Chapter 26
Objects on the GPU

- 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
The GPU parallel programs we’ve studied so far all used primitive types to represent the data being calculated. The outer product program used arrays and matrices of type double. The π estimating programs used counters of type long. The zombie program used arrays of type double for the zombie positions. In the Java main programs, I used instances of classes such as GpuLongVbl, GpuDoubleArray, and GpuDoubleMatrix to mirror the GPU variables on the CPU. Each such instance had an item field of the appropriate Java type, such as long, double[], or double[][][]. I called the hostToDev() and devToHost() methods on those instances to transfer the data between the CPU and the GPU.

For some programs, however, the appropriate type to represent the data is an object, not a primitive type. Consider the zombie program. The program needs to store the zombies’ positions. Viewed abstractly, a zombie’s position is a two-dimensional vector consisting of the X coordinate and the Y coordinate. In an object-oriented language such as Java, the proper way to represent a vector is a class with two fields:

```java
public class Vector {
    public double x;
    public double y;
}
```

I can also define methods to do operations on vectors, such as add two vectors, subtract two vectors, determine the magnitude of a vector, and so on. I can then use the vector class in other data structures, such as an array of vectors holding positions for multiple zombies. Manipulating the positions by calling vector methods is easier to read, easier to maintain, and less defect prone than the non-object-oriented approach I took in the previous chapter, where the zombies’ positions were stored in two separate arrays of X coordinates and Y coordinates.

A non-object-oriented language like C does not have objects. C does, however, have structures. The proper way to represent a vector in C is a structure with two fields:

```c
typedef struct {
    double x;
    double y;
} vector_t;
```

Apart from immaterial differences in syntax, the Java class and the C structure are really the same data type. When I write a GPU parallel program involving vectors, I want to write the Java main program code using the Vector class, and I want to write the C kernel code using the vector_t structure. Then I want to transfer instances of the Vector class from the CPU to the
// Number of threads per block.
#define NT 256

// Structure for a 2-D vector.
typedef struct {
    double x;
    double y;
} vector_t;

// Vector addition; a = b + c. Returns a.
__device__ vector_t *vectorAdd (vector_t *a, vector_t *b, vector_t *c) {
    a->x = b->x + c->x;
    a->y = b->y + c->y;
    return a;
}

// Vector subtraction; a = b - c. Returns a.
__device__ vector_t *vectorSubtract (vector_t *a, vector_t *b, vector_t *c) {
    a->x = b->x - c->x;
    a->y = b->y - c->y;
    return a;
}

// Scalar product; a = a*b. Returns a.
__device__ vector_t *scalarProduct (vector_t *a, double b) {
    a->x *= b;
    a->y *= b;
    return a;
}

// Returns the magnitude of a.
__device__ double vectorMagnitude (vector_t *a) {
    return sqrt (a->x*a->x + a->y*a->y);
}

// Variables in global memory.
__device__ double devDelta;

// Per-thread variables in shared memory.
__shared__ vector_t shrVel [NT];

Listing 26.1. ZombieGpu2.cu (part 1)
GPU and have them end up as vector_t structures on the GPU; and vice versa when transferring in the other direction.

The Parallel Java 2 Library has exactly this capability. To illustrate it, I’m going to develop a second version of the GPU parallel zombie program, this time using vector objects and structures.

Listing 26.1 is the C code for the GPU kernel. It begins by declaring a vector_t structure for a two-dimensional vector (lines 5–10). Because C is not an object oriented programming language, I can’t define methods on this structure. However, I can define functions that act like methods. To define a “method,” I simply define a function whose first argument, a, is a pointer to the structure I want the function to manipulate. I can then access the structure’s fields using syntax like a->x and a->y. Lines 12–49 define several such vector methods to do vector addition, vector subtraction, scalar multiplication, and vector magnitude. (If I were writing the kernel in C++ instead of C, I could define a class with real methods; but I’m going to stick with C.)

Each of the vector functions begins with the special CUDA keyword __device__ ("underscore underscore device underscore underscore"). This signals the CUDA compiler that the function is not a kernel function, but just a subroutine to be compiled into GPU code. (A kernel function begins with __global__ rather than __device__.)

