Chapter 25
GPU Sequential Dependencies

► Part I. Preliminaries
► Part II. Tightly Coupled Multicore
► Part III. Loosely Coupled Cluster
▼ Part IV. GPU Acceleration
    Chapter 22. GPU Massively Parallel
    Chapter 23. GPU Parallel Reduction
    Chapter 24. Multi-GPU Programming
    Chapter 25. GPU Sequential Dependencies
    Chapter 26. Objects on the GPU
► Part V. Big Data
Recall the \(N\)-body zombie program from Chapter 8. The sequential version was based on a triply-nested loop, as shown in the pseudocode below. \((x_i, y_i)\) is the current position of zombie \(i\); \((v_{xi}, v_{yi})\) is the net velocity of zombie \(i\); \((v_{xij}, v_{yij})\) is the velocity of zombie \(i\) relative to zombie \(j\); and \((next_{xi}, next_{yi})\) is the next position of zombie \(i\). The first loop goes over the time steps; the second loop goes over the zombies and calculates each zombie’s next position; the third loop also goes over the zombies and accumulates the velocity relative to every other zombie. \(\Delta\) is the total absolute distance all the zombies moved during the time step; when this falls below a threshold \(\varepsilon\), the zombies have reached equilibrium.

Initialize zombie positions
Repeat: (time step loop)
For \(i = 0\) to \(N - 1\):
\((v_{xi}, v_{yi}) \leftarrow (0, 0)\)
For \(j = 0\) to \(N - 1, j \neq i\):
Compute \((v_{xij}, v_{yij})\) using equations (7.1), (7.2), and (7.3)
\((v_{xi}, v_{yi}) \leftarrow (v_{xi} + v_{xij}, v_{yi} + v_{yij})\)
\((next_{xi}, next_{yi}) \leftarrow (x_i + v_{xi} \cdot dt, y_i + v_{yi} \cdot dt)\)
Replace current zombie positions with next zombie positions
If \(\Delta < \varepsilon\):
Exit time step loop

To make the multicore parallel version of the zombie program, I noted that the time step loop has sequential dependencies; I cannot calculate the next time step until I’ve finished calculating the previous time step; so the outer loop had to remain a regular, non-parallel loop. But the second loop does not have sequential dependencies; the next position of one zombie does not depend on the next position of any other zombie; so the second loop could become a parallel loop, calculating all the zombies’ next positions in parallel:

Initialize zombie positions
Repeat: (time step loop)
\textbf{Parallel for} \(i = 0\) to \(N - 1\):
\((v_{xi}, v_{yi}) \leftarrow (0, 0)\)
For \(j = 0\) to \(N - 1, j \neq i\):
Compute \((v_{xij}, v_{yij})\) using equations (7.1), (7.2), and (7.3)
\((v_{xi}, v_{yi}) \leftarrow (v_{xi} + v_{xij}, v_{yi} + v_{yij})\)
\((next_{xi}, next_{yi}) \leftarrow (x_i + v_{xi} \cdot dt, y_i + v_{yi} \cdot dt)\)
Replace current zombie positions with next zombie positions
If \(\Delta < \varepsilon\):
Exit time step loop
Now let’s make a GPU parallel version of the zombie program. The time step loop still has to remain a regular, non-parallel loop. But the second loop can be done in parallel, so I will do that part on the GPU. The GPU computational kernel calculates the zombies’ next positions and $\Delta$, given the zombies’ current positions, for one time step. The time step loop runs on the CPU and executes the kernel repeatedly, updates the positions, and checks for convergence:

Initialize zombie positions
Repeat: (time step loop)
  **Execute GPU computational kernel**
  Replace current zombie positions with next zombie positions
  If $\Delta < \varepsilon$:
    Exit time step loop

The GPU kernel has to do this, expressed as sequential pseudocode:

For $i = 0$ to $N - 1$:
  $(vx_i, vy_i) \leftarrow (0, 0)$
For $j = 0$ to $N - 1, j \neq i$:
  Compute $(vx_{ij}, vy_{ij})$ using equations (7.1), (7.2), and (7.3)
  $(vx_i, vy_i) \leftarrow (vx_i + vx_{ij}, vy_i + vy_{ij})$
  $(nextx_i, nexty_i) \leftarrow (x_i + vx_i \cdot dt, y_i + vy_i \cdot dt)$

