Lecture 8
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Using Shared Memory
Coalesced memory accesses
Avoiding bank conflicts
Announcements

•
Today’s lecture

• Using shared memory to increase memory locality in matrix multiply
• Memory coalescing
• Avoiding bank conflicts
Recapping form last time

- The naïve kernel loads every value of A and B N times from global memory.
- The floating point intensity is much lower than that of the processor: the kernel is memory bound.
- We ran at 116 GFlops with a 32 x 32 thread block.
- Stampede’s processors are faster: 8 cores run at 160 Gflops!
Improving locality in matrix multiply

• Naïve algorithm
  - Each thread loads all the data it needs, independently loads a row and column of input
  - Each matrix element loaded multiple times
  - Each thread computes 1 MAD for each pair of 2 loads +1 store

• Tiled algorithm with shared memory
  - Divide the matrices into tiles, as in Assignment #1
  - Threads cooperate to load a tile of A&B into on-chip shared memory
  - Each tile in the result matrix C corresponds to a thread block
  - Each thread performs: b mpy-adds + 1 load + 1 store
Recall the conventional tiled algorithm

- Each tile of C is the sum of ... matrix products of ... successive tiles from rows of A and columns of B
- The technique for tiling the code is called *strip mining*

```c
for (ii=0; ii<N; ii+=BSZ)
    for (jj=0; jj<N; jj+=BSZ){
        for (i=ii; i<ii+BSZ; i++){
            for (j=jj; j<jj+BSZ; j++){
                Cij = C[ii+i][jj+j];
                for (int kk=0; kk<N; kk+=BSZ)
                    for (int k=kk; k<kk+BSZ; k++)
                        Cij+= A[i][k] * B[k][j];
                C[i][j] = Cij
            }
        }
    }
```

- Each tile of C is the sum of matrix products of successive tiles from rows of A and columns of B
- The technique for tiling the code is called *strip mining*
Toward the CUDA algorithm

- We’ll restructure the code in a sequence of steps to arrive at the CUDA implementation
  - Rearrange loop headers
  - Tile blocking factors TI = TJ = TK

```c
for (ii=0; ii<N; ii+=TI)
    for (i=ii; i<ii+TI; i++){
        for (jj=0; jj<N; jj+=TJ){
            for (j=jj; j<jj+TJ; j++){
                Cij = 0;
                for (int kk=0; kk<N; kk+=TK)
                    for (int k=kk; k<kk+TK; k++)
                        Cij+= A[i][k]*B[k][j];
                C[i][j] = Cij
            }
        }
    }
```
Toward the CUDA algorithm

- We’ll restructure the code in a sequence of steps to arrive at the CUDA implementation
  - Rearrange loop headers
  - Tile blocking factors TI = TJ = TK
  - Block and thread dimensions require unit strides (I and J only)
  - Linearize the array indices

```c
for (ii=0; ii<N/TI; ii++)
    for (i=0; i<TI; i++){
        for (jj=0; jj<N/TJ; jj++){
            for (j=jj; j<TJ; j++)
                Cij = 0;
            for (int kk=0; kk<N; kk+=TK)
                for (int k=kk; k<kk+TK; k++)
                    Cij+= A[ii*TI+i][k]*B[k][jj*TJ+j];
            C[ii*TI+i][jj*TJ+j] = Cij
        }
    }
}
```
Converting to CUDA

- This code is not yet tiled
- Variable names deviate from the text
- Note conventions for indexing arrays & thread blocks
- Block and thread loops vanish
- Linearize the array accesses
- Each SMX computes a part of C from a block of adjacent rows of A and a block of adjacent columns B

Dim3 (dimGrid(N/TI, N/TJ); dim3 dimBlock(TI,TJ)
Matmul<<digGrid,dimBlock>>>(A,B,C);

