LSU EE 4702-1 Homework 7 Solution Due: 1 December 2016

The solution has been checked into the repo. For a syntax highlighted version of the shader code see `http://www.ece.lsu.edu/koppel/gpup/2016/hw07-cuda-sol.cu.html`.

Problem 0: If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit files `hw07-cuda.cu` and `hw07.cc`. **Find a machine with GPUs of CC (compute capability) 3.0 or higher,** see the table on `http://www.ece.lsu.edu/koppel/gpup/sys-status.html`. Compile and run the homework code unmodified. It should initially show the spring from classroom demo `demo-cuda-04-acc-pat.cc`.

The physics for the spring can be performed using three different sets of code, the one in use is shown at the beginning of the last line of green text next to Physics. The physics code can be changed by pressing `a`. Physics code *CPU* uses the CPU and will be very slow. Physics code *CUDA-M1* and *CUDA-M2* use the GPU running code in file `hw07-cuda.cu`. Code *CUDA-M1* uses routine `time_step_intersect_1` to detect and resolve intersection of helix segments, and *CUDA-M2* uses routine `time_step_intersect_2`.

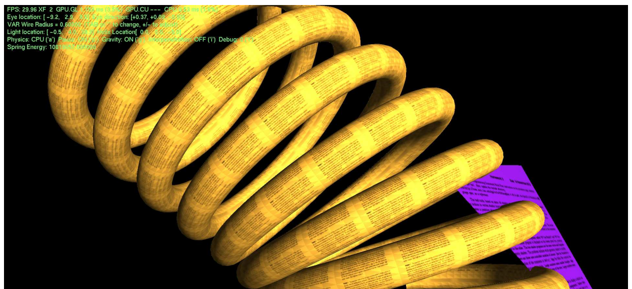
The green line starting with "Intersect" shows information about the execution of the chosen intersection routine. As discussed in class, it is this routine that will dominate performance. On that line "Bl Sz" is the number of blocks. That can be changed using the `Tab` key to find "Intersect Block Size" and then pressing `+` and `-` keys. Green text "N Blocks" shows the number of blocks. The number of blocks is determined by variable "Seg / Block" (`intersect_seg_per_block`). Green text "Bl/SM" shows the number of blocks per SM in two ways. The first value is the number of blocks running the intersect routine that *can* fit on an SM. The second number is how many are assumed running based on the number of blocks and the number of SMs. If the second number is lower than there are not enough blocks. Green text "Wp/SM" shows the number of warps per SM. That's based on the block size and the number of blocks per SM. Finally, "Bl Time" shows the number of cycles it takes to run one block. (The top green line shows timing per frame.)

On the last green line, "Shared Mem" shows how shared memory is being used to buffer the "b" segments in the intersection routine. "NONE" means that shared memory is not being used, "SYNC EXPER" is something to be used in Problem 2, "ONE ITER" means that shared memory is being loaded with "b" segments every `j` iteration (see the intersect routines), and "MULT ITERS" means that shared memory is loaded with enough data for several `j` iterations—when Problem 3 is solved correctly.

The top green line shows performance. The important items to pay attention to are "GPU.CU" and "Steps / s". "GPU.CU" shows the time per frame executing CUDA code. Use this to evaluate the performance of the intersection routines. The last item on the line (you may need to widen the window), "Steps / s", shows how many physics time steps are performed each second. If computation is fast enough it will be about 6000, otherwise it will be lower. "FPS" is the frame rate in frames per second. "XF" shows the frame update interval, an ideal number is 1. A 2 indicates that the frame is updated at half the display's refresh rate. "GPU.GL" is the time the GPU spends executiong OpenGL code. "CPU GR" is the CPU time for graphics. "CPU PH" is the CPU time for physics.

Fun Stuff: Pressing `g` will toggle gravity. The free end of the spring can be grabbed by pressing `b` and it can be moved using the arrow keys.

Initially the arrow keys, PageUp, and PageDown can be used to move around the



```
FPS: 29.96 XF  2  GPU.GL 1.153 ms (3.5%)  GPU.CU ---  CPU 0.53 ms (1.6%)
Eye location: [ -9.2,   2.9,   8.5]  Eye direction: [+0.37, +0.09, -0.93]
VAR Wire Radius = 0.600000  (TAB or ` to change, +/- to adjust)
Light location: [ -0.5,   4.0,  38.9]  Helix Location[  0.0,   5.0,  -5.0]
Physics: CPU ('a')  Pause: ON ('p')  Gravity: ON ('g')  Interpenetration: OFF ('i')  Debug: 0 ('f')
Spring Energy: 10819557.000000
```

scene. Press (lower-case) `b` and then use the arrow and page keys to move the free end of the spring around. (Since the motion is instantaneous, the spring will act like it was plucked.) Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a $5\times$ greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw07.cc` for documentation on other keys. For documentation see the CUDA C Programming Guide linked to the course `http://www.ece.lsu.edu/koppel/gpup/ref.html` page.

