Review for Midterm

Administrative

- Midterm
  - In class April 4, open notes
  - Review notes, readings and review lecture (before break)
  - Will post prior exams
- Design Review
  - Intermediate assessment of progress on project, oral and short
  - Tentatively April 11 and 13
- Final projects
  - Poster session, April 27 (dry run April 25)
  - Final report, May 4

Midterm Exam Monday, April 4

- Goal is to reinforce understanding of CUDA and NVIDIA architecture
- Material will come from lecture notes and assignments
- In class, should not be difficult to finish
- Open notes, but no computers

Parts of Exam

I. Definitions
   - A list of 5 terms you will be asked to define

II. Short Answer (4 questions, 20 points)
   - Understand basic GPU architecture: processors and memory hierarchy
   - High level questions on more recent "pattern and application" lectures

III. Problem Solving
   - Analyze data dependences and data reuse in code and use this to guide CUDA parallelization and memory hierarchy mapping
   - Given some CUDA code, indicate whether global memory accesses will be coalesced and whether there will be bank conflicts in shared memory
   - Given some CUDA code, add synchronization to derive a correct implementation
   - Given some CUDA code, provide an optimized version that will have fewer divergent branches

IV. (Brief) Essay Question
   - Pick one from a set of 4

Syllabus

L1: Introduction and CUDA Overview
- Not much there…

L2: Hardware Execution Model
- Difference between a parallel programming model and a hardware execution model
- SIMD, MIMD, SIMT, SPMD
- Performance impact of fine-grain multithreaded architecture
- What happens during the execution of a warp?
- How are warps selected for execution (scoreboarding)?

L3 & L4: Memory Hierarchy: Locality and Data Placement
- Memory latency and memory bandwidth optimizations
- Reuse and locality
- What are the different memory spaces on the device, who can read/write them?
- How do you tell the compiler that something belongs in a particular memory space?
- Tiling transformation (to fit data into constrained storage): Safety and profitability

Syllabus

L5 & L6: Memory Hierarchy III: Memory Bandwidth Optimization
- Tiling (for registers)
- Bandwidth – maximize utility of each memory cycle
- Memory accesses in scheduling (half-warp)
- Understanding global memory coalescing (for compute capability < 1.2 and > 1.2)
- Understanding shared memory bank conflicts

L7: Writing Correct Programs
- Race condition, dependence
- What is a reduction computation and why is it a good match for a GPU?
- What does __syncthreads () do? (barrier synchronization)
- Atomic operations
- Memory Fence Instructions
- Device emulation mode

Syllabus

L8: Control Flow
- Divergent branches
- Execution model
- Warp vote functions

L9: Floating Point
- Single precision versus double precision
- IEEE Compliance: which operations are compliant?
- Intrinsics vs. arithmetic operations, what is more precise?
- What operations can be performed in 4 cycles, and what operations take longer?

L10: Dense Linear Algebra on GPUs
- What are the key ideas contributing to CUBLAS 2.0 performance
- Concepts: high thread count vs. coarse-grain threads. When to use each?
- Transpose in shared memory plus padding trick

L11: Sparse Linear Algebra on GPUS
- Different sparse matrix representations
- Stencil computations using sparse matrices

Syllabus

L12&L13: Application case studies
- Host tiling for constant cache (plus data structure reorganization)
- Replacing trig function intrinsic calls with hardware implementations
- Global synchronization for MPM/GIMP

L14: Dynamic Scheduling
- Task queues
- Static queues, dynamic queues
- Wait-free synchronization

L15: Tree-based algorithms
- Flattening tree data structures
- Scheduling on a portion of the architecture

Exam: Problem III.a

a. **Managing memory bandwidth**

Given the following CUDA code, how would you rewrite to improve bandwidth to global memory and, if applicable, shared memory? Explain your answer for partial credit. Assume c is stored in row-major order, so c[i][j] is adjacent to c[i][j+1].

```
N = 512;
NUMBLOCKS = 512/64;

float a[512], b[512], c[512][512];

__global compute(float a, float *b, float *c) {
int tx = threadIdx.x;
int bx = blockIdx.x;

for (j = bx*64; j< (bx*64)+64; j++)
   a[tx] = a[tx] - c[tx][j] * b[j];
}
```


Exam: Problem III.a

a. **Managing memory bandwidth**

Given the following CUDA code, how would you rewrite to improve bandwidth to global memory and, if applicable, shared memory? Explain your answer for partial credit. Assume c is stored in row-major order, so c[i][j] is adjacent to c[i][j+1].

```
N = 512;
NUMBLOCKS = 512/64;

float a[512], b[512], c[512][512];

__global compute(float a, float *b, float *c) {
int tx = threadIdx.x;
int bx = blockIdx.x;

for (j = bx*64; j< (bx*64)+64; j++)
   a[tx] = a[tx] - c[tx][j] * b[j];
}
```

How to solve?

Copy "c" to shared memory in coalesced order

Tile in shared memory

Copy a to register

Copy b to shared memory, constant memory or texture memory

Exam: Problem III.a

```
N = 512;  NUMBLOCKS = 512/64;

float a[512], b[512], c[512][512];
float tmpa;
__global compute(float a, float *b, float *c) {
__shared__ ctmp[1024+32]; // let's use 32x32
                          // pad for bank conflicts
int tx = threadIdx.x;
int bx = blockIdx.x;
 tmpa = a[tx];
Pad1 = tx/32; Pad2 = j/32;
for (jj = bx*64; jj< (bx*64)+64; jj+=32)
  for (j=jj; j<jj+2; j++)
     Ctmp[j*512+tx+pad1] = c[j][tx];
     __syncthreads();
     tmpa = tmpa - ctmp[tx*512 + j + pad2] * b[j];
```

How to solve?

