Name _____

GPU Microarchitecture
EE 7722
Take-Home Final Examination

Wednesday, 6 May 2015 to Saturday, 9 May 2015

Work on this exam alone. Regular class resources, such as notes, past homework and exam solutions, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else except Dr. Koppelman. Collaboration with or copying from others on exams is considered a serious violation of the Student Code of Conduct, and those involved will be reported to the Dean of Students.

Problem 1	_____	(25 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing below is code based on the solution to pre-final Problem 3, which is based on Homework 1.

In this problem analyze the execution of the code below on an Kepler-like system with the following characteristics. Global memory loads have a latency of 200 cycles and all other instructions have a latency of 12 cycles. Instructions issue at a rate of one warp per cycle.

In your analyses consider only instructions performing floating-point calculations and those used to load or store from global and shared memory. Ignore instructions used to compute indices, memory addresses and so on.

Base the analysis on the high level code, **NOT** on the assembly language. Assume that the compiler will do a reasonable job scheduling (rearranging) instructions to hide latency. *Hint: This was ignored by many in the pre-final.*

```
extern "C" __global__ void mxv_sh_ochunk() {
  const int CS = 8;  // Chunk Size: Number of input vector elts to read.
  const int num_threads = blockDim.x * gridDim.x;
  const int bl_start = blockIdx.x * blockDim.x / CS;
  const int stop = d_app.num_vecs;
  const int inc = num_threads / CS;
  const int thd_c_offset = threadIdx.x % CS;
  const int thd_r_offset = threadIdx.x % CS;
  const int thd_v_offset = threadIdx.x / CS;

  const int MAX_BLOCK_SIZE = 1024;
  __shared__ Elt_Type vxfer[MAX_BLOCK_SIZE];

  // Number of output vector components assigned to each thread.
  const int ML = M / CS;

  for ( int hb = bl_start; hb<stop; hb += inc ) {
      // Vector number assigned to this thread and its CS-1 partners.
      const int vec_num = hb + thd_v_offset;

      // Initialize storage for output vector components.
      Elt_Type vout[ML];
      for ( int rl=0; rl<ML; rl++ ) vout[rl] = 0;

      /// Compute Output Vector
      // Each iteration of the c loop uses CS input vector components
      // to update ML output vector components (per thread).
      //
      for ( int c=0; c<N; c += CS ) {
          // Read in 1 component of input vector and place in shared memory.
          vxfer[threadIdx.x] = d_app.d_in[ vec_num * N + c + thd_c_offset ];

          // Transfer the CS (8) components just read to local memory.
          Elt_Type vin[CS];
          for ( int cc=0; cc<CS; cc++ )
            vin[cc] = vxfer[ thd_v_offset * CS + cc ];

          // Using the CS components just read, update our ML output
          // vector components.
          for ( int rr=0; rr<ML; rr++ )
```

```
          for ( int cc=0; cc<CS; cc++ )
            vout[rr] += d_app.matrix[ rr*CS + thd_r_offset ][c+cc] * vin[cc]; }

    /// Write Output Vector
    for ( int rr=0; rr<ML; rr++ )
      d_app.d_out[ vec_num * M + rr * CS + thd_r_offset ] = vout[rr];
  }}
```

(*a*) Estimate the latency of one iteration of the `hb` loop. Show the latency in terms of N, M, and C (in mathematical expressions use C instead of `CS`).

Assume that there are an unlimited number of registers and so decreasing or increasing `CS` will not cause a radical change in code due to register exhaustion. Also assume an unlimited amount of off-chip bandwidth.

Assume that the compiler will schedule (re-arrange) instructions to minimize latency, but will not change the order of operations. (In the next problem re-arrangement will be considered.) When considering scheduling look especially at the `rr/cc` loop nest.

(*b*) Based on the analysis in the previous part estimate the number of instructions in one iteration of the `hb` loop. Show the number in terms of N, M, and C (in mathematical expressions use C instead of `CS`).

(*c*) Using the answers from the previous two parts, compute the number of threads needed to saturate the following system.

There are four warp schedulers, and each can issue one instruction from one warp per cycle. There are 128 functional units of every type.

(*d*) Based on the answer to the previous parts, which value of `CS` will give the best throughput, measured in vectors per cycle?