Problem 1: The goal of this assignment is to understand and improve the performance of the intersection routines. The performance of the intersection code is very sensitive to the block size (variable "Intersect Block Size") and to the number of a segments per block (variable "Seg / Block", called m in our classroom analysis).

(a) Indicate the model of GPU you were using (it's shown in the text printed when the program starts). Indicate the number of SMs (shown as MP) for that GPU.

The GPU is an NVIDIA K20c which has 13 MPs.

(b) Adjust these two variables to obtain the best performance on each intersection routine. Do these with shared memory set to NONE. Provide the following information: The settings of those variables. The initial CUDA execution time ("GPU.CU") and the best one obtained.

Initial CUDA time: 290 ms. The best performance is 13.2 ms with a block size of 128 threads and $m = 8$. The tables below show the raw data. Column WP Occ shows the warp occupancy (number of warps per SM), something which performance is sensitive to. The maximum for a K20 (and other CC 3.X through CC 6.x devices) is 64 warps per SM. The first table shows the impact of block size with $m = 1$. With $m = 1$ we have the maximum number of blocks, which in general is good for warp occupancy, but for not when blocks are too small because we hit an SM's block limit (16) before the warp limit (64).

The amount of data read (in units of 16-byte elements) from global memory when accessing the "a" elements is $nB/16$ (where n is the number of segments and B is the block size), and the amount of data read from global memory when accessing the "b" elements is n^2/m for $m \leq 16$. As we've noted in class larger m's mean less data loaded because there are fewer blocks and we know that each block must load at least $n + m$ elements. That means $m = 1$ is the worst value as far as data is concerned. However, the number of blocks is n/m and so smaller m's are better for warp occupancy, up to a point. That is, once we've reached the maximum of 64 warps per SM there is no benefit of having more warps. From the first table we see that we've hit that maximum at a block size of 128, which is where performance peaks. Increasing the block size reduces the number of iterations in the j loop, which increases the impact of the code outside the loop. Also, increasing the block size increases the amount of data loaded for the "a" segments. Both of those reasons might explain the drop in performance with a larger block size.

The second table shows the impact of changing m at three different block sizes. For block sizes of 256 and 128 the best performance is the smallest value of m for which warp occupancy is below 64. If m is increased further then the benefit of less data transfer is overwhelmed by the penalty of lower occupancy. Notice that at a block size of 64 the maximum warp occupancy is 32 and so the performance of those configurations suffers.

Vary Block Size with m = 1:

B	GPU.gl	WP Occ	j Iters
32	40.9 ms	16	16
64	18.6 ms	32	8
128	15.1 ms	64	4
256	17.7 ms	64	2
512	37 ms	64	1
1024	90 ms	64 (32)	1

	-- B = 256 -----		-- B = 128 -----		-- B = 64 -----	
m	GPU.gl	WP Occ	GPU.gl	WP Occ	GPU.gl	WP Occ
1	17.7 ms	64	14.6 ms	64	18.6 ms	32
2	14.2 ms	64	14.0 ms	64	20.0 ms	32
4	14.8 ms	64	13.2 ms	39.4	19.8 ms	19.7
8	13.6 ms	39.4	19.8 ms	19.7	72 ms	9.8
16	20 ms	19.7				

(c) Based on our classroom analysis there was less data communication with a larger value of m. Are your results consistent with that? If not, explain why.

Based on the discussion in the solution above, the results are consistent, performance improves with larger m. Until, that is, warp occupancy suffers, then performance drops.

Problem 2: Set CUDA-M1 to the optimal parameters found in the previous problem. Now, cycle through the different shared memory options. The "ONE ITER" option, which caches enough data for one iteration, should be the slowest. Is it because of the syncthreads?

Modify routine `time_step_intersect_1` so that when `hi.opt_sm_option` is set to value `SMO_sync_experiment` syncthreads is called in a way similar to when `hi.opt_sm_option` is set to `SMO_one_iteration`, but without actually loading or using shared memory. Do so in a way that will determine whether syncthreads is making "ONE ITER" take longer.

Describe what you found by doing the experiment.

The sync experiment version executed like "none" except that syncthreads was called twice. The performance was almost identical to "none" and so it is not syncthreads that was causing the slowdown.

Problem 3: Modify `time_step_intersect_1` so that when `hi.opt_sm_option` is set to value `SMO_multiple_iterations` array `pos_cache` is loaded with B items, where B is the block size (value of `blockDim.x`). This should be done inside the `j` loop when data is needed. That is, **don't** load the cache every iteration.

