Name Solution

EE 7700-1

Take-Home Pre-Final Examination

Monday, 3 May 2010 to Early Morning Friday, 7 May 2010

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

Problem 1	_____	(16 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(8 pts)
Problem 6	_____	(12 pts)
Problem 7	_____	(10 pts)
Problem 8	_____	(12 pts)
Exam Total	_____	(100 pts)

Alias Fermiger

Good Luck!

Problem 1: [16 pts] The cuda kernel below performs a binary search on a table and uses the value found to perform a calculation. At the beginning of the kernel the table is copied to shared memory and the binary search operates on that copy.

```
__constant__ float *stable;   // Some sorted table.
__constant__ float *targets;
__constant__ float *results;
__constant__ int block_size_lg;
const int list_size_lg = 11;

__global__ void bsearch()
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const int tsize = 1 << list_size_lg;
  const int my_start = threadIdx.x << ( list_size_lg - block_size_lg );
  const int my_share = 1 << ( list_size_lg - block_size_lg );
  const int my_stop = my_start + my_share;

  // Copy stable to shared memory.
  __shared__ float stable_copy[tsize];
  for ( int i=my_start; i<my_stop; i++ ) stable_copy[i] = stable[i];

  __syncthreads();

  int ti = 0;      // Table indexed, used in binary search.
  const float target = targets[idx];

  // Perform binary search to find  closest element to target.
  for ( int tree_level = list_size_lg - 1; tree_level >= 0; tree_level-- )
    {
      const int ti_maybe = ti | ( 1 << tree_level );
      const float elt = stable_copy[ti_maybe];
      if ( target < elt ) continue;
      ti = ti_maybe;
    }

  // A calculation using the value that was found.
  const float diff = stable_copy[ti] - target;
  results[idx] = diff * diff;
}
```

Continued on next page.

(*a*) Suppose the kernel is launched in a grid of just one block, and the block size is 256, the value of `list_size_lg` is 11, and the kernel is run on a GPU of compute capability 1.3 as described by the CUDA Programming Guide 3.0. What is the total amount of requested global memory data per thread?

The thread accesses global memory to retrieve a value of `targets` and several values of `stable`.

The index to the array `targets` is of the form `C + threadIdx.x`, where `C` is the same for all threads in a block. (That, of course, means that `C` is the same for all threads in a half-warp, which is what's important here.) Because the address is in this form consecutive threads will accesses consecutive elements and so the accesses will be coherent. The size of an element of `targets` is four bytes. Memory requests are prepared for half-warps, and the half warps here need $16 \times 4 = 64$ bytes of data which can be satisfied by a single request. In summary, the access to `targets` needs four bytes per thread and fortunately results in requests that bring in four bytes per thread (via a single 64-byte request serving 16 threads).

The index to the array `stable` is of the form `C + i + 8 * threadIdx.x` , where `i` is the loop iterator. (The stride, 8, is $2^{\texttt{listsizelg}-\texttt{blocksizelg}}$.) Rather than accessing consecutive elements of `stable`, threads access elements at a stride of 8 (meaning every eighth element). Since the element size is 4 consecutive threads will access elements that differ in address by 32 bytes. The minimum request size is 32 and the maximum is 128. A request of size 32 would bring data for one thread (with 28 of the 32 bytes, $\frac{7}{8}$ going unused), a request of size 64 would bring in data for two threads, but that would still leave $\frac{7}{8}$ unused. Since the amount of waste is not a function of size, it would make sense for the load unit to issue 128-byte requests, satisfying 4 threads. It would take four such requests to satisfy a half warp. In summary the access to `stable` needs $8 \times 4 = 32$ bytes but results in requests with a combined size of 256 bytes per thread.

To answer the question, the total data requested per thread is $4 + 256$ bytes per thread. (The threads actually need $4 + 32$ bytes each.)

(*b*) Explain why the global memory access pattern is wasteful and show how it can be improved with a code modification.

Accesses to `stable` are not sequential. Each half warp generates sixteen requests instead of 1. The solution is to increment `i` by the block size (or some other value that would result in consecutive threads accessing consecutive items). A possible solution appears below:

```
int pf_idx = threadIdx.x;
for ( int i=0; i<my_share; i++ )
  {
    stable_copy[pf_idx] = stable[pf_idx];
    pf_idx += blockDim.x;
  }
```

(*c*) Would a texture cache improve access to `stable`? Explain, and be specific referring to values of `idx` and `i` in your explanation.