How shall I parallelize this pseudocode to run on the GPU? Recall that the GPU kernel has two levels of parallelism: the kernel can run multiple blocks in parallel on the GPU’s multiprocessors, and each block can run multiple threads in parallel on the multiprocessor’s cores. So a natural way to partition the computation is to parallelize the outer loop iterations across the blocks, and to parallelize the inner loop iterations across the threads in a block. The kernel will have a one-dimensional grid of one-dimensional blocks. Each block will compute the next position of one zombie. The threads in the block will compute the zombie’s velocity relative to every other zombie. These relative velocities must be added together to get the zombie’s net velocity; this calls for a parallel sum-reduce within the block, like the one in the GPU parallel $\pi$ estimating program in Chapter 24. The kernel must also calculate $\Delta$, the total absolute distance all the zombies moved. Each block can calculate a partial $\Delta$ for its own zombie. The blocks’ $\Delta$ values must then all be added together; this is done with atomic operations on a variable in global memory, again like the GPU parallel $\pi$ estimating program.

Turning to the GPU kernel code (Listing 25.1), line 2 declares that there will be 256 threads per block. The kernel’s inner loop iterations will be partitioned on these many threads. Why 256 threads? Why not have 1024 threads, the maximum allowed in a compute-capability-2.0 GPU? I discuss this question in the “Under the Hood” section below.
Line 5 declares the devDelta variable in the GPU’s global memory. The kernel will store the total $\Delta$ in this variable. Because it is located in global memory, all threads in all blocks can access it.

Lines 8–9 declare arrays of X and Y velocity values, one element for each thread in the block, located in the block’s fast shared memory. These arrays will be used in the parallel sum-reduce within the block to compute the block’s zombie’s net velocity. (There is one element for each thread, not one element for each zombie.)

Lines 12–27 define a subroutine to do atomic addition on a variable of type double. The subroutine adds the given value to the given variable v and stores the result back in v. Furthermore, the addition is done atomically, so multiple threads adding values to the variable will not interfere with each other. I had to write my own subroutine because a compute-capability-2.0 GPU does not have an atomic add operation for type double. See the “Under the Hood” section below for an explanation of how the atomicAdd subroutine works.

The timeStep kernel function begins on line 34. The first four arguments are pointers to arrays of type double for the zombies’ current X and Y positions $xpos$ and $ypos$ and the next X and Y positions $xnext$ and $ynext$. These have to be passed as arguments, and cannot be declared as variables like devDelta, because the number of zombies is not known at compile time; rather, the user specifies the number of zombies when running the program. Indeed, the fifth argument is the number of zombies $N$, which is the length of the arrays. The sixth and seventh arguments $G$ and $L$ are part of the formula for the zombie’s relative velocity. The eighth argument $dt$ is the time step size.

When the kernel function is called, the $xpos$ and $ypos$ arrays contain the kernel’s inputs, namely all the zombies’ current positions for the current time step. The $G$, $L$, and $dt$ arguments are also inputs. The $xnext$ and $ynext$ arrays will hold the kernel’s outputs, namely all the zombie’s next positions after the current time step. The devDelta variable is also an output.

The kernel function code proper begins by initializing several variables on lines 44–49. The variable $i$ is the index of this block’s zombie; because each block is calculating the next position of one particular zombie, $i$ is just the index of the block within the grid, from 0 to $N – 1$. The variable $j$ is the rank of the thread within the block, from 0 to $NT – 1$ (0 to 255).

When the kernel function is executing, the outer loop iterations over the zombies have already been partitioned—each outer loop iteration is being calculated by one of the blocks in the grid, in parallel with all the other blocks. Now, in the kernel function, the inner loop iterations have to be partitioned among the threads in the block. The for loop starting on line 54 partitions the inner loop iterations using, in effect, a leapfrog schedule, similar to the one in the GPU parallel $\pi$ estimating program. The loop index $k$ starts at
// Number of threads per block.
#define NT 256

// Variables in global memory.
__device__ double devDelta;

// Per-thread variables in shared memory.
__shared__ double shrVx [NT];
__shared__ double shrVy [NT];

// Atomically set double variable v to the sum of itself and value.
__device__ void atomicAdd
    (double *v,
     double value)
{
    double oldval, newval;
    do
    {
        oldval = *v;
        newval = oldval + value;
    }
    while (atomicCAS
              ((unsigned long long int *)v,
               __double_as_longlong (oldval),
               __double_as_longlong (newval))
                  != __double_as_longlong (oldval));
}