Compare the performance to NONE and ONE ITER.

A new variable `cache_idx_next` is used to keep track of which element of `pos_cache` each thread uses. The range of values is 0 through the block size minus 1. It is incremented each iteration by d (`thd_per_a`):

```
float3 b_position =
  hi.opt_sm_option == SMO_one_iteration
  ? pos_cache[ b_idx_start ] :
  hi.opt_sm_option == SMO_multiple_iterations
  ? pos_cache[ cache_idx_next ]
  : m3( helix_position[j] );

cache_idx_next += thd_per_a;
```

When `cache_idx_next` exceeds the block size it's time to load new data into `pos_cache` and set `cache_idx_next` back to its initial value, `b_idx_start`. Variable `b_idx_next` is used to indicate which "b" segment to load, that's incremented by the block size each time `pos_cache` is reloaded:

```
else if ( hi.opt_sm_option == SMO_multiple_iterations )
  {
    if ( cache_idx_next >= blockDim.x )
      {
        __syncthreads();
        cache_idx_next = b_idx_start;
        pos_cache[ threadIdx.x ] = m3(helix_position[ b_idx_next ] );
        b_idx_next += blockDim.x;
        __syncthreads();
      }
  }
```

Variable `cache_idx_next` is initialized to an out-of-range value so that on the first iteration `pos_cache` is loaded. Variable `b_idx_next` is initialized to the first "b" segment to load:

```
int cache_idx_next = b_idx_start + blockDim.x;
int b_idx_next = threadIdx.x;

for ( int j=b_idx_start; j<hi.phys_helix_segments; j += thd_per_a )
  {
```

Here is an excerpt of the code in order:

```
const float3 a_position = m3(helix_position[a_idx]);

/// SOLUTION -- Problem 3
//
//  The next element of pos_cache to use. Its value should be
//  between 0 and blockDim.x (block size) -1. It is intentionally
//  initialized to an out-of-range value so that the cache will be
//  loaded.
//
int cache_idx_next = b_idx_start + blockDim.x;
//
//  The next element of helix to put into the cache.
//
int b_idx_next = threadIdx.x;

for ( int j=b_idx_start; j<hi.phys_helix_segments; j += thd_per_a )
  {
    if ( hi.opt_sm_option == SMO_one_iteration )
      {
        __syncthreads();
        if ( threadIdx.x < thd_per_a )
          pos_cache[threadIdx.x] =
            m3(helix_position[ j - b_idx_start + threadIdx.x ] );
        __syncthreads();
      }
    else if ( hi.opt_sm_option == SMO_sync_experiment )
      {
        /// SOLUTION -- Problem 2
        //
        //  See if just executing __syncthreads slows things down.
        //
        __syncthreads();
        __syncthreads();
      }
    else if ( hi.opt_sm_option == SMO_multiple_iterations )
      {
        /// SOLUTION -- Problem 3
        //
        // If the next pos_cache element to use is out of range, then
        // load pos_cache with a new batch of data.
        //
        if ( cache_idx_next >= blockDim.x )
          {
            __syncthreads();
            cache_idx_next = b_idx_start;
            pos_cache[ threadIdx.x ] = m3(helix_position[ b_idx_next ] );
            b_idx_next += blockDim.x;
            __syncthreads();
          }
      }

    /// SOLUTION -- Problem 3
    //
    //  For the multiple iteration case the index to pos_cache
    //  is a function of j.
    //
    float3 b_position =
      hi.opt_sm_option == SMO_one_iteration
      ? pos_cache[ b_idx_start ] :
      hi.opt_sm_option == SMO_multiple_iterations
      ? pos_cache[ cache_idx_next ]
      : m3( helix_position[j] );

    /// SOLUTION -- Problem 3
    //
    cache_idx_next += thd_per_a;

    pVect ab = mv(a_position,b_position);

    // Skip if segment is too close.
    if ( abs(a_idx-j) < min_idx_dist ) continue;

    // Skip if no chance of intersection.
    if ( mag_sq(ab) >= four_wire_radius_sq ) continue;

    // Compute intersection force based on bounding sphere, an
    // admittedly crude approximation.
    //
    pNorm dist = mn(ab);
    const float pen = 2 * hi.wire_radius - dist.magnitude;
    float3 f = pen * hi.opt_spring_constant * dist;

    // Add force to shared variable. This is time consuming
    // (especially in CC 3.x and older GPUs) but done
    // infrequently. (A segment can normally only intersect a a few
    // other segments.)
    //
    atomicAdd(&force[a_local_idx].x,f.x);
    atomicAdd(&force[a_local_idx].y,f.y);
    atomicAdd(&force[a_local_idx].z,f.z);
    //
    // Optimization Note: Could acquire a lock and then update
    // all three components.
```

```
}
```