__global__ mmpy(double *A, double *B, double *C){
    int i = threadIdx.y, j = threadIdx.x;
    int ii = blockIdx.y, jj = blockIdx.x
    double Cij = 0;
    for (int kk=0; kk<N; kk+=TK)
        for (int k=kk; k<kk+TK; k++)
            Cij+= A[(ii*TI+i)N + k]*B[k*N + jj*TJ+j];
    C[(ii*TI+i)N + jj*TJ+j] = Cij;
}
The tiled CUDA algorithm

- We tile for shared memory
- Each SMX computes a tile of C from tiles of adjacent rows of A and of adjacent columns B
- Synchronize before and after updating C in shared memory

```c
Dim3 (dimGrid(N/TI, N/TJ);  dim3 dimBlock(TI,TJ)
Matmul<<<digGrid,dimBlock>>>(A,B,C);
__global__ mmpy(double *A, double *B, double *C){
  int i = threadIdx.y,    j = threadIdx.x;
  int ii = blockIdx.y, jj = blockIdx.x
  double Cij = 0;
  for (int kk=0; kk<N; kk+=TK){
    As[j][i] = A[(ii*TI+i)*N + kk*TK+j];
    Bs[j][i] = B[(kk*TK+i)*N + jj*TJ+j];
    __syncthreads();
    for (int k=0; k<TK; k++)
      Cij+= As[i][k] * Bs[k][i];
    __syncthreads();
    C[(ii*TI+i)*N + jj*TJ+j] = Cij;
```

Scott B. Baden / CSE 260, UCSD / Fall '15
Tiled Code – Final version

• Code on page 112 (slight naming variations)
• Rename i/ii j/jj as ty/by  tx/bx, TW= tile width
• All variables name match except for As/Bs

```c
__global__ mmpy(double *A, double *B, double *C){
    __shared__ double A[TW][TW], A[TW][TW];
    int tx = threadIdx.y, ty = threadIdx.x;
    int by = blockIdx.y, bx = blockIdx.x;
    int I = by*TW + ty, J = bx*TW+tx;
    double Cij = 0;
    for (int kk=0; kk<N/TW; kk++){
        As[ty][tx] = A[I*N + kk*TW+tx];
        Bs[ty][tx] = B[(kk*TW+ty)*N + J];
        __syncthreads();
    }
    for (int k=0; k<TW; k++)
        Cij+= As[ty][k] * Bs[k][tx];
    __syncthreads();
    C[I*N + J] = Cij;
}
```
Structure of the blocked algorithm

- Threads cooperate to load a block of A&B into on-chip, shared memory
- Each thread in the block performs the $ijk$ loop within shared memory

```c
for (int kk=0; kk < N/TW; kk++){
    Load blocks of A & B
    __syncthreads();
    for (int k=0; k< TW; k++)
        C_{ij} += A_{ty}[k] * B_{s[k][tx]}
    __syncthreads();
}
C[I*N+J] = C_{ij};
```
Results with shared memory

• N=512, double precision
• Shared memory enable us to improve performance! 116 GF $\rightarrow$ 227 GF
• What happened?
  • We reduced global memory accesses
  • Warps benefit from accessing a contiguous aligned region of 128 or 256 bytes
• The memory accesseses coalesced
Today’s lecture

• Using shared memory to increase memory locality in matrix multiply
• Memory coalescing
• Avoiding bank conflicts
Memory interleaving

- Memories are slow
- We load several words rather than just 1 word
- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?
Global memory coalescing

- Global memory accesses in units of 32, 64, 128 B
- Does not require sequential accesses by threads—only need to fall within the same 128B segment
- Accesses organized by warp
- Concurrent accesses by a warp’s reads coalesce into K transactions = # different cache lines (128B) covered by the accesses
- Access not cached in L1: 32B at a time (x4) in L2

\[
\text{shrd[threadIdx.x]} = \text{gbl[blockIdx.x*blockDim.x+threadIdx.x]} \quad \text{YES!}
\]
\[
\text{shrd[threadIdx.x]} = \text{gbl[blockDim.x+ blockIdx.x*threadIdx.x]} \quad \text{NOT!}
\]
Global Memory

