1 cycle)</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>No</td>
<td>Read/write</td>
<td>Threads in block</td>
<td>O(1 cycle) w/ bank conflict</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No/Yes</td>
<td>Read/write</td>
<td>All threads + host</td>
<td>O(1-100) cycles, depending on if cached</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read only</td>
<td>One thread</td>
<td>O(1-1000) cycles, depending on if cached</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read only</td>
<td>All threads + host (host may write)</td>
<td>O(1-100) cycles, depending on if cached</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read only</td>
<td>All threads + host (host may write)</td>
<td>O(1-100) cycles, depending on if cached</td>
</tr>
<tr>
<td>Surface</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read/write</td>
<td>All threads+host</td>
<td>O(1-100) cycles, depending on if cached</td>
</tr>
</tbody>
</table>

---

**Reuse and Locality**

- Consider how data is accessed
  - **Data reuse:**
    - Same data used multiple times
    - Intrinsic in computation
  - **Data locality:**
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

---

**Hardware Implementation: Memory Architecture**

- The local, global, constant, and texture spaces are regions of device memory (DRAM)
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
  - A read-only constant cache
  - A read-only texture cache
  - Data cache (Fermi only)
Data Placement: Conceptual

- Copies from host to device go to some part of global memory (possibly, constant or texture memory)
- How to use SP shared memory
  - Must construct or be copied from global memory by kernel program
- How to use constant or texture cache
  - Read-only "reused" data can be placed in constant & texture memory by host
- Also, how to use registers
  - Most locally-allocated data is placed directly in registers
  - Even array variables can use registers if compiler understands access patterns
  - Can allocate "superwords" to registers, e.g., float4
  - Excessive use of registers will "spill" data to local memory
- Local memory
  - Deals with capacity limitations of registers
  - Eliminates worries about race conditions
  - ... but SLOW

Rest of Today’s Lecture

- Mechanics of how to place data in shared memory and constant memory
- Tiling transformation to reuse data within
  - Shared memory
  - Data cache (Fermi only)
  - Constant memory

Data Placement: Syntax

- Through type qualifiers
  - __constant__, __shared__, __local__, __device__
- Through cudaMemcpy calls
  - Flavor of call and symbolic constant designate where to copy
- Implicit default behavior
  - Device memory without qualifier is global memory
  - Host by default copies to global memory
  - Thread-local variables go into registers unless capacity exceeded, then local memory

Now Let’s Look at Shared Memory

- Common Programming Pattern (5.3 of CUDA 4 manual)
  - Load data into shared memory
  - Synchronize (if necessary)
  - Operate on data in shared memory
  - Synchronize (if necessary)
  - Write intermediate results to global memory
  - Repeat until done
Mechanics of Using Shared Memory

- **__shared__** type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

```c
extern __shared__
  float
d_s_array[];
```

/* Form of dynamic allocation */
```c
d_s_array[] = ...; // synchronize threads before use
```
Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order

\[
\begin{align*}
&\text{for } (i=0; \ i<3; \ i++) \\
&\text{for } (j=0; \ j<6; \ j++) \\
\end{align*}
\]

new traversal order!

Which one is better for row-major storage?

Safety of Permutation

\- Intuition: Cannot permute two loops \(i\) and \(j\) in a loop nest if doing so changes the relative order of a read and write or two writes to the same memory location

\[
\begin{align*}
&\text{for } (i=0; \ i<3; \ i++) \\
&\text{for } (j=0; \ j<6; \ j++) \\
\end{align*}
\]

Can permute?

Tiling (Blocking): Loop Reordering Transformation

\- Tiling reorders loop iterations to bring iterations that reuse data closer in time

\[
\begin{align*}
&\text{for } (j=0; \ j<M; \ j++) \\
&\text{for } (i=0; \ i<N; \ i++) \\
&D[i] = D[i] + B[j][i];
\end{align*}
\]

Tiling Example

\[
\begin{align*}
&\text{for } (j=0; \ j<M; \ j++) \\
&\text{for } (i=0; \ i<N; \ i++) \\
&D[i] = D[i] + B[j][i];
\end{align*}
\]

Strip mine

\[
\begin{align*}
&\text{for } (i=i; \ ii<N; \ ii++) \\
&D[i] = D[i] + B[j][i];
\end{align*}
\]

Permute

\[
\begin{align*}
&\text{for } (i=0; \ i<i<N; \ i++) \\
&D[i] = D[i] + B[j][i];
\end{align*}
\]
Legality of Tiling

- Tiling is safe only if it does not change the order in which memory locations are read/written
  - We'll talk about correctness after memory hierarchies
- Tiling can conceptually be used to perform the decomposition into threads and blocks (computation partitioning)
- Tiling is also used to reduce the footprint of data to fit in limited capacity storage (more later)

CUDA Version of Example (Tiling for Computation Partitioning)

```c
for (ii=0; ii<N; ii+=s)
  for (i=ii; i<min(ii+s-1,N); i++)
    for (j=0; j<N; j++)
      D[i] = D[i] + B[j][i];

... <<<ComputeI(N/s,s)>>>(d_D, d_B, N);
...

__global__ ComputeI (float *d_D, float *d_B, int N) {
    int ii = blockIdx.x;
    int i = ii*s + threadIdx.x;
    for (j=0; j<N; j++)
      d_D[i] = d_D[i] + d_B[j*N+i];
}
```

Tiling for Limited Capacity Storage

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in constant cache exceeds cache capacity or data in registers exceeds register capacity or (as in example) data in shared memory for block still exceeds shared memory capacity

Memory Hierarchy Example: Matrix vector multiply

```c
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    a[i] += c[i][j] * b[j];
  }
}
```

Remember to:
* Consider correctness of parallelization strategy (next week)
* Exploit locality in shared memory and registers
Let's Take a Closer Look

- Implicitly use tiling to decompose parallel computation into independent work.
- Additional tiling is used to map portions of "b" to shared memory since it is shared across threads.
- "a" has reuse within a thread so use a register.

```c
__global__ mv_GPU(float* a, float* b, float** c) {
  int bx = blockIdx.x; int tx = threadIdx.x;
  __shared__ float bcpy[32];
  double acpy = a[tx + 32 * bx];
  for (k = 0; k < 32; k++) {
    bcpy[tx] = b[32 * k + tx];
    __syncthreads();
  //this loop is actually fully unrolled
  for (j = 32 * k; j <= 32 * k + 32; j++) {
    acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
  }
  __syncthreads();
  a[tx + 32 * bx] = acpy;
}
```

Summary of Lecture

- How to place data in constant memory and shared memory.
- Introduction to permutation and tiling transformation.
- Matrix-vector multiply example.