Yes, because even though the entire block of data brought into the texture cache would not be used in the first `i` iteration, it would presumably be available for subsequent iterations. Also, the scattered accesses would not require multiple issues of the same instruction.

(*d*) The amount of serialization (the need to take extra cycles to issue instructions) due to shared memory access varies with the value of `tree\_level`. Let a serialization degree of 0 indicate no serialization, meaning that instructions do not spend any extra time in a core, a 1 means one extra cycle (because memory instructions must be issued in two passes), etc. Plot the best-case and worst-case serialization degrees (y-axis) versus `tree_level` (x-axis). *Hint: In the best case the value of* `target` *is the same for all threads.*

Do this for a device of compute capability 1.3.

First, note that the shared memory bank number being accessed in `stable_copy[ti_maybe]` is the four least-significant bits of `ti_maybe`.

Recall that there will be no serialization (in 1.3 or 2.0) if all threads in a half warp access the same shared address or if each accesses data in a different bank.

Best Case: For the best case all threads read the same value of target, and so are all reading the same shared memory location. That special case, called broadcasting, can be done without serialization. *Grading Note: A better question would have been: the best case given that all values of* `target` *are distinct.*

Worse Case: For the worst case we need to have all threads in a half warp accessing the same bank (maximizing conflict) and accessing at least two different addresses (avoiding the efficiently handled broadcast special case).

The four least significant bits of `ti_maybe` will be zero up until the last four iterations. For the worst case scenario at least one thread per half warp must have a different `target < elt` outcome than the others in the first iteration. That will result in two different addresses to the same bank, and 15 extra cycles per half warp in devices of capability 1.3 or lower. (Even if there are only two different addresses.)

This serialization will continue to be present, no mater subsequent `target < elt` outcomes, until the last four iterations. The last four iterations could spread accesses over multiple banks, reducing serialization. That would normally be good, but here we are trying to achieve the worst case, and so we need to force all threads to use the same bank. That can be done by choosing `target[threadIdx.x]` so that in the last four iterations the `target < elt` outcomes are identical for all threads in a half warp.

The resulting execution would have a serialization of 15.

(*e*) Repeat the problem above for Fermi, that is a device of compute capability 2.0.

In Fermi two reads to the same bank and to the same address (referred to as a multicast) can complete without serialization. (For earlier devices all addresses had to be identical to avoid serialization with multiple accesses to the same bank.)

To achieve the worst case we need to generate as many distinct addresses as possible. That would happen by spreading target values out. If that's done there would be no serialization in the first iteration, serialization of 1 in the second, 3 in the third, it would reach a maximum of 15. In the last four iterations the `target < elt` outcomes would have to be identical to avoid spreading the accesses out.

A plot of serialization vs. `tree_level` appears in the ASCII art graph below.

```
tree_level
!
!     Serialization
V     0   1  3  7   15
10   BW12
 9    B   W2        W1
 8    B      W2     W1
 7    B         W2  W1
 6    B             W12
 5    B             W12
 4    B             W12
 3    B             W12
 2    B             W12
 1    B             W12
 0    B             W12
```

Problem 2: [15 pts] The code below can suffer from branch divergence when run on a device of CC 1.3. Suppose that without branch divergence core utilization would be 100%. That is, every core is busy with an active thread every cycle. Also suppose the code is launched in a configuration with block size $128 \times 3 = 384$, and 128 threads find a value of `0` for `my_plane`, 128 threads find a value of `1` for `my_plane`, and 128 threads find `2`.

```
__device__ void diverge(int tsize)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const float4 my_pos = targets[idx];
  const int my_plane = planes[idx];
  float distss = 0;

  for ( int i=0; i<tsize; i++ )
    {
      float4 elt = table[i];
      float dist;
      switch ( my_plane ) {
      case 0: dist = my_pos.x - elt.x; break;
      case 1: dist = my_pos.y - elt.y; break;
      case 2: dist = my_pos.z - elt.z; break;
      }
      distss += dist * dist + 1.0f / dist;
    }

  results[idx] = distss;
}
```

(*a*) With the distribution of `my_plane` values given above, it is possible that there will be no branch divergence at all or that every thread suffers the maximum divergence. Describe the conditions necessary for each extreme.

