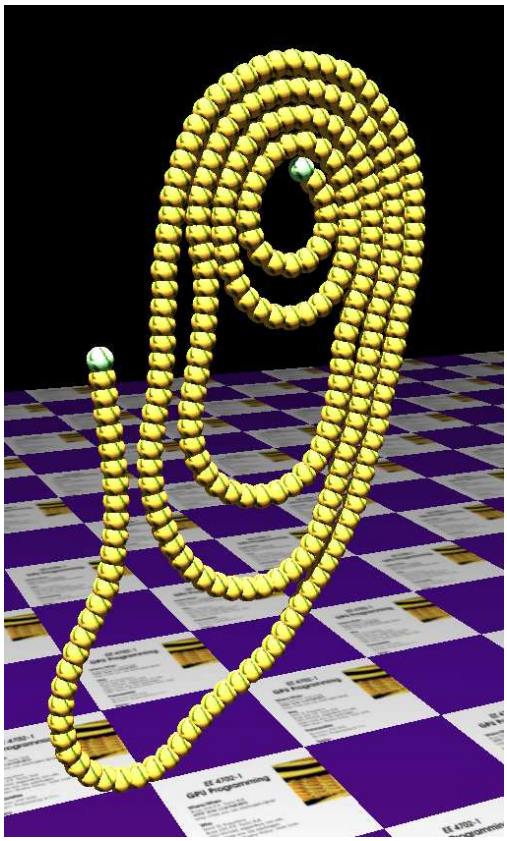
LSU EE 4702-1 **Homework 3** **Due: 20 November 2013**



Problem 0: Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for programming homework work flow, substituting hw3 where appropriate. Be sure to update the include directory in addition to checking out the homework.

Compile and run the homework code unmodified. It should initially show a string of balls in the shape of a spiral with the first and last ball fixed in space, the spiral will unravel, see the screenshot to the right.

Unless the code is run on a surprisingly fast system, the frame rate should be low, resulting in jerky movement. The physics is computed on the GPU using CUDA, the goal of this assignment is to improve the performance of the code so that animation is smooth.

For this problem, familiarize yourself with the operation of the simulation (that is, play a little). To move around the scene press `e` and then the arrow keys, PageUp, and PageDown. Press `1` to set up scene 1 (vertical string), press `2` to set up scene 2 (pendulum), press `3` to set up scene 3 (the unraveling spiral). Use key `h` to toggle between the first (head) ball being locked in place and free. Use key `t` to do the same for the last (tail) ball. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye. Pressing the space bar will pause and then single-step the simulation. Press `p` to resume (or to pause again).

To facilitate experimentation two Boolean variables have been provided, on CPU code `c.opt_test1` and `c.opt_test2`, on CUDA code `dc.opt_test1` and `dc.opt_test2`. Their values are shown on the second line of the display after opt_textX. To change their values press `x` or `X`.

Pressing `a` will switch the method used to run the physics (time step) routine, the method appears near the upper left (near Physics On:). Initially it will show GPU Sol, indicating the code that should be modified for this problem. The other two values are CPU and GPU Base. The GPU Base code is a copy of the code to be modified. As you work on this assignment you can compare the performance of your code, GPU Sol, to the original code, GPU Base, by switching between the two. If the CPU code was compiled without optimization then unoptimized will flash when running the CPU code. Since we are not comparing to the CPU in this problem, the CPU code can be left unoptimized.

A part of the solution involves choosing good values for the grid size (number of blocks) and block size (number of threads per block). These can be modified from the UI by pressing the `TAB` until Grid Size or Block Size appears, and then by using the `+` and `-` keys to adjust them. Gravity can also be adjusted, setting it to a low value provides something like slow motion.

The performance of the physics code is shown on the second line to the left of Time Per Time Step, it is the amount of time needed to compute a single simulation time step. The goal, of course, is to minimize this value. The first line shows other performance numbers. FPS shows the frame rate in frames per second. XF indicates the rate at which frames are updated by the code relative to how fast the device displays frames. A value of 1 is optimal. Under some circumstances the class library code may not be able to synchronize with the hardware, in which case it will update at 30 FPS. GPU.GL shows the amount of time spent on the GPU running OpenGL code per frame, it is shown in milliseconds and as a percent of time. What we do in this assignment should not directly affect this number. GPU.CU shows the amount of time running CUDA code per frame (and as a percent of total time). **Do not** use this number to compare the performance of your code

because its value depends on the number of time steps performed. CPU shows the amount of time per frame spent by the CPU on rendering. It does not include physics. Steps / s shows the number of simulation time steps performed per second. This number should be about 10000, if it's lower then the physics code is too slow to keep up. This will initially be the case for the GPU sol code, but when the assignment is finished it should be up to around 10000. When Steps / s is less than 10000 the simulation will appear to be running slowly (like slow motion) and motion may be jerky. Physics shows the amount of time spent computing physics, whether on the CPU or GPU.

When solving this assignment focus on Time Per Time Step, it can always be used to compare the performance of the different code versions (CPU, GPU Base, and GPU Sol).

The first place to look for CUDA documentation is the CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, start with Chapter 2, Programming Model, it provides a quick overview of CUDA concepts. See also Section 3.2. CUDA C is summarized in Appendix B. The link above takes you to a place providing documentation for the entire CUDA Toolkit. A complete list of CUDA runtime functions (the ones we call from CPU code) can be found in the CUDA Runtime API manual. The Programming Guide describes CUDA C, but a complete list of library functions callable from CUDA C can be found in the CUDA Math API manual.

For this assignment the only CUDA runtime calls needed are for allocating memory, and examples of those can be found in the assignment code.

In this assignment the GPU Sol kernel and related code will be modified to improve performance. Here is a summary of how the code works.

