LSU EE 7700-1 **Homework 4** Solution **Due: 28 April 2010**

To complete this assignment follow the setup instructions from the course Web page (if not already followed). The setup instructions bring you to the point where you can compile the cpu-only examples but for this assignment that won't be necessary. Then follow the instructions in the first problem to complete and test the setup.

Problem 0: Check out the latest copy of the balls project base code and the include directory. The transcript below shows the steps needed.

(*a*) Read the following overview of how the code operates.

This code includes something like a solution to Homework 3, except that direct cuda calls were not made and the code returns the actual proximity pairs, not just a count. The code computing proximity pairs runs inefficiently due to data access problems and other issues, the point of this homework assignment is to improve it.

Here is an outline of how the affected code operates:

The code in `balls.cc:cuda_contact_pairs_find` sorts the balls (phys) in z order using class PSList. An array on cuda, `z_sort_indices`, maps a z-order index to a balls array index. For example, `z_sort_index[5] == 12` means that ball number 12 (the index to cpu array `physs` and gpu arrays `ball_x`) has the sixth smallest z value. This array is initialized in `balls.cc:cuda_contact_pairs_find` and sent over to CUDA. The z max values are also placed in an array and sent to CUDA.

The routine launches a CUDA kernel, `pass_sched`, to compute the proximity pairs, and reads back this data, in CUDA array `cuda_prox`. The data in `cuda_prox` is unpacked in routine `contact_pairs_find`.

The problems below are related to the data moving between CPU and GPU, and data moving between the different memory areas in the GPU. Here is a summary of data movement that the proximity code currently induces: The kernel `pass_sched` reads the `z_sort_indices` and z max arrays, and also reads the data in `balls_x` that is already present on the GPU. (That is, the data for `balls_x` is only sent to the GPU when balls are added or removed. Otherwise the GPU keeps using the copy it has.) The kernel writes only the array `cuda_prox`.

Each thread of the kernel, `pass_sched` in file balls-kernel.cu, performs one outer iteration of the `idx9` loop from `contact_pairs_proximity_check`. That is, each thread has its own `idx9` value and must iterate `idx1` values until there is no possibility of proximity.

Unlike `contact_pairs_proximity_check`, `pass_sched` does not check for proximity between tiles and balls, only balls and balls. (This was to make the assignment less tedious.) The CPU code, in `contact_pairs_proximity_check`, prepares a Contact structure for each nearby idx1 it finds. The corresponding code in `pass_sched` simply passes the values idx1s back to the CPU code. It does this by compressing idx1 values to 8 bits (by subtracting them from idx9) and backs them in to a 64 bit integer names `offsets`. The CPU code uses offsets to create the contact structures.

See the comments in the code for additional information.

(*b*) Familiarize yourself with the relevant UI:

Various features can be toggled on and off, and their state is shown in green at the top of the display. An indicator will blink if the corresponding should be turned off when measuring timing. This includes CPU physics and correctness verification.

Press 'F4' to toggle CUDA proximity checking.

Press 'v' to toggle verification. If proximities are incorrect execution will exit with an abort. Within a debugger you can issue the command "frame 2" (or sometimes "frame 3") to show where

the verification failed. There will be a statement like `ASSERTS( something )` where something is false but should be true. Verification **is time consuming** and so there will be no speedup as long as its on. That's why the indicator blinks when it is on.

Press 't' to run a benchmark that drops thousands of balls on the platform with certain constants, such as friction set to special values. (Edit `benchmark_setup` to change the behavior.)

Press 'T' to run the benchmark with fewer balls, this is helpful when on a slower system or testing slower code.

Press 'a' to switch between CPU and GPU physics.

Press 'c' to toggle between natural colors and color based on CUDA block assignment. (See VAR 'Color by Bock in Pass'.)