☑ Conditions for no divergence.

All threads in a warp get the same value of `my_plane`.

☑ Conditions for maximum divergence.

Each of the three values are accessed by some thread in a warp.

(*b*) Provide an estimate of core utilization under the worst-case divergence. Several assumptions need to be stated about the code to answer this question, such as the number of instructions in various parts of the code. In terms of these provide a formula or number for core utilization.

The degree of impact of divergence on utilization depends on how much time is spend "diverged." The case statements consist of only a single subtraction, probably followed by a jump. The surrounding code would have many more instructions: 3 arithmetic operations for the `distss` line, 1 instruction for the load of `table` the loop instructions (say 3), and the instructions for the switch (say 3). The total is then 12 for the loop, of which 2 are diverged. Without divergence a path through the code would be 12 instructions, with divergence it would be $10 + 3 \times 2 = 16$. So utilization would be roughly $\frac{12}{16}$.

Problem 3: [15 pts] The kernel below can execute two ways, if `opt_plan_a` is true then it executes the same as the code in the previous problem. If `opt_plan_a` is false, call that Plan B, then the alternative code runs which does the same computation but avoids the switch statement overhead inside the loop.

```
__device__ void
plan_a_or_b(int tsize)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const float4 my_pos = targets[idx];
  const int my_plane = planes[idx];
  float distss = 0;

  if ( opt_plan_a )
    { // Plan A
      for ( int i=0; i<tsize; i++ )
        {
          float4 elt = table[i];
          float dist;
          switch ( my_plane ) {
          case 0: dist = my_pos.x - elt.x; break;
          case 1: dist = my_pos.y - elt.y; break;
          case 2: dist = my_pos.z - elt.z; break; }
          distss += dist * dist + 1.0f / dist;
        }
    } else
    { // Plan B
#     define LOOP(comp) \
      for ( int i=0; i<tsize; i++ ) \
        {  const float dist = my_pos.comp - table[i].comp; \
          distss += dist * dist + 1.0f / dist;  }

      // Each case statement below executes entire loop, not just an iteration.
      switch ( my_plane ) {  case 0: LOOP(x); break;
                             case 1: LOOP(y); break;
                             case 2: LOOP(z); break;  }
    }

  results[idx] = distss;
}
```

(*a*) If there is no divergence Plan B is better. Which is better if there is divergence, Plan A or Plan B? Explain.

Plan A, because with Plan B the time spent diverged will include the entire loop, not just the code in a switch case statement.

(*b*) Modify the code so that `opt_plan_a` is set to false if there is no divergence. The kernel must determine whether there is divergence, it can't just read a pre-computed yes/no value.

Three solutions are shown below. (They have not yet been tested so they might have syntax errors or subtle flaws.)

The best method is shown first, which uses a CUDA warp vote function, `__all()`, in this case, which returns true if all threads in a warp call it with a non-zero value.

The second method uses an atomic AND. First, each thread converts `my_plane` into a bit vector, `my_plane_vec`, in which a bit position rather than an integer value indicates the case to use. The atomic AND is used to take a bitwise and of all of a warps values of `my_plane_vec`. There will be no divergence if the value is non-zero. It is possible for hardware to implement this so that the

atomic AND is fast, but that's unlikely. So even though the path through the code is only a few instructions, the time needed for the atomic AND will be large.

The third method iterates over all threads. It also uses the `my_plane_vec` vector, but uses 32 times more shared memory and a 32-iteration loop. Note that it doesn't matter if all threads execute the loop or just one thread per warp.

