Chapter 29
GPU Massively Parallel

► Part I. Preliminaries
► Part II. Tightly Coupled Multicore
► Part III. Loosely Coupled Cluster
▼ Part IV. GPU Acceleration
  Chapter 29. GPU Massively Parallel
  Chapter 30. GPU Parallel Reduction
  Chapter 31. Multi-GPU Programming
  Chapter 32. GPU Sequential Dependencies
  Chapter 33. Objects on the GPU
  Chapter 34. GPU Heuristic Search
► Part V. Big Data
► Appendices
We now turn to graphics processing unit (GPU) accelerated parallel programming. In contrast to an unaccelerated node that might have a few or a few dozen CPU cores, a GPU accelerator might have hundreds or thousands of GPU cores. A GPU accelerated node can perform computations potentially many times faster than an unaccelerated multicore node.

To ease into GPU programming, I’ll use a simple example: computing the outer product of two vectors. The inputs are two n-element vectors (arrays) \( \mathbf{A} \) and \( \mathbf{B} \). The output is an \( n \times n \)-element matrix \( \mathbf{C} \), computed as follows:

\[
C_{ij} = A_i \times B_j, \quad 0 \leq i \leq n - 1, \quad 0 \leq j \leq n - 1
\]  

(24.1)

Here is an example of the outer product of two 10-element vectors:

<table>
<thead>
<tr>
<th>A</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>66</td>
<td>72</td>
<td>78</td>
<td>84</td>
</tr>
<tr>
<td>60</td>
<td>72</td>
<td>84</td>
<td>90</td>
<td>96</td>
<td>108</td>
<td>120</td>
<td>126</td>
<td>132</td>
<td>138</td>
<td>144</td>
</tr>
</tbody>
</table>

If I had a computer with 100 cores, I could compute the above matrix’s 100 elements all at once, each on its own separate core, in a massively parallel fashion, and get a theoretical speedup of 100 over a single core. Or consider computing the outer product of two 1000-element vectors. If I had a computer with one million cores, again I could compute all of the matrix’s elements at the same time in parallel, one element on each core.

A real compute node would be unlikely to have one million cores. But Nvidia Corporation, maker of GPU-based graphics cards, has developed a programming abstraction that makes a GPU accelerator look as though it has a nearly unlimited number of virtual cores. Nvidia calls this programming abstraction the Compute Unified Device Architecture (CUDA). The Parallel Java 2 Library supports writing GPU accelerated parallel programs that run on CUDA-capable graphics cards.

In this and the next few chapters, I’ll be using CUDA along with Parallel Java 2 to teach GPU parallel programming. CUDA’s capabilities are vast, and I’m not going to try to cover them all. I will hit CUDA’s basic features. You can pick up the rest from the CUDA documentation.

Figure 29.1 shows the hardware architecture of a typical CUDA GPU. This particular one is an Nvidia Tesla K40c, one of Nvidia’s older-generation...
GPU cards designed for high performance parallel computing. The kraken machine at RIT is a multicore node with 88 CPU cores and four Tesla K40c GPU accelerators. I’ll be using kraken to measure the performance of the GPU parallel programs in this book.

The Tesla K40c has 15 multiprocessors (MPs), each connected to a 12-gigabyte global memory. (Other GPUs have different numbers of multiprocessors and different amounts of global memory.) The global memory is connected via a high-speed bus to the CPU memory of the machine hosting the GPU; this lets the host CPU transfer data to and from the GPU global mem-
ory. Each multiprocessor can read and write data anywhere in the common global memory. The global memory is built from the same, comparatively slow, DRAM chips as are used in the CPU’s main memory.

Figure 29.2 is an expanded view of one Tesla K40c multiprocessor. It has 192 cores; a register bank with 65,536 high-speed 32-bit registers; a shared memory; and an L1 cache. The shared memory and the L1 cache together total 64 kilobytes. These can be configured either as 48 kilobytes of shared memory and 16 kilobytes of L1 cache, or as 16 kilobytes of shared memory and 48 kilobytes of L1 cache. (Other multiprocessors have different numbers of cores and registers, and different amounts of shared memory and L1 cache.) The shared memory and L1 cache use fast RAM circuitry, like the L1 cache in a CPU. The L1 cache connects to a somewhat slower L2 cache, which in turn connects to the very slow global memory. All the cores in the multiprocessor can access data located in their own multiprocessor’s fast shared memory, but cannot access data located in other multiprocessors’ shared memories. All the cores in all the multiprocessors can access data located in the slow global memory. Global memory data is cached in L2 and L1, thus reducing the time needed to access the data. However, the GPU’s L2 and L1 caches are typically much smaller than a CPU’s caches. GPU programs rely on additional techniques to reduce the effective access time for global memory data.

