L15: CUDA, cont. Memory Hierarchy and Examples

November 1, 2011

Administrative

- Final projects
- Programming Assignment

Programming Assignment 3, Due 11:59PM Nov. 7

- Purpose:
  - Synthesize the concepts you have learned so far
  - Data parallelism, locality and task parallelism
  - Image processing computation adapted from a real application
- Turn in using handin program on CADE machines
  - Handin cs4961 proj3 <file>
  - Include code + README
- Three Parts:
  1. Locality optimization (50%): Improve performance using locality optimizations only (no parallelism)
  2. Data parallelism (20%): Improve performance using locality optimizations plus data parallel constructs in OpenMP
  3. Task parallelism (30%): Code will not be faster by adding task parallelism, but you can improve time to first result. Use task parallelism in conjunction with data parallelism per my message from the previous assignment.

Here's the code

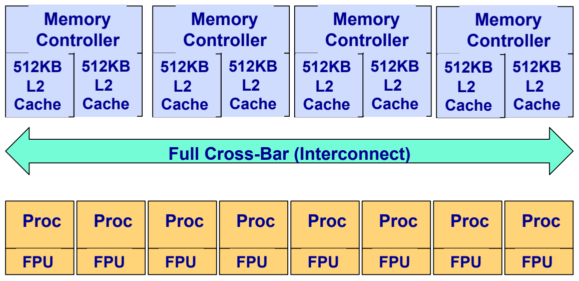
```
… Initialize th[i][j] = 0 …

 /* compute array convolution */
 for(m = 0; m < IMAGE_NROWS - TEMPLATE_NROWS + 1; m++){
   for(n = 0; n < IMAGE_NCOLS - TEMPLATE_NCOLS + 1; n++){
     for(i=0; i < TEMPLATE_NROWS; i++){
       for(j=0; j < TEMPLATE_NCOLS; j++){
         if(mask[i][j] != 0) {
             th[m][n] += image[i+m][j+n];
         }
       }
     }
   }
 }
/* scale array with bright count and template bias */
…
th[i][j] = th[i][j] * bc – bias;
```

Things to think about

- Beyond the assigned work, how does parallelization affect the profitability of locality optimizations?
- What happens if you make the IMAGE SIZE larger (1024x1024 or even 2048x2048)?
  - You'll need to use "unlimit stacksize" to run these.
- What happens if you repeat this experiment on a different architecture with a different memory hierarchy (in particular, smaller L2 cache)?
- How does SSE or other multimedia extensions affect performance and optimization selection?

Shared Memory Architecture 2: Sun Ultrasparc T2 Niagara (water)



More on Niagara

- Target applications are server-class, business operations
- Characterization:
  - Floating point?
  - Array-based computation?
- Support for VIS 2.0 SIMD instruction set
- 64-way multithreading (8-way per processor, 8 processors)

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
  - Optimizations: Data placement in nearby portion of memory hierarchy (focus on registers and shared memory in this class)
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
  - Optimizations: Global memory coalescing, avoid shared memory bank conflicts
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain
  - Requires sufficient reuse to amortize cost of copies to shared memory, for example

Global Memory Accesses

- Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities
- Given an address to load or store, memory returns/updates "segments" of either 32 bytes, 64 bytes or 128 bytes
- Maximizing bandwidth:
  - Operate on an ***entire*** 128 byte segment for each memory transfer

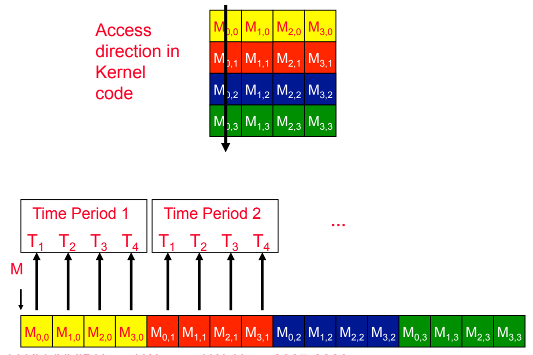
Understanding Global Memory Accesses

Memory protocol for compute capability 1.2* (CUDA Manual 5.1.2.1)

- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory and mark threads as "inactive"
- Repeat until all threads ***in half-warp*** are serviced

*Includes Tesla and GTX platforms

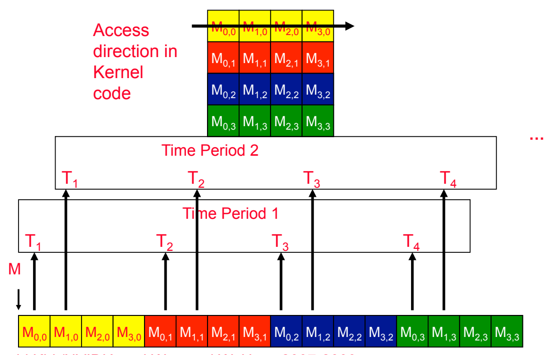
Memory Layout of a Matrix in C



Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Time Period 1
T_1 T_2 T_3 T_4
Time Period 2
T_1 T_2 T_3 T_4
...
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

Memory Layout of a Matrix in C



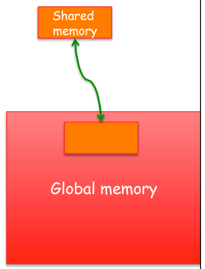
Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Time Period 2
T_1 T_2 T_3 T_4
...
Time Period 1
T_1 T_2 T_3 T_4
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

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

Familiar concept???