The rest of the kernel is functionally the same as the ZombieGpu kernel in Chapter 25, and I’m not going to describe the new kernel in detail. I will point out where the new kernel uses the vector structure instead of separate X and Y coordinate arrays. Line 55 declares an array of velocity vectors in the block’s shared memory; this array is used in the sum-reduction that computes the zombie’s net velocity. The timeStep() kernel function’s first two arguments are pointers to arrays of vectors holding the current zombie positions and the next zombie positions (lines 81–82). In the kernel function code, the velocity computations are carried out using the vector methods defined earlier (lines 100–103). Compare this code to the equivalent code in Chapter 25, and see if you agree that with this code it’s easier to understand what’s going on. The sum-reduction that computes the zombie’s net velocity likewise uses one of the vector methods defined earlier (lines 107–114), as does the single-threaded code section that computes the zombie’s next position after the time step (lines 117–127).

The Java main program, class ZombieGpu2 (Listing 26.2), begins by defining a Java class Vector that corresponds to the kernel’s vector_t structure (line 16). Class Vector is a subclass of class edu.rit.gpu.Struct (line 17). The Struct superclass declares methods that the Parallel Java 2 Library uses to transfer instances of class Vector on the CPU to instances of structure vector_t on the GPU and vice versa. Class Vector declares the same fields as structure vector_t, namely x and y of type double (lines 19–20), as well as a constructor to initialize them (lines 23–29). Next come three methods that
// 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
(vector_t *pos,
 vector_t *next,
 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
 vector_t pos_i = pos[i]; // This zombie's current position
 vector_t vel = {0.0, 0.0}; // This zombie's velocity
 int k;
 vector_t posdiff;
 double d, v;
 // Compute and accumulate velocity w.r.t. every other zombie.
 for (k = j; k < N; k += NT)
 { if (k == i) continue;
   vectorSubtract (&posdiff, &pos[k], &pos_i);
   d = vectorMagnitude (&posdiff);
   v = G*exp(-d/L) - exp(-d);
   vectorAdd (&vel, &vel, scalarProduct (&posdiff, v/d));
 }

// Compute net velocity via shared memory parallel reduction.
shrVel[j] = vel;
__syncthreads();
for (k = NT/2; k > 0; k >>= 1)
{
 if (j < k)
   vectorAdd (&shrVel[j], &shrVel[j], &shrVel[j+k]);
__syncthreads();
}

Listing 26.1. ZombieGpu2.cu (part 2)
are declared in the Struct superclass. The static \texttt{sizeof()} method (lines 32–35) returns the amount of storage occupied by one instance of the \texttt{vertex.t} structure, namely 16 bytes—two fields of type \texttt{double}, each occupying 8 bytes. This method is static because the Library must be able to determine the structure’s size before any instances of the class have been created. The \texttt{toStruct()} method (lines 39–44) is called by the Library when an instance of class Vector is being transferred from the CPU to the GPU. The \texttt{toStruct()} method must write, into the given byte buffer, the bytes of the \texttt{vector.t} structure exactly as they would appear in the GPU’s memory. In general, this is done by calling methods on the byte buffer, such as \texttt{putInt()}, \texttt{putDouble()}, and so on, for each of the fields in the class; see the “Under the Hood” section below for further discussion. Here, the \texttt{toStruct()} method puts the \texttt{x} and \texttt{y} fields into the byte buffer. The \texttt{fromStruct()} method (lines 48–53) is called by the Library when an instance of class Vector is being transferred from the GPU to the CPU. The \texttt{fromStruct()} method must read, from the given byte buffer, the bytes of the \texttt{vector.t} structure exactly as they would appear in the GPU’s memory. In general, this is done by calling methods on the byte buffer, such as \texttt{getInt()}, \texttt{getDouble()}, and so on, for each of the fields in the class; see the “Under the Hood” section below for further discussion. Here, the \texttt{fromStruct()} method gets the \texttt{x} and \texttt{y} fields from the byte buffer.

The kernel interface is declared in lines 77–87. As always, the Java argument types correspond to the C argument types: Java type \texttt{int} for C type \texttt{int}; Java type \texttt{double} for C type \texttt{double}. The current position array and next position array arguments are of C type \texttt{vector.t*}; in Java these become type \texttt{GpuStructArray}. Class \texttt{GpuStructArray} provides an array of structures on the GPU mirrored as an array of objects on the CPU. Class \texttt{GpuStructArray} is a generic class, here specified with the generic type parameter \texttt{<Vector>}, indicating that it is an array of Vector objects.