Summing up, the Tesla K40c GPU has 15 multiprocessors each with 192 cores, for a total of 2,880 cores, as well as a 12-gigabyte memory. Other GPUs have different quantities of computational resources. The newer Nvidia Tesla V100 GPU/TPU accelerator, for example, has 5,120 GPU cores, 640 TPU cores, and 16 gigabytes of memory.

To write GPU programs, however, you usually don’t need to deal with these hardware details. Instead, you work with the CUDA programming model, which presents a high level abstraction of the GPU hardware. The abstraction consists of two key components: the computational grid, and the kernel function.

The computational grid, or just grid, is an abstraction of the GPU’s multiprocessors and cores. Consider the outer product computation. The output is an $n \times n$ matrix. I want to compute the matrix elements, conceptually, all at once in parallel. To do so, I need an $n \times n$ matrix, or grid, of threads—one thread for each element. Each thread executes the same kernel function; the kernel function computes one output matrix element. The whole assemblage—a grid of threads, each thread executing the kernel function—is called the computational kernel, or just kernel.

Figure 29.3 depicts a two-dimensional grid of threads, suitable for computing the elements of a matrix. The grid is partitioned into thread blocks, or just blocks. The blocks are located within a two-dimensional coordinate system. In this example, the grid dimensions are $(6, 4)$; that is, 6 blocks in the $x$
Figure 29.3. A two-dimensional grid of two-dimensional blocks
direction by 4 blocks in the y direction. Each block has an x coordinate in the range 0–5 and a y coordinate in the range 0–3.

Each block in turn consists of a number of threads. The threads are located within a two-dimensional coordinate system inside the block. In this example, the block dimensions are (8, 8); that is, 8 threads in the x direction by 8 threads in the y direction. Each thread has an x coordinate in the range 0–7 and a y coordinate in the range 0–7. Each thread in the entire grid is uniquely identified by its block coordinates in the grid and its thread coordinates in the block.

CUDA lets you configure a kernel with a one-dimensional, a two-dimensional, or a three-dimensional grid. Each dimension of the grid can be anything up to 65,535 blocks. CUDA also lets you configure a kernel with a one-dimensional, a two-dimensional, or a three-dimensional block. Each dimension of the block can be anything, as long as the total number of threads in the block is 1,024 or less (for a Tesla K40c). Thus, the maximum grid size is 65,535×65,535×65,535 blocks times 1,024 threads per block, for a total of 288 quadrillion threads.

For the outer product GPU program, I’ll use a two-dimensional grid of two-dimensional blocks, because such a grid naturally matches the computation’s structure. Other programs in later chapters will use one-dimensional grids and blocks. I myself have not had occasion to use a three-dimensional grid; but I can see how a scientist computing a 3-D problem, like molecular modeling or fluid dynamics, might want to.

The blocks in the grid are abstractions of the multiprocessors in the GPU, and the threads in the block are abstractions of the cores in the multiprocessor. As such, the threads in one block have access to one multiprocessor’s fast shared memory. CUDA lets you declare program variables in shared memory; such variables are shared by all the threads in one block, but such variables are not accessible to threads in other blocks. CUDA also lets you declare program variables in global memory; such variables are shared by all the threads in all the blocks.

How is it possible to execute a grid with an enormous number of blocks, each block potentially containing over a thousand threads, on a GPU like the Tesla K40c with only 15 multiprocessors of 192 cores each? The same way that dozens or hundreds of threads can execute on a CPU with only a few cores. The operating system—in particular, the GPU card’s CUDA driver installed in the operating system—takes care of scheduling the grid’s blocks to execute on the GPU’s multiprocessors and scheduling each block’s threads to execute on the multiprocessor’s cores. You configure a kernel with the desired grid and block dimensions and tell the kernel to execute the desired kernel function; CUDA does the rest, ensuring that the kernel function is executed by each thread in the grid.

This brings us to the other part of the kernel, namely the kernel function.
A CUDA program consists of two parts: a main program that runs on the host CPU, and a kernel function that runs in each thread on the GPU (Figure 29.4). If you’re working purely with CUDA, you write both the main program and the kernel function in C, C++, or Fortran using the CUDA API, and you compile both pieces using the CUDA compiler, nvcc.

When doing GPU programming with the Parallel Java 2 Library, you write the main program in Java, and you write just the kernel function in C using CUDA. (I won’t be using C++ or Fortran in this book.) You compile the kernel function using the CUDA compiler, nvcc, and you compile the rest using the standard Java compiler, javac. I’d much prefer to write the whole thing in Java; but at this time, I don’t know of a way to write CUDA kernel functions in Java.

Parallel Java 2 provides Java classes for working with CUDA GPUs. Under the hood, these classes use the Java Native Interface (JNI) to invoke routines in the non-Java CUDA API, such as routines to configure a grid and execute a kernel function.