// Device kernel to update zombie positions after one time step.
// Called with a one-dimensional grid of one-dimensional blocks,
// N blocks, NT threads per block. N = number of zombies. Each block
// updates one zombie. Each thread within a block computes the
// velocity with respect to one other zombie.
extern "C" __global__ void timeStep
    (double *xpos,
     double *ypos,
     double *xnext,
     double *ynext,
     int N,
     double G,
     double L,
     double dt)
{
    int i = blockIdx.x;   // Index of this block's zombie
    int j = threadIdx.x;  // Index of this thread within block
    double xpos_i = xpos[i]; // This zombie's current X position
    double ypos_i = ypos[i]; // This zombie's current X position
    double vx = 0.0;       // This zombie's X velocity
    double vy = 0.0;       // This zombie's Y velocity
    int k;
    double dx, dy, d, v;

    // Compute and accumulate velocity w.r.t. every other zombie.
    for (k = j; k < N; k += NT)
    {
        if (k == i) continue;
        dx = xpos[k] - xpos_i;
        dy = ypos[k] - ypos_i;

        // Other zombie's position
        double x = xpos[k];
        double y = ypos[k];

        // Compute velocity
        double dx_1 = x - xpos_i;
        double dy_1 = y - ypos_i;

        // Update velocities
        vx += dx_1 / L;
        vy += dy_1 / L;

        // Apply force
        double force = -G / (sqrt(dx_1*dx_1 + dy_1*dy_1) + 1e-10);
        dx += force * dt;
        dy += force * dt;
    }

    // Update positions
    xpos_i += vx * dt;
    ypos_i += vy * dt;
    }

Listing 25.1. ZombieGpu.cu (part 1)
the thread’s own rank and increases by the number of threads in the block (256). For example, suppose there are 1000 zombies. In thread rank 0, the loop index \( k \) will be 0, 256, 512, 768; in thread rank 1, the loop index \( k \) will be 1, 257, 513, 769; and so on. In this manner, all inner loop indexes from 0 to 999 will be calculated by some thread in the block. Each thread will perform the same, or nearly the same, number of inner loop iterations; and because each inner loop iteration takes the same amount of time, the load is balanced. The inner loop body calculates the relative velocity between zombie \( i \) (the block’s zombie) and zombie \( k \) (one of the other zombies) and accumulates the relative velocity into the thread’s local variables \( v_x \) and \( v_y \). Because each thread is updating its own local variables, the threads do not need to synchronize with each other at this point.

Once the loop finishes, each thread in the block has calculated a partial net velocity. These partial net velocities must be added together to get the zombie’s total net velocity. This is accomplished using a shared memory parallel reduction tree (lines 66–77). The reduction tree code is the same as in the GPU parallel \( \pi \) estimating program, except that we are reducing two quantities at the same time, namely \( v_x \) and \( v_y \). When the reduction finishes, \( shrVx[0] \) and \( shrvy[0] \) end up holding the X and Y components of the zombie’s net velocity.

The kernel function can now calculate the zombie’s next position, as well as the zombie’s partial \( \Delta \). These calculations must be done by a single thread; so thread rank 0 executes lines 82–93, and the other threads do nothing. Lines 83–84 retrieve the zombie’s net velocity from shared memory; lines 87–88 calculate the zombie’s net change in position; lines 89–90 calculate the zombie’s next position and store it in the \( x_{next} \) and \( y_{next} \) arrays in GPU global memory, at the zombie’s own index. Because the thread 0s in all the blocks are updating different elements in \( x_{next} \) and \( y_{next} \), the threads do not need to synchronize with each other at this point. Lastly, line 93 adds the zombie’s absolute change in position to the devDelta variable in GPU global memory, using the \texttt{atomicAdd} subroutine defined earlier to synchronize the threads.

When the GPU kernel finishes, its outputs have been stored in GPU global memory. The \( x_{next} \) and \( y_{next} \) arrays contain all the zombies’ next positions after the current time step. The \( \text{devDe}lt\text{a} \) variable contains the total absolute change in position of all the zombies for the current time step.