The CUDA code is located in file `hw3-sol-cuda.cu`. It consists of four kernels. At the beginning of every frame kernel `data_pack` is called. It initially does nothing but should be modified to *pack* some of the ball data. Kernel `time_step_gpu_sol_part_1` updates the balls' velocities, it is launched once per time step. (There are many time steps per frame.) Kernel `time_step_gpu_sol_part_2` updates positions (and also velocities for platform collision), it is also called once per time step. Kernel `data_unpack` is called once per frame, after the other kernels, it initially does nothing but should be modified to *unpack* data back into the `Ball` structure. These kernels are launched from the routine `frame_callback` in file `hw3.cc`.

In kernels `time_step_gpu_sol_part_1` and `time_step_gpu_sol_part_2` each thread handles one ball. There are `c.chain_length` balls, which is set to 256. As we should know, 256 is not a large number of threads for a GPU, and so we should expect the GPUs hardware to be underutilized.

The block and grid sizes are initially set to arbitrary values and these can be adjusted by using the UI, initial values can be changed by modifying routine `World::init` in `hw3.cc`.

Several variables are used by both GPU and CPU code. For convenience, they have been placed in structure `CPU_GPU_Common` (see file `hw3.cuh`), this structure is instantiated in `World`, as `c`, and is a CUDA constant, named `dc`. One member is `chain_length`, which is the number of balls. It can be accessed on the CPU as `c.chain_length` and on the GPU as `dc.chain_length`. Members may be added to this structure. The structure is copied to the GPU by routine `data_cpu_to_gpu_constants`, which is called at initialization and whenever a key is pressed.

CUDA runtime API functions (the cuda functions called by CPU code) are wrapped by a macro named `CE`. This checks the return code of the CUDA function for errors. If there is an error it will end execution with a message.

The `Makefile` is set to compile the code without CPU optimization but with CUDA optimization. If you would like to use the cuda debugger (`cuda-gdb`), you will need to modify `Makefile`. Add a `-G` to the `OPT_FLAGS` variable. (You can uncomment a line with this flag added.) Then re-build (press `F9` or type make from the command line). The code will run excruciatingly slowly, so make sure your program reaches the problematic part quickly.

Adding `-O3` to `OPT_FLAGS` will turn on CPU optimization.

In the wee hours of the 21st tnphe TA-bot will copy the following files: `hw3-sol-cuda.cu`, `hw3.cc`, and `hw3.cuh`. Limit changes to these files. If code in another file needs to be modified, contact Dr. Koppelman.

You can place text answers to questions in file `hw3-sol-cuda.cu`, either in comments or defined out. You can also E-mail or turn in paper copies.

Problem 1: Using the UI modify the grid and block sizes and observe the impact on performance.

(*a*) Run the GPU Base code and manually set the grid and block sizes to what you would expect to be good choices. Note the performance. Try varying these to get worse or better performance. Note which values are better, and examples of ones which are worse. Note how sensitive the program is. Does it do just as well even with fewer blocks?

(*b*) Modify the code (in routine `World::init`) so that it starts off with a good configuration. Look for comments "Helpful information" and "Put Solution Here" in this routine. (The solution to this part will have to be changed to reflect modifications made due to the solution to Problem 3. If Problem 3 isn't solved this part can still get full credit. If you prefer don't work on this part until after Problem 3.)

Problem 2: The CUDA code accesses the same `Ball` structure as the CPU code, which, though convenient, is not GPU-friendly because memory requests will only partially be used. Fix it. *Spoiler: It won't help performance.*

(*a*) Modify the code to use a GPU-friendly organization for `Ball::position`.

For your convenience, a variable is already present for the new storage for ball positions, `c.d_pos` (CPU name) and `dc.d_pos` (GPU name).

- You'll need to allocate device storage for the positions, do that in routine `World::init`. Device storage for `balls` is already allocated in that routine, use that code as an example.
- Modify kernel `data_pack` to reorganize the data for the GPU (reading from the `Ball` structure and writing to your new array).
- Modify `time_step_gpu_sol_part_1` and `time_step_gpu_sol_part_2` so that they read and write positions from and to your new array.
- Modify `data_unpack` so that it puts data back into the `Ball` structures.
- Optional: include the ball radius in the w component of position and use it where needed.

(*b*) Measure the performance change by comparing GPU Sol (the code you modified) to GPU base. Explain why the change should not have made much of a difference in this code. (This can change depending on how the next problem is solved.) To answer this question look at the part of the code that's consuming the most execution time. It's only a few lines.

Problem 3: This is the problem where performance will be improved by a substantial amount. The code uses a brute-force $O(n^2)$ method (where n is the number of balls) for resolving interpenetration—that we won't change. The GPU Base and Sol kernels use one thread per ball, so each of these n threads has to check almost n other balls for collisions. We know that $n = 256$ (the number of balls the code uses) threads is not enough to get good utilization on current NVIDIA GPUs.

Modify the code so that the `time_step_gpu_sol_part_1` kernel can use more than n threads. For example, if it were launched with $2n$ threads, a pair of threads would handle one ball and each one would only need to check about $n/2$ other balls for contact. Try to write your code to use an arbitrary number of threads per ball, this can be based on a hard-coded constant (`const int thds_per_ball=4;`) or the code might compute this based on the total number of threads (this is what the spring demo does).

Be sure to update the launch configuration so that it chooses a good number of configuration for your new kernel. (If you like, you can provide a separate configuration for `time_step_gpu_sol_part_1` and the other kernels.)

The spring demo code solves a similar problem, but **unlike the spring demo** the `time_step_gpu_sol_part_1` kernel must compute both near neighbor forces and resolve penetration.