Combining a Java main program with a CUDA kernel in this way lets you take advantage of all of Parallel Java 2’s features for single-core, multi-core, and cluster parallel programming, and augment them with GPU acceleration. For example, you can easily write a parallel program that runs on multiple CPU cores, with each CPU core making use of its own GPU accelerator. This lets the program run on multiple GPUs at once, like the kraken machine’s four Tesla K40c GPUs. (We will see an example of just such a program later.)
Now we’re ready to start designing the GPU parallel vector outer product program. A typical GPU program follows this sequence of steps, which I call the computational arc:

- Running on the CPU, the main program sets up variables in the CPU’s memory and in the GPU’s memory to hold the program’s input and output data. Each variable on the CPU is said to mirror its counterpart on the GPU.
- The main program initializes the input variables in the CPU’s memory with the input data.
- The main program copies the input variables from the CPU’s memory to the GPU’s memory.
- The main program configures a kernel with certain grid dimensions and block dimensions, and with a certain kernel function.
- The main program launches the kernel and waits for the kernel to finish.
- On the GPU, every thread in the grid executes the kernel function, and stores the computed output data in the output variables in the GPU’s memory.
- Once the kernel finishes, back on the CPU, the main program copies the output variables from the GPU’s memory to the CPU’s memory.
- The main program prints the output data, stores the output data in a file, or does whatever else is necessary with the output data.

Figure 29.5 shows the vector outer product program’s input and output variables, mirrored in the CPU and the GPU: the input vectors \( A \) and \( B \), and the output matrix \( C \). The arrows show the program’s data flow as the computational arc progresses. Input data goes into \( A \) and \( B \) on the CPU; \( A \) and \( B \) are copied from the CPU to the GPU; the outer product is computed and stored into \( C \) on the GPU; \( C \) is copied from the GPU to the CPU; output data comes from \( C \) on the CPU.

As previously mentioned, the vector outer product program’s kernel is configured with a two-dimensional grid of two-dimensional blocks. There is one thread in the grid for each element in the outer product matrix \( C \). The kernel function computes one and only one matrix element, a different matrix element in each thread.

Listing 29.1 is the C source code for the kernel function, stored in the file OuterProductGpu.cu. (“cu” is the standard filename extension for a CUDA C source file.) The kernel function’s name is \texttt{outerProduct} (line 7). Every kernel function must be declared this same way:

- \texttt{extern "c"} — Needed by Parallel Java 2 to access the kernel function.
- \texttt{__global__} — Needed by \texttt{nvcc} to compile the kernel function for the GPU. (That special CUDA keyword is “underscore underscore global underscore underscore”.)
// Device kernel to compute the outer product matrix of two vectors.
// Called with a two-dimensional grid of two-dimensional blocks.
// a  First vector (input).
// b  Second vector (input).
// c  Outer product matrix (output).
// N  Vector length.
extern "C" __global__ void outerProduct
(double *a,
 double *b,
 double **c,
 int N)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    if (row < N && col < N)
        c[row][col] = a[row]*b[col];
}
void — A kernel function must not have a return value.

A kernel function can, and typically does, have arguments (lines 8–11). The vector outer product kernel function’s arguments are pointers to the input a and b arrays, of type double* (this is C, remember); a pointer to the output c matrix, of type double**; and the length of the input vectors, n. The pointers and the length must be supplied as arguments because the arrays and the matrix will be dynamically allocated at run time based on user input.

The kernel function’s sole purpose is to compute one output matrix element at a certain row and column (line 16). Each thread executing the kernel function must compute a different row and column. To do so, the thread must be able to determine its position within the grid.

A thread determines its position in the grid by querying four special pseudo-variables:

- blockDim gives the grid’s dimensions: blockDim.x the x dimension, blockDim.y the y dimension, blockDim.z the z dimension.
- blockIdx gives the indexes of the currently executing thread’s block within the grid: blockIdx.x the x index, blockIdx.y the y index, blockIdx.z the z index.
- threadIdx gives the indexes of the currently executing thread within its block: threadIdx.x the x index, threadIdx.y the y index, threadIdx.z the z index.

For a one-dimensional grid or block, only the x dimensions and indexes are used. For a two-dimensional grid or block, only the x and y dimensions and indexes are used. For a three-dimensional grid or block, the x, y, and z dimensions and indexes are used.

The expression on line 13 computes the thread’s y index (row) within the whole grid. For example, consider grid dimensions (6, 4), block dimensions (8, 8), and consider the thread at block index (2, 3), thread index (5, 4), as shown in Figure 29.3. The thread’s row is

$$\text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} = 3 \times 8 + 4 = 28.$$  

Likewise, the expression on line 14 computes the thread’s x index (column) within the whole grid. The thread’s column is

$$\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} = 2 \times 8 + 5 = 21.$$  

This particular thread computes the output matrix element at row 28, column 21, as you can verify by counting the rows and columns in Figure 29.3. (Remember that the row and column indexes start at 0.) Other threads compute
different rows and/or columns, as determined by their own block indexes and thread indexes.

