Name Solution

EE 7700-1

Take-Home Pre-Final Examination

Wednesday, 29 April 2009 to Early Monday Morning, 4 April 2009

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

Alias Warped

Good Luck!

Problem 1: [20 pts]The CUDA code below performs some calculation on each element of an array, `global data`. It operates on a GeForce 8000-like GPU with:

- Eight multiprocessors.
- Core clock frequency of 1.5 GHz.
- Four-hundred cycle global access latency.

The `cuda normal thread` routine consists of 9 instructions including one global load.

```
__host__ void launch(int element_amt_lg)
{
  const int element_amt = 1 << element_amt_lg;
  dim3 block_dim(256,1,1);
  dim3 grid_dim( 1 << element_amt_lg - 8, 1, 1 );
  cuda_normal_thread<<<grid_dim,block_dim>>>(element_amt);
}

__global__ void
cuda_normal_thread(int element_amt)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const float coord = global_data[idx];
  output_data[idx] = some_func(transform_matrix,coord);
}
```

(*a*) Determine either the time needed for 2^{20} elements (`work amt lg` is 20), or else determine the computation rate in elements per cycle. (Two ways of expressing the same thing.)

As stated, a kernel consists of 9 instructions and so there are a total of 9×2^{20} kernel instructions in the execution, or $9 \times \frac{2^{20}}{8}$ instructions per multiprocessors (MP). Each multiprocessor (MP) can execute up to 8 instructions per cycle. If the MPs execute at their peak execution will require $\frac{1}{8}9 \times \frac{2^{20}}{8} = 9 \times 2^{14}$ cycles or $9 \times 2^{14} \frac{1}{1.5 \times 10^9}$ seconds (or $98.3\,\mu$s).

A necessary condition for an MP to execute at its peak rate is that the number of cycles between instructions in a thread is long enough to cover operation latency. Most non-memory instructions have a 24-cycle latency and so can be covered with 192 threads or 6 warps in the worst case where there is no separation between dependent instructions (such as the last two instructions in the code below). To cover the memory latency requires 100 warps or 3200 threads in the worst case.

The number of warps per MP is $2^{12} = 4096$ which would be enough to cover memory latency, however an MP has a maximum of 24 active warps. *Grading note: full credit would be given if the 24-active warp limits was ignored or brushed off.* In fact, 18 warps are enough for the code below because the memory dependence is only between two instructions, other dependencies have much lower latency. Call instructions 0-6 part A and instructions 7 and 8 part B. At what we'll call step i the scheduler would run part A instructions for threads $[192i, 192(i+1)-1]$ and then run part B instructions for threads $[192(i-2), 192(i-1)-1]$. Assuming no stalls the total time for a part is $9 \times \frac{192}{8} = 216$ cycles. Consider the parts $A_x, B_{x-2}, A_{x+1}, B_{x-1}, A_{x+2}, B_x, \ldots$. Each A_i takes 168 cycles and each B_i takes 48 cycles so that the time between between A_x and B_x is $2 \times (168 + 48) = 432$ cycles, enough to cover memory latency. Each pair A_i and B_i contains 6 warps (192) threads. At most a multiprocessor needs to keep three sets of 6 warps active, in the example above the three sets needed were AB_{x-2}, AB_{x-1}, and AB_x, so the number of warps needed is 18.

So the execution time is the time given in the first paragraph. The rate for ideal execution is the number of instructions the GPU can execute per cycle divided by the number of instructions per element: $\frac{64}{9} = 7.11$.

(*b*) The code below may be the true assembly language for `cuda normal thread`. (See `http://www.cs.rug.nl/ wladimier/de`
Based on this code, what would be the minimum number of threads per multiprocessor needed to achieve peak performance? Explain.

```
0: mov.b16 $r0.hi, %ntid.y
1: cvt.u32.u16 $r1, $r0.lo
2: mad24.lo.u32.u16.u16.u32 $r0, s[0x000c], $r0.hi, $r1
3: shl.u32 $r1, $r0, 0x00000002
4: add.u32 $r0, $r1, c0[0x0040]
5: mov.u32 $r0, g[$r0]              // <- Global load of data into r0.
6: add.u32 $r1, $r1, c0[0x0044]
7: add.rn.f32 $r0, $r0, 0x3f800000
8: mov.end.u32 g[$r1], $r0
```

Based on the solution to the previous part, 576 threads (18 warps).

If one assumed the threads had to be scheduled in strict round-robin fashion then to cover the 400 cycles of latency spanning one instruction would require $\frac{400}{4\times 2} = 50$ warps or 1600 threads.