A variation of method 3 is to use a reduction tree rather than a loop. That would require just five iterations and would probably be the second fastest method.

```
// SOLUTION: Three methods shown.

  // Use warp vote function to select plan.
  // This is the best method.
  const bool opt_plan_a =
    __all( my_plane == 0 ) || __all( my_plane == 1 ) || __all( my_plane == 2 );

  // Use atomic operations.
  // This method would only work well if the atomic AND operation did not
  // serialize the warp.
  const int MAX_WARPS = 48;
  __shared__ int our_planes[MAX_WARPS];
  const int our_warp = threadIdx.x >> 5;
  const int my_warp_pos = threadIdx.x & 0x1f;
  if ( my_warp_pos == 0 ) our_planes[our_warp] = 0x7;
  const int my_plane_vec = 1 << my_plane;
  atomicAnd(&our_planes[our_warp],my_plane_vec);
  const bool opt_plan_a = our_planes[our_warp] == 0;

  // Brute force method: serially examine each thread.
  const int MAX_THREADS = 48 << 5;
  __shared__ int our_planes[MAX_THREADS];
  const int my_plane_vec = 1 << my_plane;
  our_planes[threadIdx.x] = my_plane_vec;
  const int our_plane_vec = 0x7;
  const int our_warp_thd_0 = threadIdx.x & ~0x1f;
  for ( int i=0; i<32; i++ )
    our_plane_vec &= our_planes[our_warp_thd_0+i];
  const bool opt_plan_a = our_plane_vec == 0;
```

Problem 4: [12 pts] Answer the following GF3 instructions.

The instructions in the GF3 can swizzle (rearrange vector components) and negate any of their vector operands. Consider an alternative design in which instructions could not swizzle and negate their operands, instead a separate swizzle instruction would have to be executed.

```
! GF3 Code
 add r1, r2.yzwx, -r3.zyxw

! Alternative Design, need to call new swizzle instructions.
 swizzle r12, r2.yzwz
 swizzle r13, -r3.zyxw
 add r1, r12, r13
```

(*a*) There is an obvious reason why this would reduce hardware cost (ignoring the need for more instruction memory). What is it?

The vector processor would only need one swizzle unit. That's a savings of one swizzle unit over the two that are required for a regular gf3 (one for each source operand). [Or is that three?]

(*b*) Why might the alternative design need fewer input and output buffers? That is, why might fewer threads be needed to attain full efficiency? *Hint: Think about how the number of threads is chosen.*

In the real gf3 all source operands pass through a swizzle unit, so it's likely one of the pipeline stages consuming a cycle of latency. In the alternative design that stage is no longer present, swizzling would be performed by a functional unit in the same position in the pipeline as, perhaps, the floating-point arithmetic unit.

Therefore in the alternative design pipeline latency might be one cycle shorter and so there is one cycle less of latency to cover. The number of threads is chosen to cover latency and so one less thread would be needed. With one less thead one needs one less input and output buffer.

(*c*) The following gf3 question is unrelated to the two above. The design of the gf3 limited the number of vertex attributes to 16. Describe why the following statement is not fully true: "Increasing the number of attributes to 32 would require more data transfer time (or instead bandwidth) slowing the processing of all vertices while only providing a functional benefit to vertices using more than 16 attributes."

In the gf3 there is no need to transfer data for attributes that don't change. Therefore code that did not use the new attributes (or did not change them frequently) so there would be no additional data transfer and so no additional data transfer time.

Problem 5: [8 pts] Answer each texture-related question below.

(*a*) Explain why long lines appropriate for a CPU cache is not the best organization for a texture cache.

Texture access typically has 2-dimensional spatial locality, meaning that if texel (s, t) has been accessed, texel $(s, t+\delta)$ or $(s+\delta, t)$ might soon be accessed, where δ is the distance between texels. Let A denote the address of texel (s, t). Then $(s, t + \delta)$ might be at address $A + 1$, but it might also be at address $A + W$, where W is the width in texels of the texture (or the texture tile). Suppose a cache line were the size of 16 texels. A CPU-like cache would bring in texels (s, t) to $(s + 15\delta, t)$, but a texture cache would bring in several rows of texels: $(s, t - \delta)$ to $(s + 3\delta, t - \delta)$, (s, t) to $(s + 3\delta, t)$, $(s, t + \delta)$ to $(s + 3\delta, t + \delta)$, and $(s, t + 2\delta)$ to $(s + 3\delta, t + 2\delta)$. Note that the location of the demand-fetched texel, (s, t), within the cache line (or lines) depends on the texel memory address.

(*b*) The texture mipmap level can be chosen in a vertex processor or a fragment processor. Explain why higher quality results are obtainable if a mipmap level is chosen in a fragment processor.