Having written the C kernel, I’m now able to write the Java main program, class 
\texttt{ZombieGpu} (Listing 25.2). The program declares variables to hold the zombies’ current X and Y positions (lines 25–26) and their next X and Y positions (lines 29–30). These variables are of type \texttt{GpuDoubleArray}, which provide arrays of type \texttt{double} allocated in the GPU and mirrored in the CPU. The GPU kernel takes the \texttt{xpos} and \texttt{ypos} arrays as inputs, and calculates the \( x_{next} \) and \( y_{next} \) arrays as outputs. The main program also de-
\[ d = \sqrt{dx \cdot dx + dy \cdot dy}; \]
\[ v = G \cdot \exp(-d/L) - \exp(-d); \]
\[ vx += v \cdot dx/d; \]
\[ vy += v \cdot dy/d; \]
\}

// Compute net velocity via shared memory parallel reduction.
shrVx[j] = vx;
shrVy[j] = vy;
__syncthreads();
for (k = NT/2; k > 0; k >>= 1)
{
    if (j < k)
    {
        shrVx[j] += shrVx[j+k];
        shrVy[j] += shrVy[j+k];
    }
    __syncthreads();
}

// Single threaded section.
if (j == 0)
{
    // Get net velocity.
    vx = shrVx[0];
    vy = shrVy[0];

    // Move zombie in the direction of its velocity.
    dx = vx*dt;
    dy = vy*dt;
    xnext[i] = xpos_i + dx;
    ynext[i] = ypos_i + dy;

    // Accumulate position delta.
    atomicAdd (&devDelta, abs(dx) + abs(dy));
}

\textbf{Listing 25.1.} ZombieGpu.cu (part 2)

\begin{verbatim}
package edu.rit.gpu.example;
import edu.rit.gpu.CacheConfig;
import edu.rit.gpu.Gpu;
import edu.rit.gpu.GpuDoubleArray;
import edu.rit.gpu.GpuDoubleVbl;
import edu.rit.gpu.Kernel;
import edu.rit.pj2.Task;
import edu.rit.util.Random;
public class ZombieGpu
    extends Task
{

\textbf{Listing 25.2.} ZombieGpu.java (part 1)
clares a variable to hold $\Delta$ (line 33), which is another output calculated by the kernel. This variable is of type GpuDoubleVbl, which provides a single value of type double located in the GPU and mirrored in the CPU.

Next comes the kernel interface (lines 36–48). The interface declares the kernel method `timeStep()` and its arguments, which are the same as the kernel function’s arguments in the GPU code. However, the `xpos`, `ypos`, `xnext`, and `ynext` arguments are declared to be Java type GpuDoubleArray rather than C type double*—that is, the same type as the corresponding variables on lines 25–30. (This is Java, not C; you can’t do pointers in Java.)

The main program proper begins on line 51. After parsing the command line arguments, initializing the GPU object, verifying the compute capability, and loading the compiled kernel module, the program creates the four objects `xpos`, `ypos`, `xnext`, and `ynext` (lines 74–77). Each object contains, in its item field, an array of $N$ elements of type double. Each array is also allocated storage for $N$ elements of type double in the GPU’s global memory. The program also creates the `delta` object (line 78). This object’s item field, a value of type double, mirrors the `devDelta` variable in the GPU’s global memory, which was declared in the kernel code.

Next the main program sets up the computational kernel (lines 81–84). The kernel is configured with a one-dimensional grid of one-dimensional blocks, with $N$ blocks (one for each zombie) and 256 threads per block (as assumed by the kernel code). The kernel is also configured with more L1 cache and less shared memory. On my Nvidia Tesla C2075 GPUs, this configures each multiprocessor with 48 kilobytes of L1 cache and 16 kilobytes of shared memory. The shared memory holds the `shrVx` and `shrVy` arrays; each array has 256 elements of type double; a double value occupies 8 bytes; so the two arrays occupy 4 kilobytes; this fits within the 16 kilobyte limit. I want the L1 cache to be as large as possible because the threads will be reading the zombies’ current positions from the `xpos` and `ypos` arrays in global memory, and I want these cached in L1 to speed up the memory accesses. The 48-kilobyte L1 cache can hold up to 3,072 zombies’ X and Y coordinates without experiencing cache misses.

The program initializes the zombies’ $(x, y)$ coordinates to random values on the CPU (lines 88–93) and uploads these to the GPU (lines 94–95). The program also prints a snapshot of the zombies’ initial positions (line 99).

