Name _____

EE 7700-2

Take-Home Final Examination

Tuesday, 8 May 2012 to Friday, 11 May 2012

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else. Any questions or concerns about problems should be directed to Dr. Koppelman.

Some questions in this exam are based on the paper "Debunking the 100X GPU vs. CPU Myth" by Lee *et al*, which will be referred to as Lee 2010 in the exam. The paper itself and a full reference are linked to the course references page, as are other papers cited in the exam.

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Section 4.3.1 of the paper discusses the performance of bandwidth-limited programs.

(*a*) Based on information and discussion in the paper explain why, where requested below, the i7 performs better on bandwidth-limited programs with medium-size working sets while the GTX 280 does better on those with small- and large-size working sets. Succinct answers that omit irrelevant material are preferred.

☐ GTX 280 is better on bandwidth-limited programs with small working sets because ...

☐ Intel i7 is better on certain bandwidth-limited programs with medium-size working sets because ...

☐ GTX 280 is better on bandwidth-limited programs with large working sets because ...

(*b*) The i7 can only run certain medium-size working set bandwidth-limited programs faster than the GTX 280, others the GTX 280 will run faster. Consider a difference having to do with the number of times each data item is accessed. Given that, describe the proprieties of a medium-size working set bandwidth limited that the GTX 280 can run faster. Answers with a number are preferred.

☐ The GTX 280 can run a medium-size working set bandwidth-limited program if the program ...

Problem 2: [25 pts] A goal of scientific programmers is to come as close as possible to the FLOP-limit of a device, this often means coaxing the compiler to use as few non-FP instructions as possible. (See code in `https://svn.ece.lsu.edu/svn/gp/cuda/matrix-mult/`.) Consider the inner loop of the matrix multiply based on Volkov's algorithm (CUDA code appears immediately below, Fermi machine code starts on the next page).

```
// Note: kernel name mm_blk_cache_a_local_t in file matrix-mult-kernel.cu
      /// A Block Loop:  Each iteration uses a blk x blk submatrix of A.
      //  The iterations move across columns.
      //
      while ( b_idx < array_size )
        {
          __syncthreads();
          s[b_sidx_copy] = a[a_idx];
          a_idx += dim_block;
          __syncthreads();

          /// B Value Loop: Each iteration uses one value of B.
#         pragma unroll
          for ( int kk = 0; kk < dim_block; kk++, b_idx += row_stride )
            {
              float b_val = b[b_idx];
              for ( int ii = 0; ii < dim_block; ii++ )
                cloc[ii] +=  s[ kk + dim_block * ii ] * b_val;
            }
        }
```

Volkov tuned this algorithm for a CC 1.X device, in which arithmetic instructions could get one operand from shared memory. (See mm-gt200.sass in the repo for that machine code.) But in CC 2.X devices load instructions cannot access shared memory.

For each question below, assume that all instructions can be issued at a rate of 32 per cycle per multiprocessor (as in a CC 2.0 device).

(*a*) Determine the fraction of peak FLOPS obtained based on the loop assuming that execution proceeds at full speed (there are no stalls). (Machine code appears on the following pages.) Only include `FFMA` as contributing to FLOPS.

(*b*) How much faster would the code below run if floating-point instructions could read shared memory?

(*c*) When developing the Fermi instruction set, some people might have objected to the decision to remove the ability to read source operands from shared memory in floating-point instructions, because programs would have to make up for this ability by inserting shared load instructions.

A response to those objections might have been, "It's not as bad as one shared load per fp instruction." Explain, given the machine code below.

Problem 2, continued: Listing for previous problem.