The mipmap level is based how much of the texture image a pixel covers. Consider a textured triangle with one vertex close to the viewer and the others far away. For the vertex close to the viewer we want a low MIPMAP level (larger texture image at that level), for the distant vertices we want a high level (small texture images).

If the MIPMAP level were chosen in the vertex processor and could not be changed during interpolation, then it would be too high or too low, depending on whether the near or far vertex was used. If the MIPMAP level is too high then the texture will look blurry, if it's too low there will be sampling artifacts (for example, a picket fence might not be visible). If the MIPMAP level can be interpolated or chosen in the fragment shader, then it will be appropriate for the fragment being rendered.

Problem 6: [12 pts] Answer each question below.

(a) Based on the g80 organization and the precedent of gf3, we expect that frequently changing the transformation matrix may slow down execution, whereas frequently changing the color of a vertex would not slow down execution. Why do we expect a matrix change to slow down execution whereas changing color does not. In your answer consider a case in which each vertex has a different color, but the transformation matrix is changed once every eight vertices.

By design of OpenGL the transformation matrix is part of the state, data associated with the whole program (or rendering context, to be precise), whereas color is an attribute, something associated with an individual vertex. Enough storage is needed so that each vertex can have a full set of attributes, whereas all vertices share state. If state is changed we must wait for all of the vertices using the old state to complete before we can change the state. This results in a delay.

(b) Consider a GPU microarchitecture like the g80 except that a warp has only 8 threads. Other things are unchanged, in particular, an SM still has 8 cores and instruction latencies are unchanged. Each cycle the scheduler chooses a different warp to execute. (On the g80 and gt200 a new warp is chosen every four cycles.)

☑ How might the 8-thread warp affect hardware cost or clock frequency?

The scheduler now has to choose a warp every cycle rather than every four cycles. That might take more hardware. For example, the warp scheduler needs to check which warps are ready to execute. On the g80 it can do so using hardware that can check $\frac{1}{4}$ of the threads, using it on four successive cycles. If it had to issue a warp every cycle it would have to scan all threads every cycle, requiring more hardware to do so and possibly taking more time.

(c) In the GTX 8800 (and similar GPUs) frame buffer update is performed by separate raster op processors (rops). The rops are not programmable and so we are stuck with the frame buffer update options provided. For example, we might want to update the frame buffer red channel with $a_c r_c + (1 - a_c) a_f r_f$, where a_c and r_c are alpha and red-channel values for the fragment with the smaller z value and a_f and r_f are alpha and red-channel values for the fragment with the larger z value. There is no such update option in OpenGL and won't can't implement it in a fragment shader because one can't read the frame buffer. (The fragment shader knows it's values of a, r, and z but it does not know if they are the close or far set.)

Why not let the fragment shaders read the frame buffer in GPUs such as the GTX 8800 series?

Frame buffer updates need to be atomic. That means that a particular fragment has to have exclusive access to a pixel (frame buffer location) from the time it reads it (retrieving, perhaps, z and stencil values), until the time it updates it (perhaps a new z value and colors).

This duration of this exclusive access is short if the update is done from the raster processor, because that is located close to the memory port. The operation (and exclusive access) can start when the data arrives and will be short in scope both because of the proximity of the data and because the range of operations is restricted.

In contrast, exclusive access would last much longer if it were granted to code running on the multiprocessors. Data would have to be transferred back and forth between the raster processor and stream processor, perhaps adding tens of cycles or more. But what would be much worse would be the code on the multiprocessors having to wait for global memory access or otherwise taking a long time. The effect of that would be to block other shaders trying to write the same pixel, causing a backlog. There is also the issue of a much lower pixel write rate.

Problem 7: [10 pts] Two code fragments appear below, one using a triangle strip, the other using individual triangles. Both render the same triangles. Comment on the amount of benefit realized by using triangle strips and explain why there is benefit or why there isn't benefit.

```
// Using individual triangles
glBindBuffer(GL_ARRAY_BUFFER,gpu_coor_buffer_individual_triangles);
glVertexPointer(3,GL_FLOAT,sizeof(pCoor),NULL);
glBindBuffer(GL_ARRAY_BUFFER,0);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_TRIANGLES,0,num_v_individual_triangles);
glDisableClientState(GL_VERTEX_ARRAY);

// NOTE: Either this code runs or the one above, not both.

// Using triangle strips
glBindBuffer(GL_ARRAY_BUFFER,gpu_coor_buffer_triangle_strip);
glVertexPointer(3,GL_FLOAT,sizeof(pCoor),NULL);
glBindBuffer(GL_ARRAY_BUFFER,0);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_TRIANGLE_STRIP,0,num_v_triangle_strips);
glDisableClientState(GL_VERTEX_ARRAY);
```

☑ Benefit for GPU / CPU communication, reason.