The program now commences the outermost loop over the time steps (line 102). At each time step, the program reinitializes $\Delta$ to 0 and uploads it to the GPU (lines 105–106). The program then calls the `timeStep()` kernel method (line 107), which executes the computational kernel. When the kernel method returns, the `xnext` and `ynext` arrays on the GPU contain the zombies’ calculated next positions, and the `devDelta` variable on the GPU contains $\Delta$. To go on to the next time step, the zombies’ next positions need to become the zombies’ new current positions. There’s no need to copy the elements
// Command line arguments.
long seed;
int N;
double W;
double G;
double L;
double dt;
double eps;
int steps;
int snap;

// Current body positions.
GpuDoubleArray xpos;
GpuDoubleArray ypos;

// Next body positions.
GpuDoubleArray xnext;
GpuDoubleArray ynext;

// For detecting convergence.
GpuDoubleVbl delta;

// Kernel function interface.
private static interface ZombieKernel
extends Kernel
{
    public void timeStep
        (GpuDoubleArray xpos,
         GpuDoubleArray ypos,
         GpuDoubleArray xnext,
         GpuDoubleArray ynext,
         int N,
         double G,
         double L,
         double dt);
}

// Task main program.
public void main
(String[] args)
throws Exception
{
    // Parse command line arguments.
    if (args.length != 9) usage();
    seed = Long.parseLong (args[0]);
    N = Integer.parseInt (args[1]);
    W = Double.parseDouble (args[2]);
    G = Double.parseDouble (args[3]);
    L = Double.parseDouble (args[4]);
    dt = Double.parseDouble (args[5]);
    eps = Double.parseDouble (args[6]);
    steps = Integer.parseInt (args[7]);
    snap = Integer.parseInt (args[8]);

    // Initialize GPU.
    Gpu gpu = Gpu.gpu();
gpu.ensureComputeCapability (2, 0);
from one array to the other, and there’s not even any need to download the
array elements from the GPU to the CPU. All that’s necessary is to swap the
array object references on the CPU (lines 113–115). The program downloads
Δ from the GPU, checks for convergence, and exits the time step loop if so
(lines 120–123). If the program has not reached convergence, the program
prints out a snapshot of the zombies’ positions every snap time steps, where
snap was specified on the command line (lines 126–127); and the program
repeats the time step loop. One final snapshot takes place when the program
is finished (line 131).

The snapshot() method (lines 135–143) downloads the current zombie
positions into the xpos and ypos objects on the CPU, and prints those. Taking
a snapshot is the only time the zombies’ positions need to be transferred
from the GPU to the CPU. This happens once at the beginning of the pro-
gram, once at the end, and every snap time steps in between.

To compare the CPU’s performance to the GPU’s performance on the
zombie simulation, I ran the ZombieSeq program and the ZombieGpu pro-
gram on the kraken machine, using commands like this:

```
$ java pj2 debug=makespan edu.rit.pj2.example.ZombieSeq \\
 142857 100 5.00 0.5 10 0.00001 0.001 0 0
$ java pj2 debug=makespan edu.rit.gpu.example.ZombieGpu \\
 142857 100 5.00 0.5 10 0.00001 0.001 0 0
```

I ran the programs for various numbers of zombies $N$ and various initial areas
$W$ (the second and third command line arguments). Here are the running
times $T$ in milliseconds I observed, as well as the number of time steps
needed to reach convergence:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$W$</th>
<th>Steps</th>
<th>CPU $T$</th>
<th>GPU $T$</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5.00</td>
<td>15869</td>
<td>12034</td>
<td>1704</td>
<td>7.1</td>
</tr>
<tr>
<td>200</td>
<td>7.07</td>
<td>13043</td>
<td>39568</td>
<td>2187</td>
<td>18.1</td>
</tr>
<tr>
<td>500</td>
<td>11.18</td>
<td>10186</td>
<td>192903</td>
<td>3615</td>
<td>53.4</td>
</tr>
<tr>
<td>1000</td>
<td>15.81</td>
<td>9308</td>
<td>706896</td>
<td>7947</td>
<td>90.0</td>
</tr>
<tr>
<td>2000</td>
<td>22.36</td>
<td>9595</td>
<td>2880873</td>
<td>27286</td>
<td>105.6</td>
</tr>
</tbody>
</table>

For a large enough $N$-body simulation—a problem size for which I might re-
ally want to take advantage of the GPU’s computational power—the GPU is
over 100 times faster than the CPU. Different computations, running on dif-
ferent CPU and GPU hardware, would experience different time ratios. Still,
in my experience GPU programs can be one to two orders of magnitude
faster than equivalent CPU programs.