Problem 2: [20 pts]The CUDA code below does exactly the same thing as the code from the previous problem, but it does so in an un-cuda-like fashion: looping over elements in a thread rather than spawning one thread per element. The code runs on the system described in the previous problem.

```
__host__ void launch_loop(int element_amt_lg, int thds_per_mp)
{
  const int element_amt = 1 << element_amt_lg;
  const int num_mp = 8; // Number of multiprocessors.
  dim3 block_dim(thds_per_mp,1,1);
  dim3 grid_dim(num_mp,1,1);
  const int iter_per_thread =
    ceil( double(element_amt) / ( thds_per_mp * num_mp ));
  cuda_loop_thread<<<grid_dim,block_dim>>>(iter_per_thread,element_amt);
}

__global__ void
cuda_loop_thread(int iter_per_thread, int element_amt)
{
  const int idx_start =
    ( blockIdx.x * blockDim.x + threadIdx.x ) * iter_per_thread;
  const int idx_stop = idx_start + iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx++ ) {
      const float coord = global_data[idx];    // <-   IGNORE COALESCE PROBLEMS
      output_data[idx] = some_func(transform_matrix,coord);  }
}
```

Here is the assembler for the loop body:

```
label0: mov.u32 $r3, g[$r0]       <- Global load.
add.rn.f32 $r3, $r3, 0x3f800000
add.b32 $r0, $r0, 0x00000004
mov.u32 g[$r1], $r3
set.ne.u32 $p0$o127, $r0, $r2
add.b32 $r1, $r1, 0x00000004
@$p0.ne bra.label label0
```

(a) Determine the time needed to process 2^{20} elements (or the computation rate) when `thds per mp` is 1.

Each global load will cost 400 cycles because the dependent instruction immediately follows it. Just taking memory into account the code would need at least 400 cycles per element. The computation rate is $\frac{8}{400}$ elements per cycle (since there are 8 MPs), the total time would be at least $2^{17} \times 400$ cycles.

(b) Determine the time needed to process 2^{20} elements (or the computation rate) when `thds per mp` is 32.

With 32 threads per MP, there will be 32 global accesses done in parallel. The rate would now be $\frac{8\times32}{400}$ since 8×32 loads are being performed in parallel. The total time would be $\frac{2^{17}}{32} \times 400$ cycles, again taking only memory into account.

(c) Do you think the loop approach in this problem is better or worse than the normal CUDA approach from the previous problem? Explain.

One advantage of the loop approach is that there is less arithmetic needed per element. (The loop body has 7 instructions, fewer than the 9 instructions used by the cuda-like kernel.) One big disadvantage is that within a thread multiple global accesses cannot be overlapped because the data must arrive before the loop proceeds to the next iteration. In contrast when there are many threads each performing one global access one thread waiting for global memory does not prevent threads in other warps from proceeding. This way they can overlap their global accesses, enabling maximum efficiency execution.

Problem 3: [20 pts]On a system like the GeForce 8000 memory latency can be hidden with lots of threads but that won't work on Larrabee where there are many fewer threads. For comparison purposes treat each Larrabbee thread as 16 GeForce threads, so each Larrabbee core has a total of 64 threads. The Larrabbee equivalent of a warp would be 16 threads.

Since there are too few threads to hide memory latency Larrabbee must rely on prefetching. The code below is the kernel from the previous problem, but with some kind of a prefetch call added. It is to run on a Larrabbee-like implementation and so global accesses can benefit from a cache hit, but for the code below there will only be a cache hit if the prefetch is done correctly.

Note: This paragraph did not appear in the original exam. Assume that the compiler unrolls the loop 16 times assigning each of the 16 loop body copies to a different lane. After unrolling the number of iterations will be `iter\_per\_thread / 16`. The `threadIdx.x` variable below refers to a Larrabbee thread.

(*a*) Finish the code so that `pre idx` is set to an appropriate value for the prefetch. Assume a 400-cycle memory latency. As before ignore coalesce issues. The prefetched data should arrive when and where it is needed, and should not arrive too early (wastes cache space) or too late (results in stall).

```
__global__ void
l_loop_thread(int iter_per_thread, int element_amt)
{
  const int idx_start =
    ( blockIdx.x * blockDim.x + threadIdx.x ) * iter_per_thread;
  const int idx_stop = idx_start + iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx++ )
    {
      const float coord = global_data[idx];

      const int pre_idx =  idx + 800;                 //  <-- SOLUTION


      prefetch(&global_data[pre_idx]);
      output_data[idx] = some_func(transform_matrix,coord);
    }
}
```

Assume that the code consists of 8 vector instructions plus a few IA-32 instructions, and that a loop iteration executes in 8 cycles. The prefetch must be issued 400 cycles in advance. Since there are 16 lanes each executing an iteration (before unrolling) the code is proceeding at a rate of $\frac{8}{16}$ cycles per iteration. In the time needed to fetch global data the code can advance over $400\frac{16}{8}$ elements and so prefetch must be 800 elements ahead.

(*b*) Once the problem above is understood it should not have been too hard to determine a prefetch address (or array index). The general case of inserting prefetch instructions in ordinary CPU code is much harder, often prefetches wastefully bring in data that is never used.

A Larrabbee critic might say that since prefetching is hard Larrabbee will be slowed down by cache misses (because the prefetch was wrong) and so the GeForce 8000 approach of many threads to hide latency is better. Explain the fallacy in this argument. *Hint: Consider how addresses are computed in GeForce (cuda) threads.*

The GeForce threads must compute global addresses using data that was available at the time of the kernel launch (constants and global memory) and from thread and block indices. With those restrictions it would be no problem generating prefetch addresses, just add some amount to what would be thread and block indices.