Press 'q' to toggle the value of variable `World::opt_debug` between true and false. The current value is the first number after "Debug Options." The variable `World::opt_debug` currently does nothing, it can be used to test ideas for your solution. Variable `opt_debug` is also available in CUDA.

Press 'Q' to toggle the value of variable `World::opt_debug2`. This also does nothing and is available for working on the assignment. Variable `opt_debug2` is also available in CUDA.

Timings for the scheduling pass are shown on the upper right (it will be necessary to make the window wider). CUDA Prox shows the time in the routine `World::cuda_contact_pairs_find()`, Sched shows the total time for scheduling, `CPU Prox` shows the time in `World::contact_pairs_find()`. When there are many balls (press t) and the CPU is computing proximity the value of CPU Prox should be a large fraction of Sched. With CUDA computing proximity CPU Prox will be much smaller, it's not zero because the CPU must still compute tile interactions and it must unpack the CUDA data. These timings are provided by `frame_timer.user_timer_X` calls, feel free to add your own calls to time other parts of the code.

The F2010 HW4 line shows whether CUDA proximity testing is one. The CPU test/ball shows how many proximity tests per ball are performed; total shows the total number of tests.

(*c*) Check out the code using the following transcript. Don't forget to update the include directory.

```
# Transcript of commands to check out and run code for
# this assignment.

    [prompt] % cd ~
    /home/faculty/koppel

# Check out a copy of the balls code into a directory named hw4.
    [prompt] % svn co -r 242 https://svn.ece.lsu.edu/svn/gp/proj-base/balls hw4
    A    hw4/balls-shdr.cc
    A    hw4/balls.png
# ...
    Checked out revision 242.

# Check out or update the include directory.

# Because the repository include directory has changed this step needs
# to be repeated unless the last checkout was very recent.
# (An easier way to update is to execute ‘‘svn update’’ in the include
# directory.)
    [prompt] % svn co -r 242 https://svn.ece.lsu.edu/svn/gp/include
    A    include/gl-buffer.h
```

```
    A    include/misc.h
    A    include/texture-util.h
    A    include/util.h
    A    include/gp
    A    include/glextfuncs.h
    A    include/coord.h
    A    include/pstring.h
    A    include/shader.h
    A    include/cuda-util.h
    Checked out revision 242.


# Go to the balls directory (named hw4), compile, and run the code.
    [prompt] % cd hw4
    /home/faculty/koppel/example/hw4
    [prompt] % make -j 4
    /usr/local/cuda/bin/nvcc  -Xcompiler -Wall --ptxas-options=-v -I/usr/X11R6/include/
-I/opt/local/include -I../include -I../../include  -g -Xcompiler -Wno-strict-aliasing
-Xcompiler -Wno-parentheses -c balls-kernel.cu
# ...

    [prompt] % balls
    S GL_VENDOR: "NVIDIA Corporation"
    S GL_RENDERER: "Quadro FX 3800/PCI/SSE2"
    S GL_VERSION: "3.2.0 NVIDIA 190.53"
```

(*d*) Check out, build, and play a bit with the code.

Problem 1: In this problem determine various limits on how fast contact pairs can be computed. The limits will be measured in proximity tests per second, and called an execution rate. (A proximity test is what is done by an iteration of the `idx1` loop in `contact_pairs_proximity_check`.) Let n denote the number of balls and let p denote the average number of proximity tests per ball. The total work done by the code is then np proximity tests. Call the rate of proximity tests, measured in proximity tests per second, the *execution rate*.

Base your answers to the questions below on the GeForce 8800 GTX as described in the 2006 NVIDIA technical brief and on the code checked out above.

(*a*) Examine the PTX code for `pass_sched` and estimate a number of instructions in a `idx1` loop iteration. Choose a useful iteration or iterations. Provide the number of instructions and give the execution rate based on this number of instructions assuming a sufficiently large number of threads and unlimited memory bandwidth. Bonus points for usefully taking into account the fraction of proximity tests for which idx1 is nearby.