The task main program proper begins on line 90. It is functionally the same as the task main program in Chapter 25, except it uses the Vector class. Lines 113–114 create arrays of \texttt{vector.t} structures on the GPU to hold the zombies’ current and next positions. The arrays are mirrored on the CPU in the \texttt{pos} and \texttt{next} fields of the \texttt{pos} and \texttt{next} variables. These variables are of type \texttt{GpuStructArray<Vector>}, the same type as the kernel function arguments. Lines 125–131 initialize the \texttt{pos} array elements to random \((x, y)\) coordinates and initialize the \texttt{next} array elements to \((0, 0)\). The initial \texttt{pos} array is then transferred from the CPU to the GPU (line 132). Similarly, the \texttt{snapshot()} method transfers the \texttt{pos} array from the GPU to the CPU (line 174) and prints the position vectors.

To compare the CPU’s performance to the GPU’s performance on the zombie simulation using structures, I ran the ZombieSeq program and the
Chapter 26. Objects on the GPU

26–7

// Single threaded section.
if (j == 0)
{
    // Get net velocity.
    vel = shrVel[0];

    // Move zombie in the direction of its velocity.
    vectorAdd (&next[i], &pos_i, scalarProduct (&vel, dt));

    // Accumulate position delta.
    atomicAdd (&devDelta, abs(vel.x) + abs(vel.y));
}

Listing 26.1. ZombieGpu2.cu (part 3)

package edu.rit.gpu.example;
import edu.rit.gpu.CacheConfig;
import edu.rit.gpu.Gpu;
import edu.rit.gpu.GpuDoubleVbl;
import edu.rit.gpu.GpuStructArray;
import edu.rit.gpu.Kernel;
import edu.rit.gpu.Module;
import edu.rit.gpu.Struct;
import edu.rit.pj2.Task;
import edu.rit.util.Random;
import java.nio.ByteBuffer;
public class ZombieGpu2
extends Task
{
    // Structure for a 2-D vector.
    private static class Vector
    extends Struct
    {
        public double x;
        public double y;

        // Construct a new vector.
        public Vector
        (double x,
         double y)
        {
            this.x = x;
            this.y = y;
        }

        // Returns the size in bytes of the C struct.
        public static long sizeof()
        {
            return 16;
        }
}