**Under the Hood**

Here’s how I decided to use $NT = 256$ threads per block in the GPU ker-
nel. After writing the complete program, I ran the program on a typical prob-
// Set up GPU variables.
Module module = gpu.getModule("edu/rit/gpu/example/ZombieGpu.cubin");
xpos = gpu.getDoubleArray(N);
ypos = gpu.getDoubleArray(N);
xnext = gpu.getDoubleArray(N);
ynext = gpu.getDoubleArray(N);
delta = module.getDoubleVbl("devDelta");

// Set up GPU kernel.
ZombieKernel kernel = module.getKernel(ZombieKernel.class);
kernel.setBlockDim(256);
kernel.setGridDim(N);
kernel.setCacheConfig(CacheConfig.CU_FUNC_CACHE_PREFER_L1);

// Set zombies' initial (x,y) coordinates at random in a WxW square region.
Random prng = new Random(seed);
for (int i = 0; i < N; ++i)
{
    xpos.item[i] = prng.nextDouble()*W;
    ypos.item[i] = prng.nextDouble()*W;
}
xpos.hostToDev();
ypos.hostToDev();

// Snapshot all bodies' initial positions.
int t = 0;
snapshot(t);

// Do repeated time steps.
for (;;)
{
    // Do one time step.
    delta.item = 0.0;
    delta.hostToDev();
kernel.timeStep(xpos, ypos, xnext, ynext, N, G, L, dt);
    // Advance to next time step.
    ++t;
    // Update positions.
    GpuDoubleArray tmp;
tmp = xpos; xpos = xnext; xnext = tmp;
tmp = ypos; ypos = ynext; ynext = tmp;
    // Stop when position delta is less than convergence threshold or when the specified number of time steps
    // have occurred.
    delta.devToHost();
    if ((steps == 0 && delta.item < eps) ||
        (steps != 0 && t == steps))
        break;

    // Snapshot all bodies' positions every <snap> time steps.
    if (snap > 0 && (t % snap) == 0)
        snapshot(t);
}

Listing 25.2. ZombieGpu.java (part 3)
lem with $NT$ defined to be different powers of 2, namely 32, 64, 128, 256, 512, and 1024. I picked powers of 2 to simplify the parallel reduction code in the kernel function. I measured the program’s running time for each value of $NT$, and picked the $NT$ value that yielded the smallest running time. Fewer than 256 or more than 256 threads per block turned out to yield larger running times. There’s no simple explanation for this. In my experience, rather than trying to predict the optimum value of $NT$ ahead of time, it’s easier to measure the program’s actual performance and adjust $NT$ accordingly.

The ZombieGpu kernel included the `atomicAdd()` function for a variable of type `double` because CUDA compute capability 2.0 does not have that function built in, so I had to write it myself. I used a function that is built in, called `atomicCAS()`, which stands for “atomic compare and swap.”

The atomic CAS operation takes three arguments: a variable, an old value, and a new value. The atomic CAS operation also uses the variable’s current value. The operation does the following: If the variable’s current value equals the old value, then set the variable’s value to the new value and return the variable’s previous value; otherwise leave the variable’s value unchanged and return the variable’s current value. Furthermore, the operation is done atomically; when one thread is in the middle of an atomic CAS on a variable, it will not be interrupted by any other thread trying to do an atomic CAS on that variable. Stated another way, atomic CAS compares the variable to the old value, and if they are the same, swaps the variable with the new value. Atomic CAS is typically implemented in hardware.

Here is pseudocode for atomically updating a variable using the atomic CAS operation:

```
1  Do:
2      old value = current value of the variable
3      new value = value to be stored in the variable
4      While atomicCAS (variable, old value, new value) ≠ old value
```

Step 2 queries the variable’s current value. Step 3 computes the variable’s updated value; usually the updated value depends on the current value. Step 4 tries to atomically set the variable to the new value. The attempt will not succeed, however, if at Step 4 the variable’s value is not the same as the old value retrieved at Step 2. Why might the value not be the same? Because another thread might have changed the variable’s value between the time this thread did Step 2 and the time this thread did Step 4. In that case, the value returned by the atomic CAS will not be the same as the old value the thread is expecting; so the thread stays in the do-while loop, re-queries the variable’s value, recomputes the updated value, and retries the atomic CAS. The thread might stay in this loop for several iterations. When the update succeeds—that is, when the atomic CAS returns the old value the thread is expecting—the thread gets out of the loop and goes on.
// Snapshot all bodies' final positions.
snapshot(t);
}