Problem 2: [20 pts] Continue to consider the code and system from the previous problem. There when analyzing latency we were to assume that the compiler could not change the order of operations. That is, if the program specified `x = a + b + c` the compiler would need to perform the additions in this order: `add r1, a, b; add x, r1, c`. It *could not* do additions in this order: `add r1, b, c; add x, r1, a`. The two orders are algebraically equivalent but can produce different results due to floating point rounding.

In this problem we *will* allow the compiler to re-arrange the order of operations.

(*a*) Recompute the latency of an iteration of the `hb` loop assuming that the compiler can re-arrange operations as described above. In particular, focus on the loop nest below.

```
for ( int rr=0; rr<ML; rr++ )
  for ( int cc=0; cc<CS; cc++ )
    vout[rr] += d_app.matrix[ rr*CS + thd_r_offset ][c+cc] * vin[cc]; }
```

(*b*) Compute the number of threads needed when the compiler can reorder operations. Do this for the same system as used in the previous problem.

(*c*) Here's the interesting part. Your answer to the last question should show that fewer threads are needed to saturate instruction issue when the compiler can re-arrange operations. How do you think that the number of registers per thread will change compared to the analysis in the last problem? The change in the number of registers will be due to how the calculation in the loop nest was changed.

Problem 3: [10 pts] The Homework 1 code used shared memory to pass vertex components to threads in a chunk. A shortened version of the code appears below.

The number of shared memory banks on a Kepler device is 32. Suppose we could change the number of banks in a design (perhaps for a future GPU design). What is the minimum number of banks needed to avoid bank conflict in the code below? Your answer should be in terms of C (`CS`). Explain your answer. A correct answer without an explanation will only get partial credit.

```
extern "C" __global__ void shared_conflicts() {
  const int CS = 8;  // Chunk Size: Number of input vector elts to read.
  const int num_threads = blockDim.x * gridDim.x;
  const int bl_start = blockIdx.x * blockDim.x / CS;
  const int stop = d_app.num_vecs;
  const int inc = num_threads / CS;

  const int thd_c_offset = threadIdx.x % CS;
  const int thd_v_offset = threadIdx.x / CS;

  __shared__ Elt_Type vxfer[1024];

  for ( int hb = bl_start; hb<stop; hb += inc ) {
      const int vec_num = hb + thd_v_offset;

      for ( int c=0; c<N; c += CS ) {
          vxfer[threadIdx.x] = d_app.d_in[ vec_num * N + c + thd_c_offset ];

          Elt_Type vin[CS];
          for ( int cc=0; cc<CS; cc++ )
            vin[cc] = vxfer[ thd_v_offset * CS + cc ];
```

Problem 4: [15 pts] Phi arithmetic instructions can optionally swizzle the lanes in one of its source operands, this is performed by the `VC1` and `VC2` stages of the pipeline. In the example below the values in certain lanes of `zmm2` are swapped the result is added to `zmm1` and stored in `zmm3`.

```
vaddpd    %zmm3 = %zmm2{badc}, %zmm1
```

(*a*) There is no single Kepler instruction that can do the same thing. The CUDA kernel below performs the equivalent of the Phi instruction above without the swizzle. (The CUDA code also loads and stores operands, ignore that for now.) Modify the CUDA kernel below to perform the same operation as the Phi instruction above. Note that variable names have been chosen to match the Phi register names above.

```
extern "C" __global__ void phi_equiv() {
  const int idx = threadIdx.x + blockIdx.x * blockDim.x;
  double zmm1 = a[idx];
  double zmm2 = b[idx];



  double zmm3 = zmm1 + zmm2;



  c[idx] = zmm3;
}
```

(*b*) The Phi `VC1` and `VC2` stages are not just used for register swizzling. What is it that they do that might come in handy in the code below. The code examples below do not require swizzling. Show how the Phi code can be modified to take advantage of those stages.

```
// CUDA Code.
//
extern "C" __global__ void simple_add() {
  const int idx = threadIdx.x + blockIdx.x * blockDim.x;
  double zmm1 = a[idx];
  double zmm2 = b[idx];
  double zmm3 = zmm1 + zmm2;
  c[idx] = zmm3;
}

// Roughly Equivalent Phi Assembly Code Fragment

        vmovapd   %zmm1 = 0(%r8,%rax,8)   # zmm1 = Mem[ r8 + rax * 8]
        vmovapd   %zmm2 = 0(%r9,%rax,8)   # zmm2 = Mem[ r9 + rax * 8]

        vaddpd    %zmm3 = %zmm2, %zmm1

        vmovapd   0(%r10,%rax,8), %zmm3
```

(*c*) The Phi code above would need to be embedded in a loop. Which register would have to be incremented for the next iteration, and by how much?