Note: The full listing is at: `https://svn.ece.lsu.edu/svn/gp/cuda/matrix-mult/mm-fermi.sass` *look for the routine* `_Z22mm_blk_cache_a_local_tILi3EEvv`.

```
/*0148*/        BAR.RED.POPC RZ, RZ;
/*0150*/        MOV R1, c [0x2] [0x18];
/*0158*/        IMUL.HI R0, R37, 0x4;
/*0160*/        IMAD R2.CC, R37, 0x4, R1;
/*0168*/        IADD R37, R37, 0x8;
/*0170*/        IADD.X R3, R0, c [0x2] [0x1c];
/*0178*/        LD.E R0, [R2];
/*0180*/        STS [R40], R0;
/*0188*/        BAR.RED.POPC RZ, RZ;
/*0190*/        MOV32I R1, 0x4;
/*0198*/        IMUL.HI R0, R38, 0x4;
/*01a0*/        MOV R41, c [0x2] [0x8];
/*01a8*/        IMAD R2.CC, R38, R1, c [0x2] [0x20];
/*01b0*/        LDS.128 R8, [0x0];
/*01b8*/        IMUL.HI R44, R41, 0x4;
/*01c0*/        IADD.X R3, R0, c [0x2] [0x24];
/*01c8*/        IMAD R0.CC, R41, 0x4, R2;
/*01d0*/        LDS.128 R4, [0x20];
/*01d8*/        LD.E R46, [R2];
/*01e0*/        IADD.X R1, R3, R44;
/*01e8*/        IMAD R32.CC, R41, 0x4, R0;
/*01f0*/        LDS.128 R12, [0x60];
/*01f8*/        LD.E R45, [R0];
/*0200*/        IADD.X R33, R1, R44;
/*0208*/        ISCADD R38, R41, R38, 0x3;
/*0210*/        LD.E R43, [R32];
/*0218*/        LDS.128 R0, [0x40];
/*0220*/        ISETP.LT.AND P0, pt, R38, c [0x2] [0x0], pt;
/*0228*/        IMAD R32.CC, R41, 0x4, R32;
/*0230*/        IADD.X R33, R33, R44;
/*0238*/        FFMA.FTZ R0, R0, R46, R21;
/*0240*/        FFMA.FTZ R8, R8, R46, R18;
/*0248*/        FFMA.FTZ R4, R4, R46, R17;
/*0250*/        FFMA.FTZ R1, R1, R45, R0;
/*0258*/        FFMA.FTZ R0, R12, R46, R16;
/*0260*/        LDS.128 R16, [0x80];
/*0268*/        FFMA.FTZ R8, R9, R45, R8;
/*0270*/        FFMA.FTZ R0, R13, R45, R0;
/*0278*/        LD.E R13, [R32];
/*0280*/        FFMA.FTZ R4, R5, R45, R4;
/*0288*/        FFMA.FTZ R9, R14, R43, R0;
/*0290*/        FFMA.FTZ R20, R16, R46, R20;
/*0298*/        IMAD R0.CC, R41, 0x4, R32;
/*02a0*/        FFMA.FTZ R2, R2, R43, R1;
/*02a8*/        FFMA.FTZ R5, R17, R45, R20;
/*02b0*/        IADD.X R1, R33, R44;
/*02b8*/        LDS.128 R20, [0xa0];
/*02c0*/        FFMA.FTZ R6, R6, R43, R4;
/*02c8*/        IMAD R4.CC, R41, 0x4, R0;
/*02d0*/        FFMA.FTZ R8, R10, R43, R8;
/*02d8*/        LD.E R32, [R0];
/*02e0*/        FFMA.FTZ R10, R18, R43, R5;
/*02e8*/        IADD.X R5, R1, R44;
/*02f0*/        FFMA.FTZ R24, R20, R46, R24;
/*02f8*/        IMAD R0.CC, R41, 0x4, R4;
/*0300*/        LD.E R33, [R4];
/*0308*/        FFMA.FTZ R12, R21, R45, R24;
/*0310*/        LDS.128 R24, [0xc0];
/*0318*/        IADD.X R1, R5, R44;
/*0320*/        FFMA.FTZ R4, R22, R43, R12;
/*0328*/        FFMA.FTZ R28, R24, R46, R28;
/*0330*/        FFMA.FTZ R14, R25, R45, R28;
/*0338*/        LDS.128 R28, [0xe0];
/*0340*/        FFMA.FTZ R5, R26, R43, R14;
/*0348*/        FFMA.FTZ R16, R28, R46, R42;
/*0350*/        LD.E R42, [R0];
/*0358*/        FFMA.FTZ R16, R29, R45, R16;
/*0360*/        IMAD R0.CC, R41, 0x4, R0;
/*0368*/        FFMA.FTZ R17, R30, R43, R16;
/*0370*/        IADD.X R1, R1, R44;
/*0378*/        LD.E R44, [R0];
/*0380*/        FFMA.FTZ R14, R7, R13, R6;
/*0388*/        FFMA.FTZ R18, R3, R13, R2;
/*0390*/        FFMA.FTZ R24, R23, R13, R4;
/*0398*/        FFMA.FTZ R28, R27, R13, R5;
/*03a0*/        LDS.128 R0, [0x10];
/*03a8*/        LDS.128 R4, [0x30];
/*03b0*/        FFMA.FTZ R12, R11, R13, R8;
/*03b8*/        FFMA.FTZ R16, R15, R13, R9;
/*03c0*/        FFMA.FTZ R20, R19, R13, R10;
/*03c8*/        FFMA.FTZ R43, R31, R13, R17;
/*03d0*/        LDS.128 R8, [0x50];
/*03d8*/        FFMA.FTZ R0, R0, R32, R12;
/*03e0*/        FFMA.FTZ R4, R4, R32, R14;
/*03e8*/        LDS.128 R12, [0x70];
/*03f0*/        FFMA.FTZ R8, R8, R32, R18;
/*03f8*/        FFMA.FTZ R0, R1, R33, R0;
/*0400*/        FFMA.FTZ R1, R5, R33, R4;
/*0408*/        FFMA.FTZ R4, R9, R33, R8;
/*0410*/        FFMA.FTZ R5, R12, R32, R16;
/*0418*/        LDS.128 R16, [0x90];
/*0420*/        FFMA.FTZ R5, R13, R33, R5;
/*0428*/        FFMA.FTZ R8, R16, R32, R20;
/*0430*/        LDS.128 R20, [0xb0];
/*0438*/        FFMA.FTZ R8, R17, R33, R8;
```

```
/*0440*/        FFMA.FTZ R9, R20, R32, R24;
/*0448*/        LDS.128 R24, [0xd0];
/*0450*/        FFMA.FTZ R0, R2, R42, R0;
/*0458*/        FFMA.FTZ R2, R10, R42, R4;
/*0460*/        FFMA.FTZ R4, R21, R33, R9;
/*0468*/        FFMA.FTZ R1, R6, R42, R1;
/*0470*/        FFMA.FTZ R6, R18, R42, R8;
/*0478*/        FFMA.FTZ R9, R24, R32, R28;
/*0480*/        LDS.128 R28, [0xf0];
/*0488*/        FFMA.FTZ R5, R14, R42, R5;
/*0490*/        FFMA.FTZ R8, R25, R33, R9;
/*0498*/        FFMA.FTZ R4, R22, R42, R4;
/*04a0*/        FFMA.FTZ R18, R3, R44, R0;
/*04a8*/        FFMA.FTZ R8, R26, R42, R8;
/*04b0*/        FFMA.FTZ R9, R28, R32, R43;
/*04b8*/        FFMA.FTZ R17, R7, R44, R1;
/*04c0*/        FFMA.FTZ R21, R11, R44, R2;
/*04c8*/        FFMA.FTZ R9, R29, R33, R9;
/*04d0*/        FFMA.FTZ R16, R15, R44, R5;
/*04d8*/        FFMA.FTZ R20, R19, R44, R6;
/*04e0*/        FFMA.FTZ R9, R30, R42, R9;
/*04e8*/        FFMA.FTZ R24, R23, R44, R4;
/*04f0*/        FFMA.FTZ R28, R27, R44, R8;
/*04f8*/        FFMA.FTZ R42, R31, R44, R9;
/*0500*/        @P0 BRA 0x148;
```