Listing 26.2. ZombieGpu2.java (part 1)
ZombieGpu2 program on the kraken machine, using commands like this:

```bash
$ 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.ZombieGpu2 \
   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>1743</td>
<td>6.9</td>
</tr>
<tr>
<td>200</td>
<td>7.07</td>
<td>13043</td>
<td>39568</td>
<td>2211</td>
<td>17.9</td>
</tr>
<tr>
<td>500</td>
<td>11.18</td>
<td>10186</td>
<td>192903</td>
<td>3485</td>
<td>55.4</td>
</tr>
<tr>
<td>1000</td>
<td>15.81</td>
<td>9308</td>
<td>706896</td>
<td>7228</td>
<td>97.8</td>
</tr>
<tr>
<td>2000</td>
<td>22.36</td>
<td>9595</td>
<td>2880873</td>
<td>23602</td>
<td>122.1</td>
</tr>
</tbody>
</table>

The GPU program is over 120 times faster than the CPU program for a large enough problem size.

In fact, the GPU program using vector structures (ZombieGpu2) is even faster than the GPU program using separate X and Y coordinate arrays (ZombieGpu). Here are the running times $T$ in milliseconds for the ZombieGpu and ZombieGpu2 programs:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$W$</th>
<th>Steps</th>
<th>ZombieGpu $T$</th>
<th>ZombieGpu2 $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5.00</td>
<td>15869</td>
<td>1704</td>
<td>1743</td>
</tr>
<tr>
<td>200</td>
<td>7.07</td>
<td>13043</td>
<td>2187</td>
<td>2211</td>
</tr>
<tr>
<td>500</td>
<td>11.18</td>
<td>10186</td>
<td>3615</td>
<td>3485</td>
</tr>
<tr>
<td>1000</td>
<td>15.81</td>
<td>9308</td>
<td>7947</td>
<td>7228</td>
</tr>
<tr>
<td>2000</td>
<td>22.36</td>
<td>9595</td>
<td>27286</td>
<td>23602</td>
</tr>
</tbody>
</table>

With 2000 zombies, the ZombieGpu2 program is about 14 percent faster than the ZombieGpu program. Why? Most likely because the ZombieGpu2 program is more “cache friendly.” The X and Y coordinates of each vector structure are stored in adjacent memory locations. When the kernel code pulls a vector’s X coordinate from GPU global memory, a number of adjacent memory locations also get pulled into the L2 and L1 caches—possibly including the vector’s Y coordinate. Then when the kernel code accesses the vector’s Y coordinate, much of the time the Y coordinate is already in the cache and can be retrieved quickly without needing to access global memory again. In contrast, the ZombieGpu program is not as cache friendly: a vector’s X and Y coordinates are not in adjacent memory locations, rather they are in separate X and Y coordinate arrays. When the kernel code pulls in a vector’s X coordinate, the Y coordinate is usually not also pulled into the cache; so when the
// Write this Java object to the given byte buffer as a C struct.
public void toStruct
(ByteBuffer buf)
{
    buf.putDouble (x);
    buf.putDouble (y);
}

// Read this Java object from the given byte buffer as a C struct.
public void fromStruct
(ByteBuffer buf)
{
    x = buf.getDouble();
    y = buf.getDouble();
}

// Command line arguments.
long seed;
int N;
double W;
double G;
double L;
double dt;
double eps;
int steps;
int snap;

// Current body positions.
GpuStructArray<Vector> pos;

// Next body positions.
GpuStructArray<Vector> next;

// For detecting convergence.
GpuDoubleVbl delta;

// Kernel function interface.
private static interface ZombieKernel
extends Kernel
{
    public void timeStep
    (GpuStructArray<Vector> pos,
    GpuStructArray<Vector> next,
    int N,
    double G,
    double L,
    double dt);
}

// Task main program.
public void main
(String[] args)
throws Exception
{
    // Parse command line arguments.
}
kernel accesses the Y coordinate, it has to go back to global memory to get it, which is slower than getting the Y coordinate from the cache.

**Under the Hood**

Writing the `toStruct()` and `fromStruct()` methods in a subclass of class `edu.rit.gpu.Struct` requires knowing how the fields of a C structure are laid out in the GPU’s memory. If the kernel function and the main program are both written in C, the compiler takes care of the structure layout automatically. But when the main program is written in Java, the Java compiler knows nothing about the C structure’s layout, and the onus is on the programmer.

When working with the Parallel Java 2 Library’s C structure capability, my first recommendation is to use only simple structures whose fields are all primitive types, like the `vector_t` structure in the ZombieGpu2 program. More complicated structures, especially structures with fields that are pointers to variable-sized dynamically-allocated data, become too difficult to deal with.

Confining our attention to structures with primitive fields, the first thing to understand is the correspondence between C types and Java types, and the size (number of bytes) occupied by each:

<table>
<thead>
<tr>
<th>C type</th>
<th>Java type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long long int</td>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>short int</td>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>char</td>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>byte</td>
<td>1</td>
</tr>
</tbody>
</table>

My second recommendation is that when defining the C structure in the kernel code, put all the 8-byte fields (if any) first, put all the 4-byte fields (if any) second, put all the 2-byte fields (if any) third, and put all the 1-byte fields (if any) last. This ensures (in conjunction with the padding rule described below) that each field is properly aligned in memory: 8-byte fields aligned to addresses that are multiples of 8 bytes, 4-byte fields aligned to addresses that are multiples of 4 bytes, and so on.

Once the fields of the C structure are defined in the kernel code, you can define the corresponding Java class in the main program code. The Java class’s fields are named the same as the C structure’s fields, *appear in the*
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]);
n snap = Integer.parseInt (args[8]);

// Initialize GPU.
Gpu gpu = Gpu.gpu();
gpu.ensureComputeCapability (2, 0);

// Set up GPU variables.
Module module = gpu.getModule
("edu/rit/gpu/example/ZombieGpu2.cubin");
pos = gpu.getStructArray (Vector.class, N);
next = gpu.getStructArray (Vector.class, N);
delta = module.getDoubleVbl ("devDelta");

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

// Set zombies' initial (x,y) coordinates at random in a WxW square region. Also allocate zombies' next positions.
Random prng = new Random (seed);
for (int i = 0; i < N; ++ i)
{
pos.item[i] = new Vector
(prng.nextDouble()*W, prng.nextDouble()*W);
next.item[i] = new Vector (0, 0);
}
pos.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();
kern el.timeStep (pos, next, N, G, L, dt);

    // Advance to next time step.
    ++ t;

    // Update positions.
    GpuStructArray<Vector> tmp;
tmp = pos; pos = next; next = tmp;
same order, and use the Java types corresponding to the C types as shown above.

The Java class’s `sizeof()` method returns the size of the C structure in bytes. This is the sum of the sizes of the fields, plus possibly some extra padding. Padding must be added if necessary to make the size of the structure be a multiple of the size of the largest field in the structure—the first field, according to my recommendation above. This ensures that if an array of structures is created, each structure in the array is properly aligned.

The `ZombieGpu2` program defined this C structure and Java class:

