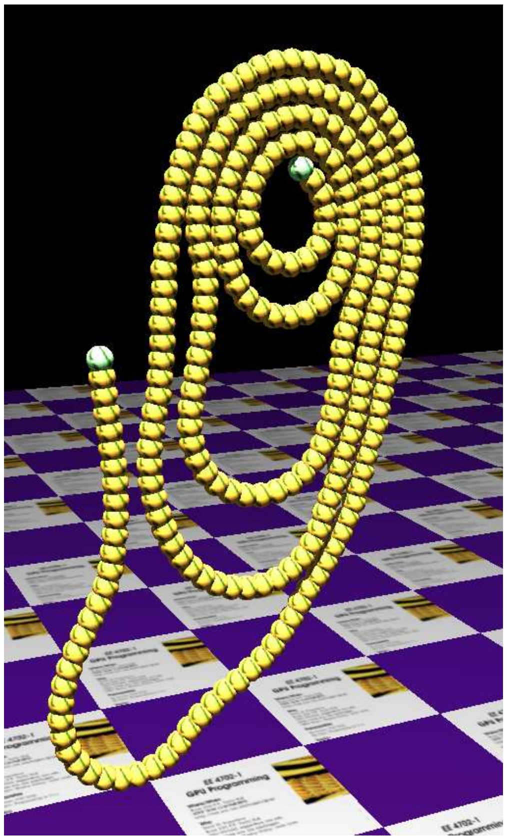
LSU EE 4702-1 Homework 3 Solution Due: 20 November 2013



The solution has been checked into the repository, use `svn update` to retrieve it. Running `make` will build two executables, `hw3`, the unsolved version, and `hw3-sol` which includes the solution. The solution itself is in files `hw3-sol-cuda-sol.cu` and `hw3-sol.cc` (there is no `hw3-sol.cuh`).

In the solution pressing `x` will turn contact pair computation on and off, the results should be dramatic. Pressing `X` will turn the use of structures partially on and off. The effect should be small. See the solution for details.

The solution that follows includes excerpts from the code in the repository, sometimes shortened for clarity. The files can be accessed directly from the repository, preferably using an svn client, at `https://svn.ece.lsu.edu/svn/gp/hw/hw3` or after the semester ends at `https://svn.ece.lsu.edu/svn/gp/hw-archive/gpup-2013/hw3`.

HTML versions of the solution code, which may not be updated, are at
`http://www.ece.lsu.edu/koppel/gpup/2013/hw3-sol-cuda-sol.cu`
and `http://www.ece.lsu.edu/koppel/gpup/2013/hw3-sol.cc`.

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_testX. 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?

For the code to run correctly it needs at least 256 threads. Reducing the block and grid sizes so that their product is less than 256 will result in incorrect execution.

Generally, the grid size should be a multiple of the number of multiprocessors and the block size should be as large as possible, at least 256. Looking at the beginning of the `part_1` and `part_2` kernels we can see that no matter how many threads are launched only 256 (or chain length) will be active. Therefore changing the total number of threads will have no effect.

Changing the grid size, however, will affect where the threads run. For example, if the grid size is 1 and the block size is 256 all the threads will run on one multiprocessor. If the grid size is 2 and the block size is 128, the threads will be spread across two multiprocessors. But, if the grid size is 2 and the block size is 256 the threads will be on one multiprocessor (because of the way the code is written, see the code fragment below). Therefore, to have any kind of an impact when adjusting the grid size the block size would have to be chosen so that the product is 256. (Any lower and the code would not run correctly, any higher and the threads doing work would not use all of the multiprocessors.)

Still, choosing grid and block sizes this way results in little change in performance. That is because 256 threads is barely enough to utilize the resources on one multiprocessor, so splitting the threads to multiple multiprocessors does not help much.

```
__global__ void
time_step_gpu_base_part_1(float delta_t)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int bi = tid; // Ball Index

  if ( bi >= dc.chain_length ) return;
```

(*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.)

The solution to this part is based on the solution to Problem 3. The `part_1` kernel for the solution to Problem 3 will keep as many threads as busy as possible, up to the maximum of $256^2 = 65536$ for the default chain length of 256. Since it is possible to keep that many threads busy, here we will launch as many threads as possible in each block, and set the grid size (the number of blocks) equal to the number of processors.