Constructing a prefetch address is hard when the load address depends upon how earlier code executes (for example, the direction in which past branches were taken). That hard case won't occur in GPU code because a thread ordinarily can't depend on prior threads (or the branches in them), and when they do (through shared or global memory, they loose some benefit of latency hiding.

Problem 4: [20 pts]The familiar CUDA code below will probably be slowed down because of inefficient global memory access. Re-write the code to fix the problem. The code runs on a GeForce 8000-like system. (Refer to the CUDA Programming guide for help.)

```
__global__ void
cuda_loop_thread(int iter_per_thread, int element_amt)
{
  const int idx_start =
    ( blockIdx.x * blockDim.x + threadIdx.x ) * iter_per_thread;
  const int idx_stop = idx_start + iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx++ )
    {
      const float coord = global_data[idx];
      output_data[idx] = some_func(transform_matrix,coord);
    }
}

__global__ void
cuda_loop_thread_sol(int iter_per_thread, int element_amt)
{
  const int idx_start =
    blockIdx.x * blockDim.x * iter_per_thread + threadIdx.x;
  const int idx_stop = idx_start + blockDim.x * iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx += blockDim.x )
    {
      const float coord = global_data[idx];
      output_data[idx] = some_func(transform_matrix,coord);
    }
}
```

For global access to be most efficient consecutive threads must access consecutive elements covering a contiguous block of memory. That is, if thread 0 is accessing element 1024 thread 1 must access element 1025. In the (pre-solution) code above if thread 0 accessed element 1024 thread 1 would be accessing 1024 + `iter per thread`. Since these elements are not consecutive memory bandwidth would be wasted because a separate memory request would be made for each load, rather than coalescing the two loads into one request. The solution is to re-organize the order in which threads access elements to that the expression for `idx` is something plus `threadIdx.x`, so that consecutive threads access consecutive elements. Note that the loop iterator is incremented by block size instead of 1.

Problem 5: [20 pts]The code below sums values provided by all threads of a block. The `block lg` argument is the log base 2 of the block size, `shared array` is a pointer to shared memory, `my value` is the thread's value to sum, and `all` determines whether all callers get the sum, or just thread 0. (This is a simplified version of the code in https://svn.ece.lsu.edu/svn/gp/gpgpu/balloon-kernel.cu.)

```
__device__ float
reduce(int block_lg, float *shared_array, float my_value, bool all)
{
  const int tid = threadIdx.x;
  const int block_lg_h = block_lg >> 1;
  const int block_lg_l = block_lg - block_lg_h;
  const int upper_size = 1 << block_lg_h;
  const int lower_size = 1 << block_lg_l;

  float vol_sum = shared_array[tid] = my_value;
  __syncthreads();

  // Round 1
  //
  if ( tid < lower_size )
    {
      for ( int i=1; i<upper_size; i++ )
        vol_sum += shared_array[ (i << block_lg_l ) + tid ];
      shared_array[tid] = vol_sum;
    }

  // Round 2
  //
  if ( lower_size > warpSize ) __syncthreads();

  if ( tid == 0 )
    for ( int i=1; i<lower_size; i++ ) vol_sum += shared_array[i];

  if ( !all ) return vol_sum;
  if ( tid == 0 ) shared_array[0] = vol_sum;
  __syncthreads();
  return shared_array[0];
}
```

Questions on next page.

Problem 5, continued:

(*a*) How much divergence is caused by the `if` statement right after the "Round 2" comment. (Easy)

☑ How much? Explain.

It causes no divergence at all because the branch is a function of things that are the same for all threads (warp size and block log) and so in every thread the branch is taken the same way. There is no loss of efficiency when all threads in a warp take the same path through a branch.

(*b*) Explain why the `syncthreads` call just after the "Round 2" comment is definitely needed when `lower size` is greater than the warp size.

☑ Why needed?

The code in round 1 updates shared memory that will be read by thread 0 in round 2. Without `syncthreads` thread 0 might read the shared array before other threads wrote it.

☑ Show an example in which execution would be incorrect without it.

```
               Time ---->             DIAGRAM FOR SOLUTION
Thread Number  Activity
-------------  --------------------
Threads  0-31  R1 i=1,  R1 i=2,          R1 sa[]=          R2 v +=sa[]
Threads 32-63                     R1 i=1            R1 i=2               R1 sa[]=
```

The diagram above shows thread activity versus time for 64 threads spanning two warps. Notice that execution switches between the two warps but that the first warp is ahead and reaches Round 2 before warp 1 (threads 32-63) writes the shared array.

(*c*) Explain why the `syncthreads` call might not be needed when `lower size` is not larger than the warp size.

☑ Why not needed?

If `lower_size` is not larger than the warp size then all of the threads that write the shared array will be in the same warp as thread 0 (which also writes it). If they are in the same warp then they execute together and so thread 0 can't reach Round 2 before the others.

☑ Explain why execution is correct with example from previous part.

With syncthreads the first warp could not enter round 2 (past syncthreads) until all warps reached syncthreads. That would avoid the read-before-write problem.