Problem 5: [15 pts] Consider again CUDA and Phi code samples:

```
// CUDA Code.
//
extern "C" __global__ void simple_add() {
  const int idx = threadIdx.x + blockIdx.x * blockDim.x;
  double zmm1 = a[idx];
  double zmm2 = b[idx];
  double zmm3 = zmm1 + zmm2;
  c[idx] = zmm3;
}

// Roughly Equivalent Phi Assembly Code Fragment

        vmovapd   %zmm1 = 0(%r8,%rax,8)   # zmm1 = Mem[ r8 + rax * 8]
        vmovapd   %zmm2 = 0(%r9,%rax,8)   # zmm2 = Mem[ r9 + rax * 8]

        vaddpd    %zmm3 = %zmm2, %zmm1

        vmovapd   0(%r10,%rax,8), %zmm3
```

(*a*) The Phi code above assumes that `r8`, `r9`, and `r10` are aligned addresses. The CUDA code does not exactly have an equivalent concept, and the code will run fine as long as the address is a multiple of 8 bytes (which is the element size, not the warp size). Nevertheless the CUDA code will suffer a performance penalty if the array address used by, say, thread 0 is not a multiple of something. What is that performance penalty, be as precise as possible.

(*b*) How would the Phi routine be modified if we could not guarantee that the addresses would be aligned?

(*c*) Appearing is the Kepler assembly code for the CUDA code above. Which instructions perform duplicate work that would not be duplicated on the Phi.

```
/*0008*/                   MOV R1, c[0x0][0x44];
/*0010*/                   S2R R3, SR_TID.X;
/*0018*/                   MOV32I R7, 0x8;
/*0020*/                   S2R R0, SR_CTAID.X;
/*0028*/                   IMAD R0, R0, c[0x0][0x28], R3;
/*0030*/                   IMAD R8.CC, R0, R7, c[0x0][0x140];
/*0038*/                   IMAD.HI.X R9, R0, R7, c[0x0][0x144];
/*0048*/                   IMAD R10.CC, R0, R7, c[0x0][0x148];
/*0050*/                   IMAD.HI.X R11, R0, R7, c[0x0][0x14c];
/*0058*/                   LD.E.64 R4, [R8];
/*0060*/                   LD.E.64 R2, [R10];
/*0068*/                   IMAD R6.CC, R0, R7, c[0x0][0x150];
/*0070*/                   IMAD.HI.X R7, R0, R7, c[0x0][0x154];
/*0078*/                   DADD R2, R4, R2;
/*0088*/                   ST.E.64 [R6], R2;
/*0090*/                   EXIT;
```

Problem 6: [15 pts] Based on reverse-engineering we discovered that Kepler load instructions can have a 32-bit signed offset. Thirty-two bits! The CUDA code below can almost take advantage of these large offsets in two ways.

```
__constant__ int size;

void calling_routine() {
  const int stride = 48611;
  const int num_blocks = 13;
  const int thd_per_block = 1024;
  const int gap = 16 * stride;
  const int size_h = thd_per_block * num_blocks * 12345;

  cudaMemcpyToSymbol
      ( size, &size_h, sizeof(size_h), 0, cudaMemcpyHostToDevice );

  double *a_d, *c_d;
  const size_t size_bytes = ( 2 * size_h + gap ) * sizeof(a_d[0]);
  cudaMalloc( &a_d,  size_bytes * 2 );
  c_d = &a_d[size_h+gap];
  init_array(a_d);

  loops<<<num_blocks,thd_per_block>>>(a_d, c_d, stride);
}

extern "C" __global__ void loops(double *a, double *c, int stride) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;

  for ( int idx = tid; idx < size; idx += num_threads )
    {
      double x = 0;
      for ( int j=0; j<16; j++ ) x += a[ idx + j * stride ];

      c[idx] = x;
    }
}
```

(*a*) Identify and fix the problems. *Hint: One concerns code in the* `j` *loop and one concerns code outside the* `j` *loop.*

(*b*) A correct answer to the question above should have reduced instruction count due to the use of offsets. However, overall performance would not have improved much on current GPUs. Explain why. Is there a way to re-write the code to avoid the problem? If yes, provide some ideas on how to do it.