Copy "c" to shared memory in coalesced order

Tile in shared memory

Copy a to register

Copy b to shared memory, constant memory or texture memory

Exam: Problem III.b

b. **Divergent Branch**

Given the following CUDA code, describe how you would modify this to derive an optimized version that will have fewer divergent branches. …

```
Main() {
    float h_a[1024], h_b[1024];
    …
    /* assume appropriate cudaMalloc called to create d_a and d_b, and d_a is */
    /* initialized from h_a using appropriate call to cudaMemcpy */
    dim3 dimblock(256);
    dim3 dimgrid(4);
    compute<<<dimgrid, dimblock,0>>>(d_a,d_b);
    /* assume d_b is copied back from the device using call to cudaMemcpy */
}

__global__ compute (float *a, float *b) {
float a[4][256], b[4][256];
int tx = threadIdx.x; bx = blockIdx.x;
if (tx % 16 == 0)
   (void) starting_kernel (a[bx][tx], b[bx][tx]);
else /* (tx % 16 > 0) */
   (void) default_kernel (a[bx][tx], b[bx][tx]);
}
```

Key idea:
Separate multiples of 16 from others

Problem III.b

Approach:

Renumber thread to concentrate case where not divisible by 16

if (tx < 240) t = tx + (tx/16) + 1;

Else t = (tx – 240) * 16;

- Now replace tx with t in code
- Only last "warp" has divergent branches

Exam: Problem III.c

c. **Tiling**
The following sequential image correlation computation compares a region of an image to a template. Show how you would tile the image and threshold data to fit in 128MB global memory and the template data to fit in a 16KB shared memory? Explain your answer for partial credit.

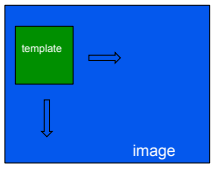
```
TEMPLATE_NROWS = TEMPLATE_NCOLS = 64;
IMAGE_NROWS = IMAGE_NCOLS = 5192;

int image[IMAGE_NROWS][IMAGE_NCOLS], th[IMAGE_NROWS][IMAGE_NCOLS];
int template[TEMPLATE_NROWS][TEMPLATE_NCOLS];

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(abs(image[i+m][j+n] – template[i][j]) < threshold)
                              th[m][n]+= image[i+m][j+n]
     }
    }
  }
 }
```


View of Computation

- Perform correlation of template with portion of image
- Move "window" horizontally and downward and repeat



Problem III.c

i. How big is image and template data?

Image = 5192^2 * 4 bytes/int = 100 Kbytes

Th = 100 Kbytes

Template = 64^2 * 4 bytes /int = exactly 16KBytes

Total data set size = 216 Kbytes so fits in global memory and no need for tiling at this level

Template data does not fit in shared memory due to other things placed there…

ii. Partitioning to support tiling for shared memory

Hint to exploit reuse on template by copying to shared memory

Could also exploit reuse on portion of image

Dependences only on th (a reduction)

Problem III.c

(iii) Need to show tiling for template

Can copy into shared memory in coalesced order

Copy half or less at a time

Exam: Problem III.d

d. **Parallel partitioning and synchronization (LU Decomposition)**

Without writing out the CUDA code, consider a CUDA mapping of the LU Decomposition sequential code below. Answer should be in three parts, providing opportunities for partial credit: (i) where are the data dependences in this computation? (ii) how would you partition the computation across threads and blocks? (iii) how would you add synchronization to avoid race conditions?

```
float a[1024][1024];

for (k=0; j<1023; k++) {
   for (i=k+1; i<1024; i++)
      a[i][k] = a[i][k] / a[k][k];
   for (i=k+1; i<1024; i++)
      for (j=k+1; j<1024; j++)
         a[i][j] = a[i][j] - a[i][k]*a[k][j];
}
```

Key Features of Solution:

i. Dependences:
True <a[i][j],a[k][k]>,<a[i][j],a[i][k]> carried by k
True <a[i][j],a[k][k]>, <a[i][j],a[i][k]>, carried by k
True <a[i][j],a[i][j]>, <a[i][j], a[k][j]>, carried by k

ii. Partition:
Merge i loops, interchange with j, partition j
Across blocks/threads (sufficient ||ism?) or
Partition I dimension across threads
using III.a. trick
Load balance? Repartition on host

iii. Synchronization:
On host

Exam: Problem III.d

d. **Parallel partitioning and synchronization (LU Decomposition)**

Without writing out the CUDA code, consider a CUDA mapping of the LU Decomposition sequential code below. Answer should be in three parts, providing opportunities for partial credit: (i) where are the data dependences in this computation? (ii) how would you partition the computation across threads and blocks? (iii) how would you add synchronization to avoid race conditions?

```
float a[1024][1024];

for (k=0; j<1023; k++) {
   for (i=k+1; i<1024; i++)
      a[i][k] = a[i][k] / a[k][k];
   for (i=k+1; i<1024; i++)
      for (j=k+1; j<1024; j++)
         a[i][j] = a[i][j] - a[i][k]*a[k][j];
}
```