The ptx code appears below. Within the loop there are special cases for which proximity will not be calculated, in those cases `incomplete` is set to something. Assuming those cases are rare, they won't be counted.

The number of instructions in the `idx1` loop body is 66, the number that are executed when `incomplete` is not assigned and loop doesn't exit: $66 - 8 = 58$. For np proximity tests the number of instructions would therefore be $58np$.

A GeForce 8800 GTX has 16 multiprocessors each having 8 cores, together they can execute $16 \times 8 = 128$ instructions per cycle. Based on the white paper, they execute at 1.35 GHz for a peak total instruction throughput of $128 \times 1.35 \times 10^9$ instructions per second.

The limit on execution rate is found by solving $58np = 128 \times 1.35 \times 10^9$ for np. This yields a execution rate of $np = \frac{128 \times 1.35 \times 10^9}{58}$ or about 2.98 billion pairs per second.

```
 // SOLUTION: Body of idx1 loop.
$L_2_7938:
 //<loop> Loop body line 1048
.loc 3 1048 0
cvt.u64.s32  %rd17, %r8;
mul.lo.u64  %rd18, %rd17, 4;
add.u64  %rd19, %rd18, %rd16;
ld.global.f32  %f15, [%rd19+0];
setp.lt.f32  %p4, %f15, %f14;
@%p4 bra  $L_2_8194;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1050 0
add.u64  %rd20, %rd18, %rd2;
ld.global.s32  %r12, [%rd20+0];
mov.u32  %r13, 0;
setp.ge.s32  %p5, %r12, %r13;
@%p5 bra  $Lt_2_9986;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1054 0
mov.s32  %r10, 116;
bra.uni  $Lt_2_514;
$Lt_2_9986:
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1056 0
cvt.u64.s32  %rd21, %r12;
mul.lo.u64  %rd22, %rd21, 16;
add.u64  %rd23, %rd22, %rd11;
ld.global.v4.f32  {%f16,%f17,%f18,_}, [%rd23+0];
.loc 3 1059 0
add.u64  %rd24, %rd22, %rd13;
ld.global.v4.f32  {%f19,%f20,%f21,_}, [%rd24+0];
.loc 3 134 0
sub.f32  %f22, %f2, %f17;
sub.f32  %f23, %f1, %f16;
sub.f32  %f24, %f3, %f18;
mul.f32  %f25, %f22, %f22;
mad.f32  %f26, %f23, %f23, %f25;
mad.f32  %f27, %f24, %f24, %f26;
mov.f32  %f28, 0f00000000;     // 0
setp.eq.f32  %p6, %f27, %f28;
@!%p6 bra  $Lt_2_10754;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
mov.f32  %f29, 0f00000000;     // 0
bra.uni  $Lt_2_10498;
$Lt_2_10754:
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 142 0
sqrt.approx.f32  %f29, %f27;
$Lt_2_10498:
```

```
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1072 0
sub.f32  %f30, %f6, %f20;
sub.f32  %f31, %f5, %f19;
sub.f32  %f32, %f7, %f21;
ld.global.f32  %f33, [%rd23+12];
add.f32  %f34, %f33, %f4;
mov.f32  %f35, 0f3f8ccccd;     // 1.1
mul.f32  %f36, %f34, %f35;
mul.f32  %f37, %f30, %f30;
mad.f32  %f38, %f31, %f31, %f37;
mad.f32  %f39, %f32, %f32, %f38;
sqrt.approx.f32  %f40, %f39;
.loc 3 1042 0
ld.param.f32  %f12, [__cudaparm__Z10pass_schedif_lifetime_delta_t];
.loc 3 1072 0
mul.f32  %f41, %f12, %f40;
sub.f32  %f42, %f29, %f41;
setp.lt.f32  %p7, %f36, %f42;
@%p7 bra  $Lt_2_514;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
mov.u32  %r14, 7;
setp.le.s32  %p8, %r11, %r14;
@%p8 bra  $Lt_2_11266;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
mov.s32  %r10, 102;
bra.uni  $Lt_2_11522;
$Lt_2_11266:
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
sub.s32  %r15, %r3, %r8;
mov.u32  %r16, 254;
setp.le.s32  %p9, %r15, %r16;
@%p9 bra  $Lt_2_11778;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
mov.s32  %r10, 111;
bra.uni  $Lt_2_11522;
$Lt_2_11778:
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1089 0
cvt.s64.s32  %rd25, %r15;
shl.b64  %rd26, %rd15, 8;
or.b64  %rd15, %rd25, %rd26;
$Lt_2_11522:
$Lt_2_11010:
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1091 0
add.s32  %r11, %r11, 1;
$Lt_2_514:
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
.loc 3 1042 0
sub.s32  %r8, %r8, 1;
```

```
mov.u32  %r17, 0;
setp.ne.s32  %p10, %r10, %r17;
@%p10 bra  $L_2_8194;
 //<loop> Part of loop body line 1048, head labeled $L_2_7938
mov.u32  %r18, 0;
setp.ge.s32  %p11, %r8, %r18;
@%p11 bra  $L_2_7938;
bra.uni  $L_2_8194;
```