Looking at the code towards the end of `World::init` we see a variable that holds the number of multiprocessors with a comment encouraging us to use it:

```
  // Helpful information for solution.
  //
  const int number_of_multiprocessors = cuda_prop.multiProcessorCount;

  Kernel_Info *ki_part_1 = NULL;
  for ( int i=0; i<gpu_info.num_kernels; i++ )
    if ( pString(gpu_info.ki[i].name) == "time_step_gpu_sol_part_1" )
      ki_part_1 = &gpu_info.ki[i];

  const int max_threads_part_1 =
    ki_part_1 ? ki_part_1->cfa.maxThreadsPerBlock : 0;
```

Notice that there is also a variable that gives the maximum number of threads per block possible when launching the `part_1` kernel. We'll use that to set the block size:

```
/// SOLUTION

// Set the grid size to the number of multiprocessors.
//
opt_grid_size = number_of_multiprocessors;

// Set the block size to the maximum number of threads possible for
// the part_1 kernel.
//
opt_block_size = max_threads_part_1;
```

The block and grid size apply to all kernels that are launched but we chose the block and grid size just for the `part_1` kernel that we wrote for Problem 3. That means the block and grid size for the other kernels are larger than they have to be. Because those other kernels only need 256 threads, it doesn't matter if they use one multiprocessor or several (as we saw in part (a)). So rather than have separate block and grid size variables for the `part_2`, `pack`, and `unpack` kernels, we'll just launch them with non-optimal sizes that anyway won't make a difference.

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.

The line below in `hw3.cc` allocates the storage:

```
// Allocate storage for the "flat" position array.
//
CE( cudaMalloc( &c.d_pos, c.chain_length*sizeof(c.d_pos[0])) );
```

The pack and unpack routines contain the following lines:

```
// data_pack added lines:
Ball* const ball = &dc.d_balls[bi];
dc.d_pos[bi] = ball->position;
dc.d_pos[bi].w = ball->radius;

// data_unpack added lines:
```

```
Ball* const ball = &dc.d_balls[bi];
ball->position = dc.d_pos[bi];
ball->position.w = 1;
```

Code that had been reading `ball->position` must now be changed to read `d_pos`. Here are some sample changed lines:

```
// In part_1
// Original code:
const float4 ball_position = ball->position;
// Modified code:
const float4 ball_position = dc.d_pos[bi];

// In part_2
// Original code:
ball->position += ball->velocity * delta_t;
// Modified code:
dc.d_pos[bi] += ball->velocity * delta_t;
```

(*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.

In the solution checked in to the repository a user can switch between the two methods of accessing positions by pressing `X` (upper-case). If the green text to the right of `opt_testX` shows `01` then the contact pair (penetration) code accesses the position from the structure, if the text shows `00` then its using the array `d_pos`. (The lower-case `x` turns the contact code on and off.)

The difference in performance is very small. On a Kepler K20c the time per time step is $30.7\,\mu$s when accessing the array and $31.4\,\mu$s when accessing the structure.

The explanation for this small performance difference can be seen in the part of the code consuming most of the execution time, the contact pair loop. Here is an excerpt of the code in the contact pair loop (the original one):

```
for ( int j=0; j<dc.chain_length; j++ ) if ( abs(j-bi)>1 ) {
      Ball* const ball2 = &dc.d_balls[j];
      pNorm ball_to_2 = mn(ball_position,ball2->position);
      const float rsum = ball_radius + ball2->radius;
      if ( rsum * rsum > ball_to_2.mag_sq )
          force -= ( rsum - ball_to_2.magnitude) * 1000 * ball_to_2;
    }
```

In the original code each thread executes that loop for 256 iterations, and one can assume that it is responsible for most of the execution time. So why doesn't replacing `ball2->position` with `dc.d_pos[j]` have much of an impact?

The short answer is: because all threads in a warp are accessing the same element, `j`, and so it doesn't matter if adjacent elements are adjacent in memory.

