L6: Memory Hierarchy Optimization I, Data Placement

Overview
- Where data can be stored
  - And how to get it there
- Some guidelines for where to store data
- High level description of how to write code to optimize for memory hierarchy
  - Fill in details Wednesday

Administrative
- Projects mostly graded
  - Grades by Wednesday, possibly sooner
- Homework coming out this afternoon
  - Due Wednesday, Feb. 18

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
Optimizing the Memory Hierarchy on GPUs

- 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.
  - **Align** data structures to address boundaries
  - **Coalesce** global memory accesses
  - **Avoid memory bank conflicts** to increase memory access parallelism

Reuse and Locality

- Consider how data is accessed
  - **Data reuse**: Some 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

Recover and Locality

- Consider how data is accessed
  - **Data reuse**: Some 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

Programmer’s View: Memory Spaces

- 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
  - Read only per-grid texture memory
- The host can read/write global, constant, and texture memory

Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory.
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space
Terminology Review

- **device** = GPU = set of multiprocessors
- **Multiprocessor** = set of processors & shared memory
- **Kernel** = GPU program
- **Grid** = array of thread blocks that execute a kernel
- **Thread block** = group of SIMD threads that execute a kernel and can communicate via shared memory

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A - resident</td>
<td>Read/write</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
</tr>
</tbody>
</table>

Access Times (REWRITE?)

- **Register** - dedicated HW - single cycle
- **Constant and Texture caches** - possibly single cycle, proportional to addresses accessed by warp
- **Shared Memory** - dedicated HW - single cycle
- **Local Memory** = DRAM, no cache - *slow*
- **Global Memory** = DRAM, no cache - *slow*
- **Constant Memory** = DRAM, cached, 1...10s...100s of cycles, depending on cache locality
- **Texture Memory** = DRAM, cached, 1...10s...100s of cycles, depending on cache locality
- **Instruction Memory (invisible)** = DRAM, cached

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
- Local memory
  - Deals with capacity limitations of registers and shared memory
  - Eliminates worries about race conditions
  - ... but SLOW

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 variables go into registers unless capacity exceeded, then local memory
Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Type Qualifier</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__.
- Automatic variables without any qualifier reside in a register.  
  - Except arrays that reside in local memory.

Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    ```
    __global__ void KernelFunc(float* ptr)
    ```
  - Obtained as the address of a global variable:
    ```
    float* ptr = &GlobalVar;
    ```

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
  - Constant cache

Constant Memory Example

- Signal recognition:
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Compute $M_1 V$, $M_2 V$, ..., $M_n V$

```c
__constant__ float d_signalVector[M];
__device__ float R[N][M];
__host__ void outerApplySignal ()
{
    float *h_inputSignal;
    dim3 dimGrid(N);
    dim3 dimBlock(M);
    cudaMemcpyToSymbol(d_signalVector, h_inputSignal, M * sizeof(float));
    ApplySignal<<<dimGrid, dimBlock>>>(M);
}

__global__ void ApplySignal(int M) {
    float result = 0.0; /* register */
    for (i=0; i<M; i++)
        result += d_M[blockIdx.x][threadIdx.x][i] * d_signalVector[i];
    R[blockIdx.x][threadIdx.x] = result;
}
```
More on Constant Cache

- Example from previous slide
  - All threads in a block accessing same
    element of signal vector
  - Brought into cache for first access, then
    latency equivalent to a register access

Additional Detail

- Suppose each thread accesses different
  data from constant memory on same
  instruction
  - Reuse across threads?
    - Consider capacity of constant cache and locality
    - Code transformation needed? (later in lecture)
    - Cache latency proportional to number of
      accesses in a warp
  - No reuse?
    - Should not be in constant memory.

Now Let’s Look at Shared Memory

- Common Programming Pattern (5.1.2
  of CUDA 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[];
  __global__ void compute()
  {
      d_s_array[i] = …;
  }
  __global__ void compute2()
  {
      __shared__ float d_s_array[M];
      d_s_array[j] = …;
      d_g_array[j] = d_s_array[j];
  }
  ```
Matrix Transpose (from SDK)

```c
__global__ void transpose(float *odata, float *idata, int width, int height)
{
  __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
  // read the matrix tile into shared memory
  unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
  unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
  unsigned int index_in = yIndex * width + xIndex;
  block[threadIdx.y][threadIdx.x] = idata[index_in];
  __syncthreads();
  // write the transposed matrix tile to global memory
  xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
  unsigned int index_out = yIndex * height + xIndex;
  odata[index_out] = block[threadIdx.x][threadIdx.y];
}
```

Recall Reuse and Locality

- Consider how data is accessed
  - **Data reuse:**
    - Some 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

Temporal Reuse in Sequential Code

- Same data used in distinct iterations $I$ and $I'$
  ```c
  for (i=1; i<N; i++)
    for (j=1; j<N; j++)
  ```
- $A[j]$ has self-temporal reuse in loop $i$

Spatial Reuse (Ignore for now)

- Same data transfer (usually cache line) used in distinct iterations $I$ and $I'$
  ```c
  for (i=1; i<N; i++)
    for (j=1; j<N; j++)
  ```
- $A[j]$ has self-spatial reuse in loop $j$
- Multi-dimensional array note: C arrays are stored in row-major order
Group Reuse

- Same data used by distinct references