(*b*) Compute the total amount of data sent from the CPU to the GPU for n balls.

The additional data for computing proximity includes the `z sort indices` and `z sort z max`; each is 4 bytes per element or 8 bytes per ball. The total is then $8n$.

(*c*) Compute the execution rate based on the amount of data fetched from device memory to the multiprocessors. If a data item is fetched twice, count it twice, and if a memory fetch has a width of 32 bytes but only four bytes are used, count that as 32 bytes. See the CUDA Programming guide. Base the number of fetches on a machine with compute capability 1.3, but base the timing and bandwidth on the 8800. To get the execution rate first compute the total amount of data fetched for n balls and p tests per ball, then divide by the maximum data transfer rate (based on the 8800 description), then divide by np.

Refer to routine `pass_sched` in the file ball-kernel.cu r 242. The data for `idx9` is fetched once per thread, based on the tabulation below the size is 36 bytes per ball (and one thread handles one `idx9`), and the total transfer will be $36n$ bytes.

In contrast the `idx1` ball is fetched within the loop, which iterates on average p times per ball. The data that is indexed using `idx1` `z_sort_z_max` and `z_sort_indices` will be fetched coherently, but the data fetched using `bidx` will not. This is taken into account in the tabulation below, which is for a device of compute capability 1.3.

Assuming $p \gg n$, the data fetched per proximity test is $72np$ bytes. Based on the white paper, the GPU has a total off-chip bandwidth of 86.4 GBps. (See page 38 in the white paper.)

Based on a data limit the maximum execution rate is 1.2 billion proximity tests per second. Note that this is slower than the limit imposed just by instruction execution.

```
Data for the 'idx9' ball. All accesses are coherent.  Fetched once per thread.
Variable        Size
z_sort_indices   4
pos_rad9        16
vel9            16  (four elements are fetched, only 3 used)
--------------------
Total           36  per ball

Data for idx1 ball.  Fetched many times per thread.
Variable      Size   Req Size
z_sort_z_max     4   4   Consecutive threads, consecutive elements.
z_sort_indices   4   4   Consecutive threads, consecutive elements.
pos_rad         16   32  Based on minimum req size. Not coherent, index arbitrary.
vel1            16   32  Based on minimum req size. Not coherent, index arbitrary.
-----------------------
                40   72
```

(*d*) Compute the execution rate based upon the amount of data fetched, but assuming perfect coalescing and alignment and assuming unlimited multiprocessor cache (count a data item once, even if it was used multiple times).

That would reduce the amount of data per pair to 40 bytes, so the limit would be 2.16 billion pairs per second, much closer to the instruction-limited rate though still slower.