Here is the more detailed explanation. Global memory is accessed by forming requests. (In class a request was illustrated using a guy with a tray.) A *request* has a starting address and a size. On current generation Kepler systems the request size can be 32, 64, or 128 bytes. On the older Fermi systems (which we have in the lab) the request size normally is just 128 bytes. A request is formed on behalf of the threads in a warp (or half warp on some systems). If all 32 threads are accessing the same location then there will just be a single request. If the 32 threads are each accessing one byte and the memory locations are consecutive (say, they are accessing `char_array[threadIdx.x]`), then there will be one 32-byte request. But, if each thread is accessing one byte but the addresses are very different, (say, they are accessing `char_array[1024*threadIdx.x]`), then 32 requests will be needed. Since the minimum request size is 32 bytes, the total amount of memory accessed will be $32 \times 32 = 1024$ bytes. For Fermi systems the waste is greater, $32 \times 128 = 4096$ bytes.

Remember that within a warp all threads are at the same place in the program and so in the code above when the threads in a warp access `ball2->position` they will all have the same value of `j` and so will be accessing the same location. The amount of data needed will be 16 bytes (four floats which are four bytes each). This can be satisfied with a single request. The request size is exactly the same when the array is used. Because the request sizes (and number of requests) are the same in both cases we don't expect the performance of the contact pair loop to be affected.

Elsewhere in the code the use of an array for positions does have an advantage. Consider the code below from the top of the `part_1` routine:

```
const int bi = threadIdx.x + blockIdx.x * blockDim.x;
Ball* const ball = &dc.d_balls[bi];
const float4 ball_position = ball->position;  // Structure access.
const float4 ball_position = dc.d_pos[bi];    // Array access.
```

Consider consecutive threads in a warp. (Consecutive threads have consecutive values of `threadIdx.x` [say, 0, 1, 2, ...] and the same value of `blockIdx.x` [say, all are equal to 12].) First, consider access to `d_pos`. We are accessing 32 consecutive elements for a total size of $32 \times 16 = 512$ bytes. That can be serviced with four 128-byte requests, and every byte will be used. (Ignoring the `w` component.)

Next, consider the access to `ball->position`. The size of the `Ball` structure is 48 bytes, but the line of code we are interested in only accesses 16 of those bytes. To to retrieve `ball->position` for a warp 32 requests of size 32 will be used, for a total size of 1024 bytes. This will consume twice as much bandwidth for the same amount of data, slowing execution. The waste is even larger when accessing scalar values such as `mass_inv`, there only 4 bytes of each 32-byte request is used.

This waste has a small impact because execution time is dominated by the contact-pair loop in which, as we have seen, structure access is not a problem.

The answer to this question can change depending on how Problem 3 was solved. For the posted solution the answer **does not** change. Consecutive threads still access the same element.

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 block and grid size 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.

The first step is to assign balls to threads. A thread has is assigned a ball for which it is responsible for computing forces, the ball number is given by `bi` in both the original code and the solution. In the solution we want multiple threads to operate on the same ball and we would like all threads operating on the same ball to be in the same block. To realize that we first compute how many balls will be handled by each block (based on the grid size) and then how many threads would be assigned to each ball (based on the block size):

```
const int balls_per_block = ( gridDim.x - 1 + dc.chain_length ) / gridDim.x;
const int thds_per_ball = blockDim.x / balls_per_block;
```

Based on the number of balls per block and the block index we can find the smallest numbered ball handled by any thread in a block:

```
const int bi_block = blockIdx.x * balls_per_block;
```

Based on the code so far, a block will handle balls numbered `bi_block` to `bi_block+balls_per_block-1`. There are two reasonable ways to assign these to threads. The solution below assigns consecutive threads to consecutive balls:

```
const int b_local = threadIdx.x % balls_per_block;
const int bi = bi_block + b_local;
```

An alternative would assign consecutive threads to the same ball, that would be achieved by setting `b_local=threadIdx.x/thds_per_ball;`.

The reason for assigning consecutive threads to consecutive balls is to make the contact pair loop more efficient. That is, if the threads in a warp are all responsible for computing forces for different balls then they can each access *the same "ball2"* for comparison.

