L7: Memory Hierarchy Optimization, cont.

Administrative

• Homework #2, posted on website
  – Due 5PM, Thursday, February 19
  – Use handin program to submit
• Project proposals
  – Due 5PM, Friday, March 13 (hard deadline)
  – Discuss today

Outline

• Homework discussion
• Project discussion
• Complete tiling discussion and matrix multiply example
• Calculating data reuse and data footprint

Project Proposal

• Project Logistics:
  – 2-3 person teams
  – Significant implementation, worth 55% of grade
  – Parts: proposal, design review (3/30 and 4/1), final presentation and report (end of semester)
  – Each person turns in the proposal (should be same as other team members)
• Proposal:
  – 3-4 page document (11pt, single-spaced)
  – Submit with handin program:
    “handin cs6963 prop <pdf-file>“
Content of Proposal

I. Team members: Name and a sentence on expertise for each member

II. Problem description
   - What is the computation and why is it important?
   - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page

III. Suitability for GPU acceleration
   - Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation. Use measurements from CPU execution of computation if possible.
   - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
   - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.

IV. Intellectual Challenges
   - Generally, what makes this computation worthy of a project?
   - Point to any difficulties you anticipate at present in achieving high speedup.

Capacity Questions

- How much shared memory, global memory, registers, constant memory, constant cache, etc.?
  - deviceQuery function (in SDK) instantiates variable of type cudaDeviceProp with this information and prints it out.
- Summary for 9400 M (last homework problem)
  - 8192 registers per SM
  - 16KB shared memory per SM
  - 64KB constant memory
    - stored in global memory
    - presumably, 8KB constant cache
  - 256MB global memory

Main points from Previous Lecture

- Considered latencies of different levels of memory hierarchy
  - Global memory latency roughly hundreds of cycles
  - Registers, shared memory and constant cache roughly single cycle latency
  - Constant memory (stored in global memory) can be used for read-only data, but only a win if it is cached
- Examples showing how to place data in constant or shared memory
- Tiling transformation for managing limited capacity storage (shared memory, constant cache, global memory, even registers)

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:
    - 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

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

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

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

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

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
  \[
  (d_{i-1}, \ldots, d_{i-1}) > 0
  \]
  or
  \[
  \text{forall } k, i \leq k \leq j, d_k \geq 0
  \]
- **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

- Distance vectors
- Ok to permute?

Tiling (Blocking): Another Loop Reordering Transformation

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

```cpp
for (i=0; i<3; i++)
for (j=0; j<6; j++)
```
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++)
    for (ii=1; ii<N; ii+=s)
      for (i=ii, i<min(ii+s-1,N), i++)
        D[i] = D[i] + B[j][i];

Permute

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

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
  – Special form (unroll-and-jam) used for registers

Locality Optimization

• Reuse analysis can be formulated in a manner similar to dependence analysis
  – Particularly true for temporal reuse
  – Spatial reuse requires special handling of most quickly varying dimension (still ignoring)
• Simplification for today’s lecture
  – Estimate data footprint for innermost loop for different scenarios
  – Select scenario that minimizes footprint
Reuse Analysis: 
Use to Estimate Data Footprint

\[ \text{for } (i=0; i<N; i++) \] 

\[ \text{for } (j=0; j<M; j++) \] 

---

Allen & Kennedy: 
Innermost memory cost

- Innermost memory cost: \( C_M(L_i) \)
  - Assume \( L_i \) is innermost loop
  - \( i \) = loop variable, \( N \) = number of iterations of \( L_i \)
  - For each array reference \( r \) in loop nest:
    - \( r \) does not depend on \( i \): cost (\( r \)) = 1
    - \( r \) such that \( i \) strides over a dimension: cost (\( r \)) = \( N \)
    - (Can be more precise if taking transfer size into account, ignored today)
  - \( C_M(L_i) \) = sum of cost (\( r \))

Implicit in this cost function is that \( N \) is unknown and sufficiently large that "storage" capacity is exceeded by data footprint in innermost loop.

---

Canonical Example: matrix multiply
Selecting Loop Order for Cache-based Architecture

\[ \text{DO } I = 1, N \]
\[ \text{DO } J = 1, N \]
\[ \text{DO } K = 1, N \]
\[ C(I,J) = C(I,J) + A(I,K) * B(K,J) \]

- \( C_M(I) = 2N^3/\text{cls} + N^2 \)
- \( C_M(J) = 2N^3 + N^2 \)
- \( C_M(K) = N^3 + N^3/\text{cls} + N^2 \)
- Ordering by innermost loop cost: \( (J, K, I) \)

---

Canonical Example: Matrix Multiply
Selecting Tile Size

Choose \( T_i \) and \( T_k \) such that data footprint does not exceed cache capacity

\[ \text{DO } K = 1, N \text{ by } T_k \]
\[ \text{DO } I = 1, N \text{ by } T_i \]
\[ \text{DO } J = 1, N \]
\[ \text{DO } II = 1, \min(II+T_i, N) \]
\[ \text{DO } KK = K, \min(KK+T_k, N) \]
\[ C(II,J) = C(II,J) + A(II,KK) * B(KK,J) \]
"Tiling" for Registers

- A similar technique can be used to map data to registers
- Unroll-and-jam
  - Unroll outer loops in a nest and fuse together resulting inner loops
- Scalar replacement
  - May be followed by replacing array references with scalar variables to help compiler identify register opportunities

Unroll II, TI = 4 (Equiv. to unroll-and-jam)

```
DO K = 1, N by T
DO I = 1, N by 4
DO J = 1, N
DO KK = K, min(KK + T - N)
  C(I,J) = C(I,J) + A(I,KK) * B(KK,J)
  C(I+1,J) = C(I+1,J) + A(I+1,KK) * B(KK,J)
  C(I+2,J) = C(I+2,J) + A(I+2,KK) * B(KK,J)
  C(I+3,J) = C(I+3,J) + A(I+3,KK) * B(KK,J)
```

In other architectures with deep instruction pipelines, this optimization can also be used to expose instruction-level parallelism.

Scalar Replacement

```
DO K = 1, N by T
DO I = 1, N by 4
DO J = 1, N
DO KK = K, min(KK + T - N)
  C1 = C(I,J) + A(I,KK) * B(KK,J)
  C2 = C(I+1,J) + A(I+1,KK) * B(KK,J)
  C3 = C(I+2,J) + A(I+2,KK) * B(KK,J)
  C4 = C(I+3,J) + A(I+3,KK) * B(KK,J)
```

Now C accesses are to named registers.
Compiler guaranteed to map to registers.

Matrix Multiplication
A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double precision
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 $P_{sub}$ of size BLOCK_SIZE.
- One thread computes one element of $P_{sub}$.
- Assume that the dimensions of $M$ and $N$ are multiples of BLOCK_SIZE and square shape.

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
  - 6,1024*6124 P Matrix gives 64*64 = 4096 Thread Blocks
- Each thread block performs $2\times256 = 512$ float loads from global memory for $256 \times (2\times16) = 8,192$ 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.

Shared Memory Usage

- Assume each SMP has 16KB shared memory and
  BLOCK_SIZE = 16
  - Each Thread Block uses $2\times256\times4B = 2KB$ of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing.
  - For BLOCK_SIZE = 16, this allows up to $8\times512 = 4,096$ 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 $2\times32\times32\times4B = 8KB$ shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time.

L7: Memory Hierarchy II
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
__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);
Ns[ty][tx] = GetMatrixElement(Nsub, tx, ty);
```

CUDA Code - Compute Result

```c
// Synchronize to make sure the sub-matrices are loaded
__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

```c
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