Name _____

GPU Microarchitecture
EE 7722
Take-Home Mid-Term Examination

Friday, 15 April 2016 to Wednesday, 20 April 2016

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, except questions or concerns about problems should be directed to Dr. Koppelman.

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

Alias _____

Good Luck!

Warning: This problem is harder than the rest. Read it, think about it a little, but solve the other problems and then come back to this first one.

Problem 1: [25 pts] Read the Echelon/exascale paper, Villa 14, "Scaling the Power Wall," and to help clarify certain concepts, look at an earlier description of Echelon in Keckler 11, "GPUs and the Future of Parallel Computing." Both are linked to the course references page at
`http://www.ece.lsu.edu/gp/gpu-descriptions.html` and the papers can be accessed free on campus.

The paper describes a technique called temporal SIMT (TSIMT). Two benefits of TSIMT are described, reduced branch divergence penalty and scalarization.

(*a*) Show an if/else example in which the TSIMT version has better performance than its execution on something using warps in a way more like a CC 3.5 devices. The two systems must be similar enough to be comparable. Note that the reduced divergence penalty considered in this problem is **not** due to the smaller warp size, it's due to TSIMT.

(*b*) Arguably, the divergence penalty can be reduced to zero with TSIMT. What's the catch? In other words, suppose a region of code diverges to the point that warps have only one active thread. Why might execution slow down anyway? *Hint: It has to do with the primary design limiter.* Explain your answer, don't just state the cause.

Problem 1, continued:

(*c*) The other benefit of TSIMT was scalarization. In the code below circle or otherwise identify parts that might benefit from scalarization and explain why.

```
__global__ void sort_segments_1_bit_split() {
  int elt_per_thread = 4;
  int elt_per_block = elt_per_thread * blockDim.x;
  int idx_block_start = elt_per_block * blockIdx.x;

  int idx_start = idx_block_start + threadIdx.x;

  for ( int i = 0;  i < elt_per_thread; i++ ) {
      const int sidx = threadIdx.x + i * blockDim.x;
      s[ sidx ] = sort_in[ idx_block_start + sidx ];
    }}
```

(*d*) The paper describes scalarization of instructions within a warp. But it would also be possible to scalarize instructions in a block.

☐ Is the code above a candidate for block-wide scalarization?

☐ if the fragment above is a candidate for block-wide scalarization then provide an example of code that is only a candidate for warp-wide scalarization. Otherwise, provide an example of code that isn't a candidate for warp-wide scalaraization but is a candidate for block-wide scalaraization.

(*e*) Explain why it would be much harder to exploit block-wide scalarization than warp-wide scalarization.

Problem 2: [20 pts] Answer the following questions about the structure below.

```
struct My_Struct {
 float a;
 char c;
 int b;
 double d;
};

My_Struct *my_structs;
```

(*a*) Consider the statement below, which refers to the structure above.

Accessing the array `my_structs` *in the usual way in CUDA, having each thread operate on component* `a` *of a different element of* `my_structs` *will be inefficient because of request underutilization. The problem can be avoided by having, say, thread 0 do something with component* `a`*, thread 1 do something with component* `c`*, thread 2 and 3 work with* `b` *and* `d`*, thread 4 with* `a` *of the next element of* `my_structs`*, and so on.*

☐ Explain why the technique for avoiding the problem is much harder to implement than it sounds.

Problem 2, continued:

(*b*) Consider the two routines below operating on `My_Struct`.

```
__global__ void ms_as_needed(My_Struct *din) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;

  something(din[tid].a);
  something_else(din[tid].b);
  something_other_thing(din[tid].c);
  something_final_thing(din[tid].d);
}

__global__ void ms_all_at_once(My_Struct *din) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;

  My_Struct ms = din[tid];

  something(ms.a);
  something_else(ms.b);
  something_other_thing(ms.c);
  something_final_thing(ms.d);
}
```

☐ Explain why the performance of the two routines will be the same on a Kepler-generation GPU.

☐ Suppose that the size of the array were n elements. Ignoring the L2 cache, how much data will be moved from device memory to the GPU when accessing the array. (The answer is not, of course, $n\times$`sizeof(My_Struct)`.)