```c
for (i=1; i<N; i++)
for (j=1; j<N; j++)
```

- \( A[j], A[j+1] \) and \( A[j-1] \) have group reuse (spatial and temporal) in loop \( j \)

Can Use Reordering Transformations!

- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - Safety? (doesn’t reverse dependences)
  - Profitability? (improves locality)

Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order:

```c
for (i=0; i<3; i++)
for (j=0; j<6; j++)
```

```c
for (i=0; i<3; i++)
for (j=0; j<6; j++)
```

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 reverses the direction of any dependence.
- Loops \( i \) through \( j \) of an \( n \)-deep loop nest are fully permutable if for all dependences \( D \), either
  
  \( (d_k, ..., d_l) > 0 \)

  or

  \( \forall k, l, i \leq k \leq j, d_k \geq 0 \)

- Stated without proof: Within the affine domain, \( n-1 \) inner loops of \( n \)-deep loop nest can be transformed to be fully permutable.
Simple Examples: 2-d Loop Nests

for (i = 0; i < 3; i++)
for (j = 0; j < 6; j++)

• Distance vectors

• Ok to permute?

Tiling (Blocking): Another Loop Reordering Transformation

• Blocking reorders loop iterations to bring iterations that reuse data closer in time

Tiling Example

for (j = 1; j < M; j++)
for (i = 1; i < N; i++)
D[i] = D[i] + B[j][i];

Strip mine
for (j = 1; j < M; j++)
for (i = 1; i < N; i += s)
for (ii = i, ii < min(ii+s-1, N), ii++)
D[ii] = D[ii] + B[j][ii];

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

Legality of Tiling

• Tiling = strip-mine and permutation
  – Strip-mine does not reorder iterations
  – Permutation must be legal
  OR
  – strip size less than dependence distance
A Few Words On Tiling

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations:
  - Between grids if data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity
  - Within threads if data in constant cache exceeds cache capacity

Matrix Multiplication
A Simple Host Version in C

```c
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j)
      double sum = 0;
      for (int k = 0; k < Width; ++k)
        double a = M[i*Width + k];
        double b = N[k*Width + j];
        sum += a * b;
      P[i*Width + j] = sum;
}
```

Tiled Multiply Using Thread Blocks

- One block computes one square sub-matrix $P_{sb}$ of size BLOCK_SIZE
- One thread computes one element of $P_{sb}$
- Assume that the dimensions of $M$ and $N$ are multiples of BLOCK_SIZE and square shape

Shared Memory Usage

- Assume each SMP has 16KB shared memory
  - Each Thread Block uses $2\times 256\times 4B = 2KB$ of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
  - For BLOCK_SIZE = 16, this allows up to $8\times 512 = 4,096$ pending loads
    - In practice, there will probably be up to half of this due to scheduling to make use of SPs.
  - The next BLOCK_SIZE 32 would lead to $2\times 32\times 32\times 4B = 8KB$ shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time
First-order Size Considerations

- Each Thread Block should have a minimal of 192 threads
  - BLOCK_SIZE of 16 gives 16*16 = 256 threads
- A minimal of 32 Thread Blocks
  - A 1024*1024 P Matrix gives 64*64 = 4096 Thread Blocks
- Each thread block performs 2*256 = 512 float loads from global memory for 256 * 2*16 = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor

CUDA Code – Kernel Execution Configuration

```c
// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.width / dimBlock.x,
              M.height / dimBlock.y);
```

For very large N and M dimensions, one will need to add another level of blocking and execute the second-level blocks sequentially.

CUDA Code – Kernel Overview

```c
// Block index
int bx = blockIdx.x;
int by = blockIdx.y;
// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;
// Pvalue stores the element of the block sub-matrix
// that is computed by the thread
float Pvalue = 0;
// Loop over all the sub-matrices of M and N
// required to compute the block sub-matrix
for (int m = 0; m < M.width/BLOCK_SIZE; ++m) {
  // code from the next few slides ;
}
```

CUDA Code - Load Data to Shared Memory

```c
// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);
// Get a pointer to the current sub-matrix Nsub of N
Matrix Nsub = GetSubMatrix(N, bx, m);
__shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];
// each thread loads one element of the sub-matrix
Ms[ty][tx] = GetMatrixElement(Msub, tx, ty);
// each thread loads one element of the sub-matrix
Ns[ty][tx] = GetMatrixElement(Nsub, tx, ty);
```
CUDA Code - Compute Result

// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();

// each thread computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
  Pvalue += Ms[ty][k] * Ns[k][tx];

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of M and N in the next iteration
__syncthreads();

CUDA Code - Save Result

// Get a pointer to the block sub-matrix of P
Matrix Psub = GetSubMatrix(P, bx, by);

// Write the block sub-matrix to device memory;
// each thread writes one element
SetMatrixElement(Psub, tx, ty, Pvalue);

This code should run at about 45 GFLOPS

Matrix Multiply in CUDA

• Imagine you want to compute extremely large matrices.
  – That don’t fit in global memory
• This is where an additional level of tiling could be used, between grids

Summary of Lecture

• How to place data in constant memory and shared memory
• Reordering transformations to improve locality
• Tiling transformation
• Matrix multiply example
Next Time

- Complete this example
- Reasoning about reuse and locality
- Talk about projects and assign proposal