With a triangle strip all but four vertices are part of three triangles. A vertex sent just once as part of a strip would have to be sent three times when rendered as individual triangles. Therefore, data communication is reduced by two thirds.

☑ Benefit for vertex shader workload, reason.

The vertex shader is run on each vertex. There are nearly one third as many, so the vertex shader workload is reduced by two thirds.

☑ Benefit for fragment shader workload, reason.

Since the exact same triangles are rendered, the same fragments will be generated. There will be no change in workload. (Except perhaps for shared edges.)

Problem 8: [12 pts] Below are CUDA code fragments followed by generated PTX code. Describe the problem revealed by the PTX and how the code might be improved using this info.

(*a*) The CUDA code below does something reckless (or foolish or overly trusting), but the PTX code shows that the compiler was smart, avoiding the problem.

```
 // CUDA

      const float3 pos1 = xyz(balls_x.position[bidx1]);
      const float radius1 = balls_x.position[bidx1].w;
      const float3 vel1 = xyz(balls_x.velocity[bidx1]);

 // PTX Code
ld.const.u64  %rd11, [balls_x+0];
add.u64  %rd12, %rd10, %rd11;
ld.global.v4.f32  {%f1,%f2,%f3,%f4}, [%rd12+0];
.loc 3 1029 0
ld.const.u64  %rd13, [balls_x+16];
add.u64  %rd14, %rd10, %rd13;
ld.global.v4.f32  {%f5,%f6,%f7,_}, [%rd14+0];
```

☑ Explain, what the compiler did.

The compiler used one 4-element vector load instruction to get values for `pos1` and `radius1`, even though the CUDA code showed them as two separate array accesses.

☑ Modify the PTX code, showing how a less capable compiler would not save the user from reckless coding.

```
// Solution

 // PTX Code
ld.const.u64  %rd11, [balls_x+0];
add.u64  %rd12, %rd10, %rd11;
ld.global.v4.f32  {%f1,%f2,%f3,_}, [%rd12+0];
ld.global.f32  %f4, [%rd12+12];
.loc 3 1029 0
ld.const.u64  %rd13, [balls_x+16];
add.u64  %rd14, %rd10, %rd13;
ld.global.v4.f32  {%f5,%f6,%f7,_}, [%rd14+0];
```

Problem 8, continued:

(*b*) Shown below are portions of cuda and the corresponding ptx code used to collect offsets. (This is an alternate implementation of the code in the repo.) The offsets are held in a union. For writing individual offsets the c member is used, which is an array of eight characters. Once all offsets have been written into the array the entire array is written to memory using the integer member.

```
// CUDA Code  (Excerpts, surrounding code not shown.)

  // Initialize
  union { int64_t i; char c[8];} offsets_u;
  offsets_u.i = 0;

  // Assign offset  (this code within a loop)
  offsets_u.c[proximity_cnt] = offset;

  // Write offset to memory.
  cuda_prox[idx9] = offsets_u.i;


// PTX Code

        // Initialize
mov.s64  %rd15, 0;
st.local.s64  [__cuda___cuda_offsets_u_0104+0], %rd15;

        // Assign offset
cvt.u64.s32  %rd25, %r11;
add.u64  %rd26, %rd25, %rd16;
st.local.s8  [%rd26+0], %r15;


        // Write offset to memory.
ld.local.s64  %rd27, [__cuda___cuda_offsets_u_0104+0];
st.global.s64  [%rd7+0], %rd27;
```

The PTX code indicates that execution will be slow.

☑ Why will execution be slow?

The PTX code shows that there is local memory access, which is slow, as slow as global memory.

☑ What about the CUDA code led the compiler generate this slow code?

The code accesses the offsets union using an array index, `proximity_cnt` in the following statement `offsets_u.c[proximity_cnt] = offset;`.