(*c*) Suppose the two routines were run on a CPU (after removing the `__global__` and doing something about `threadIdx` and `blockIdx`). One of the routines will run faster than the other. Suppose that the `something` routines access memory.

☐ Which one is faster and why is it faster?

Problem 3: [20 pts] Since `My_Struct` (from the previous problem) is not too large one solution is to buffer it in shared memory.

(*a*) Modify the code below so that the structure elements are first copied to shared memory, then each thread operates on its own element. *Hint: Look at the type punning used in the solution to Homework 2 Problem 2 in which a 2D array was punned to a 1-D array.*

```
struct My_Struct {
 float a;
 char c[4];
 int b;
 double d;
};

__global__ void ms_bufered(My_Struct *din) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;

  My_Struct ms = din[tid];

  something(ms.a);
  something_else(ms.b);
  something_other_thing(ms.c);
  something_final_thing(ms.d);
}
```

(*b*) Suppose we solved the previous part correctly. Describe the difference in performance of the code above with `My_Struct` as it is above, with the performance when `double d` in `My_Struct` is changed to `float d`. Only consider the performance of the code *after* shared memory is initialized. Be as specific as possible about how much longer it will take.

Problem 4: [10 pts] Answer each question below.

(*a*) Suppose we were designing an NVIDIA-like GPU with the following specifications: Two schedulers, one dispatch unit for each scheduler and a total of 32 functional units split between the schedulers. There is only one kind of functional unit and all instructions use them. Operation latency is 20 cycles, even for loads. Assume that on average there is one instruction between each pair of dependent instructions.

Choose the hardware limit on the number of resident warps. (If the answer is 1 million then we'll build a GPU having SMs that can have up to 1 million resident warps each.)

☐ The number of warps an MP in our design should hold is:

☐ Explain.

(*b*) A Fermi (CC 2.x) GPU has a 48 kiB L1 data cache that backs the global address space (Kepler and Maxwell lack these). The size of a typical CPU L1 data cache is 64 kiB. Why would it be very misleading to say that the Fermi cache is almost as large this CPU cache? Provide a quick code sample showing the difference.

Problem 5: [25 pts] The code below uses a small array in several different ways. When the kernel starts `some_array` is properly initialized with the array, and the pointer `sa_global` is initialized to a pointer to global memory which is pre-loaded with the same data. So `some_array[x]` and `sa_global[x]` will return the same data, though not in the same amount of time.

(*a*) The routine below contains two fragments, Version A and Version B. Version B will contain twice as many instructions. Explain why. *Hint: The extra instructions are not add instructions.*

```
const int SIZE = 128;
__constant__ float some_array[SIZE];
__constant__ float *sa_global;  // Set to pointer to global space.

__global__ void short_answer(float *din, float *dout, int* offset) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  float x = 0;

  // Version A
  for ( int i=0; i<SIZE; i++ ) x += some_array[i] * din[tid+i];
  dout[tid] = x;

  // Version B
  const int off = offset[tid];
  for ( int i=0; i<SIZE; i++ ) x += some_array[i+off] * din[tid+i];
  dout[tid] = x;
}
```

(*b*) Version A below uses constant-space storage and Version C copied the array from global to local storage. Consider the execution on a CC 3.5 GPU.

```
const int SIZE = 128;
__constant__ float some_array[SIZE];
__constant__ float *sa_global;  // Set to pointer to global space.

__global__ void short_answer(float *din, float *dout, int* offset) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  float x = 0;

  // Version A
  for ( int i=0; i<SIZE; i++ ) x += some_array[i] * din[tid+i];
  dout[tid] = x;

  // Version C
  float sa_local[SIZE];
  for ( int i=0; i<SIZE; i++ ) sa_local[i] = sa_global[i];
  for ( int i=0; i<SIZE; i++ ) x += sa_local[i] * din[tid+i];
  dout[tid] = x;
}
```

☐ For Version A, what is the maximum size of `SIZE`?

☐ For Version A, at what value of `SIZE` will performance start to suffer?

☐ Explain.

☐ For Version C, what is the maximum size of `SIZE`?

☐ For Version C, at what value of `SIZE` will performance start to suffer?

☐ Explain.