Problem 3: [25 pts] Lee mentions gather/scatter operations as a factor that distinguishes CC 1.X GPUs from the i7 (and the same holds for CC 2.0). The GPU code fragment below shows an example of a gather, the key feature being that the data items are non-consecutive.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
const int STRIDE = 8;
const int elt_idx_base = idx * STRIDE;
float a_sum = 0;
for ( int i=0; i< STRIDE; i++ ) a_sum += a[ elt_idx_base + i ];
c[ idx ] = a_sum;
```

(*a*) Those unfamiliar with the CUDA execution model might consider the accesses to array `a` consecutive. Why are the `a` accesses considered *non*-consecutive for CUDA execution and in the context of the gather/ scatter operations discussion from Section 4.3.4?

☐ The accesses are considered non-consecutive because.

(*b*) Why would the code above be slow on a CC 1.X device?

(*c*) Show how to use shared memory to execute the code quickly on a CC 1.X device.

Problem 3, continued: Continue to consider gather operations, but this time on a simpler program.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
const int STRIDE = 8;
const int elt_idx_base = idx * STRIDE;
float elt = a[ elt_idx_base ];
// ...
```

(*d*) In the corresponding CPU code using four-element vector registers, a thread would have to load four values (requiring four load instructions plus address computation instructions), then issue a pack instruction to put them in the SSE register. The paper claims thirteen instructions are needed (but not for this exact example). Suppose the GPU code needs only four instructions.

The GPU code will certainly execute more quickly, but the approximate number of operations (including address computation instructions) will be the same in the two devices. Explain why.

(*e*) The GPU's faster execution on this code comes at the expense of having more registers than are necessary. This register waste was emphasized in class when comparing NVIDIA GPUs to Larrabee/Knight's Corner/ Knight's Ferry/MIC. Explain what those "wasted" registers are being used for in the non-gather code below:

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
float a_sum = 0;
for ( int i=0; i< 8; i++ ) a_sum += a[ idx + i * thread_count ];
c[ idx ] = a_sum;
```

Problem 4: [25 pts] Near the end of Section 4.1 the paper explains that for their programs 4 to 8 warps work best (out of a maximum of 32), the reason given is increased pressure on (demand for) registers and on-chip memory resources with a larger number of warps.

(*a*) Explain why, say, 8 warps would work better than a larger number of warps when considering demands for shared memory. Try to be specific, perhaps using an example. *Hint: the intended answer is simple. Think about the radix sort.*

(*b*) Consider a program that frequently accesses global memory and so has alot of latency to hide. Explain why ordinarily, having more warps can hide more latency (going against what the paper said). Illustrate your answer with a timing diagram (the kind used in class notes and homework).

(*c*) Again consider a program that frequently accesses global memory and so has alot of latency to hide. Explain how a configuration with fewer warps but more registers per warp can also hide latency using those bountiful registers. (The matrix multiply machine code does this). Illustrate your answer with a timing diagram, showing two examples, both using the same number of warps, but one using more registers to hide latency. In those two examples the number of loads per thread will be the same (because they are running the same program and have the same number of threads).

(*d*) Considering the last two parts, try to argue that to some limit (say 8) fewer warps are better. Try comparing specific cases such as 8 warps with 63 registers each to 16 warps with 32 registers each. Argue that latency hiding is roughly equivalent but that there is some advantage to having fewer warps. Illustrate your answer with timing diagrams.