THE UNIVERSITY OF UTAH

Mechanics of Using Shared Memory

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

```
extern __shared__ float  d_s_array[];

/* a form of dynamic allocation */
/* MEMSIZE is size of per-block  */
/* shared memory*/
__host__ void outerCompute() {
   compute<<<gs,bs,MEMSIZE>>>();
}
__global__ void compute() {
     d_s_array[i] = …;
}
```

```
__global__ void compute2() {
   __shared__ float d_s_array[M];

   /* create or copy from global memory */
   d_s_array[j] = …;

  /* write result back to global memory */
   d_g_array[j] =  d_s_array[j];
}
```

THE UNIVERSITY OF UTAH

Bandwidth to Shared Memory: Parallel Memory Accesses

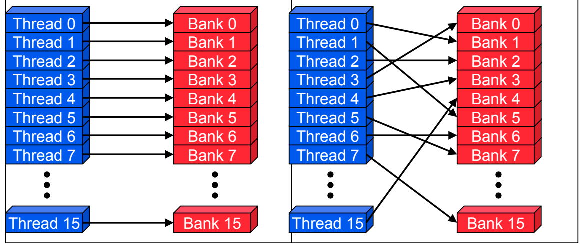
- Consider each thread accessing a different location in shared memory
- Bandwidth maximized if each one is able to proceed ***in parallel***
- Hardware to support this
  - ***Banked memory:*** each bank can support an access on every memory cycle

THE UNIVERSITY OF UTAH

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation

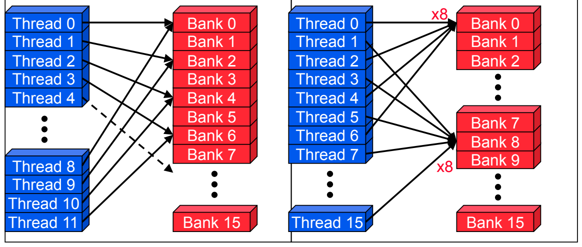


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign

THE UNIVERSITY OF UTAH

Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2
- 8-way Bank Conflicts
  - Linear addressing stride == 8



How addresses map to banks on G80 (older technology)

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

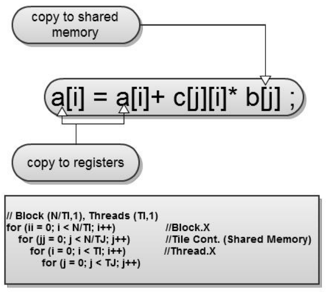
Example: Matrix vector multiply

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

Remember to:
- Consider data dependences in parallelization strategy to avoid race conditions
- Derive a partition that performs global memory coalescing
- 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



```
// Block (N/TI,1), Threads (TI,1)
for (ii = 0; i < N/TI; i++)           //Block.X
   for (jj = 0; j < N/TJ; j++)        //Tile Cont. (Shared Memory)
      for (i = 0; i < TI; i++)        //Thread.X
         for (j = 0; j < TJ; j++)
```

Resulting CUDA code (Automatically Generated by our Research Compiler)

```
__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];
    }
    __synchthreads();
  }
  a[tx + 32 * bx] = acpy;
}
```

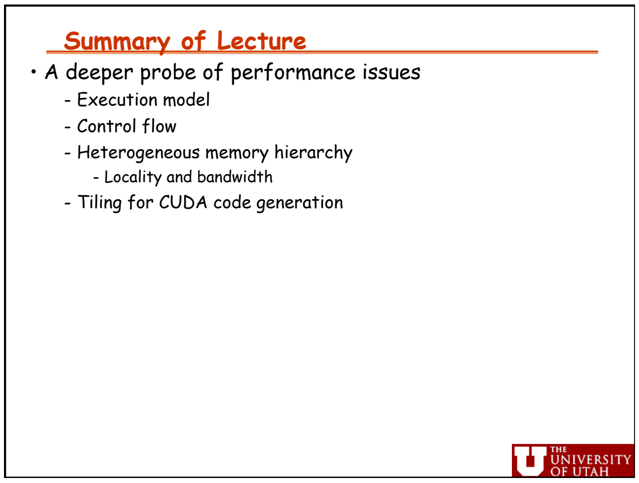
What happens if we transpose C?

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

What else do we need to worry about?

Resulting CUDA code for Transposed Matrix Vector Multiply

```
__global__ mv_GPU(float* a, float* b, float** c) {
  int bx = blockIdx.x; int tx = threadIdx.x;
  __shared__ float bcpy[16];
  __ shared__ float P1[16][17];  //pad
  double acpy = a[tx + 16 * bx];
  for (k = 0; k < 16; k++) {
     bcpy[tx] = b[16 * k + tx];
    for (l=0; l<16; l++) {
       _P1[l][tx] = c[k*bx+l][16*bx+tx];   // copy in coalesced order
    }
     __syncthreads();
    //this loop is actually fully unrolled
    for (j = 16 * k; j <= 16 * k + 16; j++) {
       acpy = acpy + _P1[tx][j] * bcpy[j];
    }
    __synchthreads();
  }
```



Summary of Lecture

- A deeper probe of performance issues
  - Execution model
  - Control flow
  - Heterogeneous memory hierarchy
    - Locality and bandwidth
  - Tiling for CUDA code generation