Problem 2: Improve performance of the code by using the texture cache. *Hint: This should only take a few lines of code.*

(*a*) Modify the code so that it uses the texture cache.

The solution is checked in under revision 245. To check out the solution follow the instructions in Problem 0 except replace r 242 with r 245. The modified line is shown below, followed by the changes (for those who don't want to check out the code).

```
[prompt] % svn co -r 245 https://svn.ece.lsu.edu/svn/gp/proj-base/balls hw4
```

```
Index: balls-kernel.cu
===================================================================
--- balls-kernel.cu (revision 242)
+++ balls-kernel.cu (revision 245)
@@ -983,6 +983,8 @@
 // use in debugging.
 __constant__ float3 *pass_sched_debug;

+texture<float4> balls_pos_tex;
+texture<float4> balls_vel_tex;

 __global__ void pass_sched(int ball_count, float lifetime_delta_t);
 __device__ float ball_min_z_get
@@ -990,8 +992,16 @@

 __host__ void
 pass_sched_launch
-(dim3 dg, dim3 db, int ball_count, float lifetime_delta_t)
+(dim3 dg, dim3 db, int ball_count, float lifetime_delta_t,
+ void *pos_array_dev, void *vel_array_dev)
 {
+  size_t offset;
+  const size_t size = ball_count * sizeof(float4);
+  const cudaChannelFormatDesc fd =
+    cudaCreateChannelDesc(32,32,32,32,cudaChannelFormatKindFloat);
+  cudaBindTexture(&offset, balls_pos_tex, pos_array_dev, fd, size);
+  cudaBindTexture(&offset, balls_vel_tex, vel_array_dev, fd, size);
+
   pass_sched<<<dg,db>>>(ball_count,lifetime_delta_t);
 }

@@ -1023,10 +1033,11 @@

   // Fetch position, radius (packed in position vector), and velocity.
   //
-  const float4 pos_rad9 = balls_x.position[bidx9];
+  const float4 pos_rad9 = tex1Dfetch(balls_pos_tex,bidx9);
   const float3 pos9 = xyz(pos_rad9);
```

```
   const float radius9 = pos_rad9.w;
-  const float3 vel9 = xyz(balls_x.velocity[bidx9]);
+  const float4 vel9_pad = tex1Dfetch(balls_vel_tex,bidx9);
+  const float3 vel9 = xyz(vel9_pad);

   const float z_min = ball_min_z_get(pos9,vel9,radius9, lifetime_delta_t);

@@ -1053,10 +1064,11 @@
       // (t is for tile)
       if ( bidx1 < 0 ) { incomplete = 't'; continue; }

-      const float4 pos_rad = balls_x.position[bidx1];
+      const float4 pos_rad = tex1Dfetch(balls_pos_tex,bidx1);
       const float3 pos1 = xyz(pos_rad);
+      const float4 vel_pad1 = tex1Dfetch(balls_vel_tex,bidx1);
+      const float3 vel1 = xyz(vel_pad1);
       const float radius1 = pos_rad.w;
-      const float3 vel1 = xyz(balls_x.velocity[bidx1]);

       // Use the pNorm constructor to compute the distance between two balls.
       pNorm dist = mn(pos1,pos9);
```

(*b*) Measure the change in performance and comment on it.

Solution omitted.

Problem 3: Show how shared memory can be used to improve performance. Outline how shared memory would be used, code is optional.

The key point is that the same `idx1` ball is accessed by multiple threads. The `idx1` balls could be loaded into shared memory and then shared by multiple threads. The exact procedure might be for a thread to load its idx9 ball into shared memory (which is an idx1 ball for a neighbor) then also load one or more additional `idx1` balls (which would be needed by the smaller-numbered threads in the block).

Shared memory access is faster than texture cache access, however because the `idx1` balls are not coherent, the initial loading of shared memory will reduce or eliminate the shared memory advantage.