```c
typedef struct {
  double x;
  double y;
} vector_t;
```

The sum of the field sizes is 16. This is a multiple of the largest field size, 8. So no extra padding is needed, and the `sizeof()` method returns 16.

Here’s another example:

```c
typedef struct {
  double a;
  double b;
  int c;
} example_t;
```

The sum of the field sizes is 20. This is not a multiple of the largest field size, 8. So extra padding is needed, and the `sizeof()` method would return 24 rather than 20.

With the C structure’s fields laid out as recommended above, the Java class’s `toStruct()` method must put the fields into the byte buffer passed as an argument, in the order the fields are declared, using the `ByteBuffer` method corresponding to each field’s type: `putDouble()`, `putLong()`, `putFloat()`, `putInt()`, `putShort()`, or `putByte()`. Likewise, the Java class’s `fromStruct()` method must get the fields from the byte buffer passed as an argument, in the order the fields are declared, using the `ByteBuffer` method corresponding to each field’s type: `getDouble()`, `getLong()`, `getFloat()`, `getInt()`, `getShort()`, or `getByte()`.

In addition to the fields and methods described above, the Java class can define other constructors and methods as needed by the Java main program.

In the Java code, a Java variable that mirrors a C structure is created by calling the `getStructVbl()` method on a `Gpu` object or by calling the `getStructVbl()` method on a `Module` object. These methods return an instance of class `GpuStructVbl<T>`, where `T` is the Java class corresponding to the C
// 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);
}

// Snapshot all bodies' final positions.
snapshot(t);

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

// Print a usage message and exit.
private static void usage()
{
    System.err.println("Usage: java pj2 " +
        "edu.rit.pj2.example.ZombieGpu2 <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;
}

Listing 26.2. ZombieGpu2.java (part 4)
structure. The address of the C structure in GPU memory is stored inside the GpuStructVbl object. The GpuStructVbl object has a public item field of type T; this field is initially null and must be assigned an instance of class T. Thereafter, calling the GpuStructVbl object’s hostToDev() method uploads the item field to the GPU. The hostToDev() method creates a byte buffer, calls the toStruct() method on the item field, which puts the object’s contents into the byte buffer, and transfers the byte buffer’s contents to GPU memory at the address of the GPU variable; the number of bytes transferred is determined by calling the item field’s sizeof() method. Likewise, calling the GpuStructVbl’s devToHost() method downloads the item field from the GPU. The devToHost() method creates a byte buffer, transfers the contents of GPU memory from the address of the GPU variable into the byte buffer, and calls the fromStruct() method on the item field, which gets the object’s contents from the byte buffer.

A Java variable that mirrors an array of C structures is created by calling the getStructArray() method on a Gpu object or by calling the getStructArray() method on a Module object. These methods return an instance of class GpuStructArray<T>, where T is the Java class corresponding to the C structure. The address of the C structure array in GPU memory is stored inside the GpuStructArray object. The GpuStructArray object has a public item field of type T[]; the elements of this array are initially null and must be assigned instances of class T. Thereafter, calling the GpuStructArray object’s hostToDev() method uploads the item array to the GPU, and calling the devToHost() method downloads the item array from the GPU. The toStruct() and fromStruct() methods are called on each item array element to effect the transfer.

**Points to Remember**

- When designing the kernel function for a GPU parallel program, consider whether data might better be represented with a C structure.
- Limit the C structure to consist of fields of primitive types.
- Lay out the structure with the largest fields first and the smallest fields last.
- In the Java main program, define a class that extends class edu.rit.gpu.Struct, with the same fields as the C structure. Also define the class’s sizeof(), toStruct(), and fromStruct() methods.
- To mirror a C structure variable, create an instance of generic class edu.rit.gpu.GpuStructVbl. Assign an instance of the Java class to the GpuStructVbl’s item field.
- To mirror an array of C structures, create an instance of generic class edu.rit.gpu.GpuStructArray. Assign an instance of the Java class to each
element of the GpuStructArray’s item field.

- Use the usual hostToDev() and devToHost() methods to transfer structures from the CPU to the GPU and vice versa.
- Representing data with a structure might make the program more cache friendly, thereby improving its performance.