• If accessed word > 4 bytes, warp’s memory request split into separate, independently issued 128-byte memory requests
• Non-atomic, concurrent writes within a warp: writer not defined
• cudaMalloc() is guaranteed to be aligned to at least 256 bytes
Memory coalescing

- Simplest: addresses fall within the same 128B segment
- Accesses organized by warps (32 threads)
How does memory coalescing work (compute capability ≥1.2)

- Find the segment containing the address request of the lowest numbered active thread
- Find all other active threads requesting in same segment
- Reduce transaction size (if possible)
- Mark the serviced threads as inactive
- Repeat until all threads in the warp are complete
- Handles permutations, too

1 transaction – 128B segment

2 transactions – 2 x 128B segments

Nvidia Cuda C Programming Guide: Appendix G.4.2
Coalescing with 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns, because we add tx to the index

\[ I = by \times TW + ty; \]
\[ J = bx \times TW + tx; \]
\[ As[ty][tx] = A[I \times N + kk \times TW + tx]; \]
\[ Bs[ty][tx] = B[(kk \times TW + ty) \times N + J]; \]

- Accesses by threads in a block along a column don’t coalesce

\[ a[tx][ty] = A[I \times N + kk \times TW + ty]; \]
\[ b[ty][tx] = B[J + N \times (kk \times TW + tx)]; \]
\[ c += a[kk][ty] \times b[tx][kk]; \]

- No real benefit from shared memory
- Performance drops to 126 GF (recall 116 GF → 227 GF)
Today’s lecture

• Using shared memory to increase memory locality in matrix multiply
• Memory coalescing
• Avoiding bank conflicts
Shared memory banks

- A load or store of $n$ addresses spanning $n$ distinct memory banks can be serviced simultaneously, effective bandwidth $n$ times than single bank bandwidth

- Multiple addresses map to same memory bank
  - Accesses are serialized
  - Hardware splits request into as many separate conflict-free requests as necessary
    Exception: if all access the same address: broadcast

- Devices of compute capability 2.x and beyond have the additional ability to multicast shared memory accesses

- See *CUDA C Best Practices Guide*

- $N=32$ for Kepler
Shared memory bank access

- Load/store of \( n \) addresses spanning \( n \) distinct memory banks can be serviced simultaneously, effective BW = \( \times n \) a single bank’s
- Each bank can service 1 address / cycle (broadcast, too)
- Access to shared memory is fast unless…
  - 2 or more instructions in a warp access the same bank: we have a conflict
  - Exception: not if accesses to the same 32 bit word: broadcast
- For writes, only one thread writes, writer is undefined

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = a[idx]+1.0f;
```
Conflict free access

- Consider
  ```c
  __shared__ float shared[256];
  float foo = shared[base + s * threadIdx.x];
  ```
- If $s$ has no common factors with the number of banks (32), then there are no conflicts ($s$ is odd)
Identifying bank conflicts

- Traditional wisdom for exploiting cache locality can result in bank conflicts
- What if a thread loads 2 consecutive array elements?
  ```c
  int tid = threadIdx.x;
  shared[2*tid] = global[2*tid];
  shared[2*tid+1] = global[2*tid+1];
  ```
- To avoid conflicts
  ```c
  shared[tid] = global[tid];
  shared[tid + blockDim.x] = global[tid + blockDim.x];
  ```

A memory system with 4 banks
Kepler’s hared memory design

- 32 banks, 8 bytes wide
  - Bandwidth = 8 bytes/bank/clock/SMAX
  - 256 bytes/clock/SMX
  - Optimal: each thread accesses a different bank
  - Exception: multicast (all access the same bank)

- There are 2 modes
  - 4 byte access (default):
    8 byte bandwidth for some access patterns
  - 8 bytes access
  - Controlled with cudaDeviceSetSharedMemConfig()
Fin