// Snapshot all bodies' positions.
private void snapshot(int t) {
xpos.devToHost();
ypos.devToHost();
for (int i = 0; i < N; ++ i)
    System.out.printf("%d\t%d\t%g\t%g\n", t, i, xpos.item[i], ypos.item[i]);
}

// Print a usage message and exit.
private static void usage() {
    System.err.println("Usage: java pj2 " +
        "edu.rit.pj2.example.ZombieGpu <seed> <N> <W> <G> <L> " +
        "<dt> <eps> <steps> <snap>");
    System.err.println("<seed> = Random seed");
    System.err.println("<N> = Number of bodies");
    System.err.println("<W> = Region size");
    System.err.println("<G> = Attraction factor");
    System.err.println("<L> = Attraction length scale");
    System.err.println("<dt> = Time step size");
    System.err.println("<eps> = Convergence threshold");
    System.err.println("<steps> = Number of time steps (0 = " +
        "until convergence)");
    System.err.println("<snap> = Snapshot interval (0 = none)");
    throw new IllegalArgumentException();
}

// Specify that this task requires one core.
protected static int coresRequired() {
    return 1;
}

// Specify that this task requires one GPU accelerator.
protected static int gpusRequired() {
    return 1;
}
Using atomic CAS, I can atomically update a variable any way I please. But there’s still a problem: CUDA compute capability 2.0 has atomic CAS operations for integer variables—types int, unsigned int, long long int, and unsigned long long int. CUDA compute capability 2.0 does not have atomic CAS operations for floating point variables—types float and double.

To deal with this problem, I’ll trick the atomic CAS operation into thinking that a value of type double is really a value of type unsigned long long int. I can get away with this because both types occupy the same amount of memory, eight bytes. On line 23, I cast the variable pointer from double* to unsigned long long int*; this doesn’t change the pointer, but it makes it look like the correct type for the atomic CAS. On lines 24–26, I run the double values through the special __double_as_long_long() CUDA function. This special CUDA function does not do a type conversion—it does not convert a floating point value to an integer value, which would alter the value’s bit pattern. Rather, this special CUDA function leaves the value’s bit pattern intact and merely reinterprets that bit pattern as being a long integer, which is the correct type for the atomic CAS. The atomic CAS then manipulates those bit patterns: it compares the variable’s current value’s bit pattern to the old value’s bit pattern, and if they are the same, swaps the variable with the new value’s bit pattern and returns the variable’s previous value’s bit pattern. The result is the same as if the atomic CAS operation had operated on the double values directly.

The ZombieGpu main program mirrors the xpos, ypos, xnext, ynext, and delta variables between the CPU and the GPU. In Chapter 23 I pointed out that mirrored variables are not copied automatically between the CPU and the GPU; rather, you have to call the hostToDev() and devToHost() methods explicitly. Now we can see why. In the zombie program, there’s no need to transfer the xpos and ypos arrays from the GPU to the CPU at every time step. The arrays need to be transferred only when the program is printing a snapshot of the zombies’ positions. Refraining from transferring the arrays at other times reduces the program’s sequential overhead. The xnext and ynext arrays never need to be transferred at all; they are used solely in the kernel. On the other hand, the delta variable does need to be transferred at each time step, so the CPU can check the convergence criterion. But because delta is just a single double value, the overhead is minimal.

Points to Remember

• For a program with sequential dependencies, consider doing the sequential loop in the CPU main program and the parallelizable loop or loops in the GPU kernel.
• The GPU has two levels of parallelism: the blocks within the grid, and the threads within the block. Keep this flexibility in mind when parallelizing a program for the GPU.
• If a section of code in the GPU kernel function must be performed by only one thread in the block, enclose the code section in an if statement that checks whether the thread index is 0.
• When mirroring variables on the CPU and the GPU, transfer the variables from one side to the other only when necessary.
• Determine the kernel’s number of threads per block $NT$ empirically: measure the program’s running time for different $NT$ values, and pick the one that yields the smallest running time.
• Use the $\text{atomicCAS}()$ function to atomically update a variable, if CUDA does not support the desired updating operation.
• Use the $\text{__double_as_long_long}()$ function, and similar functions (see the CUDA documentation), to treat a bit pattern of one type as a bit pattern of another type for purposes of atomic CAS.