The line below computes the index of the first ball accessed in the contact pair loop:

```
const int ai_first = threadIdx.x / balls_per_block;
```

Since there are several threads assigned to a ball we need to pick one to handle gravitational force, in the solution that would be the thread that accesses ball 0 in the contact pair loop:

```
if ( ai_first == 0 ) force += ball->mass * dc.gravity_accel;
```

(If all threads executed the line above it would be as though gravity were `thds_per_ball` times higher. That was a common mistake in submitted solutions.)

This problem specifically mentioned the contact pair loop as a place to use the extra threads but the neighbor loop too can benefit. (Points were not deducted if the neighbor loop was not handled by multiple threads.)

We'll divide the iterations of the neighbor loop amongst the threads using the same variable, `ai_first`, that we use for the contact pair loop. Rather than start at `-3` the contact pair loop starts at `ai_first-3` and rather than incrementing `j` by 1 we increment by `thds_per_ball`:

```
for ( int j = ai_first - 3;  j < 4;  j += thds_per_ball )
```

When the neighbor loop finishes, the forces for each ball will be split across seven (really six) threads. We need to move those forces to the thread responsible for updating velocity, the one with `ai_first==0`.

We do that by first writing the force into a shared array, and then have the `ai_first==0` thread add up those forces one by one. The method below should only be used when the number of items being added are small, in this case 6.

```
__shared__ float3 forces[1024];
forces[threadIdx.x] = force;
__syncthreads();
if ( ai_first == 0 ) {
    const int iters = min(7,thds_per_ball);
    for ( int i=1; i<iters; i++ )
       force += forces[threadIdx.x + i * balls_per_block];
    forces[threadIdx.x] = force;
 }
```

Next is the contact pair loop. In the solution this loop is skipped if the `opt_test1` variable is true, that's to demonstrate how long its taking.

The contact pair loop has two changes. The original loop iterated from `j=0` to the chain length (256). Now we have to adjust the loop so each thread handling a ball accesses a different set of "ball2" balls. The starting point is `ai_first`. Since there are `thds_per_ball` threads handling each ball (each with a different value of `ai_first`), the increment must be `thds_per_ball`.

```
if ( !dc.opt_test1 )
  for ( int j=ai_first; j<dc.chain_length; j += thds_per_ball )
    if ( abs(j-bi)>1 )
```

The other change to the contact pair loop is the method used to update the force variable. In the neighbor loop each thread updated its own copy of `force` and these were combined after the loop completed. For the contact pair loop we'll use a different approach. Rather than combine the forces after the contact loop exits, we will have each thread for a ball add the force to the same location. Variable `b_local` is the same for all threads handling the same ball (within a block). So that will be used as the index into the `forces` shared array.

We can't just use the code `forces[b_local]+=sep_f` because multiple threads with the same value of `b_local` may execute that line at the same time (which would result in an incorrect sum). An easy but costly way to avoid the problem is by using an atomic add. Remember that an atomic add will correctly add something to a memory location (shared or global) by first locking the location, performing the operation, and then unlocking the location. The locking and unlocking slows things down, especially when multiple threads are trying to access the same location.

The reason that an atomic add is acceptable here is because we expect only a few balls to be in contact, and so the atomic adds will be reached for only a small fraction of the iterations. Here is the simplified code:

```
          const float rsum = ball_radius + dc.d_pos[j].w );
          if ( rsum * rsum > ball_to_2.mag_sq ) {
              float3 sep_f = ( rsum - ball_to_2.magnitude) * 1000 * ball_to_2;
              atomicAdd(&forces[b_local].x,sep_f.x);
              atomicAdd(&forces[b_local].y,sep_f.y);
              atomicAdd(&forces[b_local].z,sep_f.z);
            }
```

Finally, the code needs to use the computed force to update the velocity. Since only one thread per ball updates the velocity the others can return. But first, a `syncthreads` is executed so that the thread assigned to update the velocity does not start working until the other threads have finished updating the force.

A common mistake was to allow multiple threads to update the same ball's velocity. This would result in incorrect velocities. The solution avoids it by having each ball's velocity written by just one thread, the one with `ai_first`==0.

```
  __syncthreads();
  if ( ai_first ) return;
  float3 velocity = ball_velocity;
  velocity += delta_t * ball->mass_inv * forces[b_local];
  const double fs = powf(1+dc.opt_air_resistance,-delta_t);
  velocity *= fs;
  ball->velocity = velocity;
```