Before proceeding to compute the matrix element, however, the kernel function has to check that the row index and the column index are in bounds (line 15). This is critical, because while all the blocks in the grid have the same dimensions, the input vector length $n$ might not be a whole multiple of the blocks’ $x$ or $y$ dimension. In that case, the grid will have to have extra blocks to cover the leftover matrix elements. For example, consider a $25 \times 25$-element matrix being computed by a grid of blocks with $10 \times 10$ threads (Figure 29.6). The grid needs to have $3 \times 3$ blocks to cover all the matrix elements; but some of the threads do not have a corresponding matrix element. The test on line 15 ensures that such threads do not try to read or write nonexistent elements.

Figure 29.6. $3 \times 3$ grid of $10 \times 10$ blocks computing a $25 \times 25$ matrix
The kernel function is compiled with this command:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \n--ptxas-options="-v" -o OuterProductGpu.cubin \nOuterProductGpu.cu
```

This command tells the CUDA compiler to compile the source file OuterProductGpu.cu and to generate a **CUDA module file** named OuterProductGpu.cubin. (For a more detailed explanation of the compilation process, see the “Under the Hood” section below.) The module file ends up in the same directory as the .class files produced by the Java compiler.

Having written the kernel function in C, we turn to the other part of the GPU parallel vector outer product program, namely the main program written in Java, class OuterProductGpu (Listing 29.2). The main program class is typically a Task, like other Parallel Java 2 programs (line 11).

Parallel Java 2 makes the kernel function appear to be a regular Java method that you can call. This **kernel method** is declared in a **kernel interface** (lines 14–22). The name of the kernel interface can be anything; I called it OuterProductKernel. The kernel interface must extend interface edu.rit.gpu.Kernel. The name of the kernel method must be identical to the kernel function, outerProduct. The kernel method must be declared to return void and must be declared with the same arguments in the same order as the kernel function. The kernel interface must declare one and only one method.

The Java data types of the kernel method’s arguments depend on the C data types of the kernel function’s arguments. A complete list of the corresponding Java and C argument data types appears in the Javadoc documentation for interface edu.rit.gpu.Kernel. Here, the input arrays a and b of C type *double* are declared as Java type edu.rit.gpu.GpuDoubleArray; the output matrix c of C type *double** is declared as Java type edu.rit.gpu.GpuDoubleMatrix; and the argument N of C type *int* is declared as Java type *int*.

It is extremely important to declare the Java kernel method properly. If this does not correspond to the C kernel function exactly, the program will fail in strange and unpredictable ways when run. Because the Java compiler knows nothing about CUDA kernel functions, there is unfortunately no way for the Java compiler to catch errors in the kernel method declaration.

To access the GPU, the main program must first obtain a **GPU object** by calling the static Gpu.gpu() method (line 40). The ensureComputeCapacity() method call on line 41 has to do with the way the kernel function was compiled; see the “Under the Hood” section below for further information.

The main program creates an array to hold the input A vector by calling the GPU object’s getDoubleArray() method, passing in the array length (line 45). This method creates the array in the CPU’s memory, stores a reference to the array in a GpuDoubleArray object, and returns the GpuDoubleArray. This method also creates the array in the GPU’s global memory. Like-
Chapter 29. GPU Massively Parallel

package edu.rit.gpu.example;
import edu.rit.gpu.CacheConfig;
import edu.rit.gpu.Kernel;
import edu.rit.gpu.Gpu;
import edu.rit.gpu.GpuDoubleArray;
import edu.rit.gpu.GpuDoubleMatrix;
import edu.rit.gpu.Module;
import edu.rit.pj2.Task;
import edu.rit.util.Random;

public class OuterProductGpu
extends Task
{
    // Kernel function interface.
    private static interface OuterProductKernel
        extends Kernel
    {
        public void outerProduct
            (GpuDoubleArray a,
             GpuDoubleArray b,
             GpuDoubleMatrix c,
             int N);
    }

    // GPU kernel block dimensions = NT x NT threads.
    private static final int NT = 32;

    // Task main program.
    public void main
        (String[] args)
    throws Exception
    {
        long t1 = System.currentTimeMillis();

        // Validate command line arguments.
        if (args.length != 2) usage();
        long seed = Long.parseLong (args[0]);
        int N = Integer.parseInt (args[1]);

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

        // Set up input vectors and output matrix.
        Random prng = new Random (seed);
        GpuDoubleArray a = gpu.getDoubleArray (N);
        GpuDoubleArray b = gpu.getDoubleArray (N);
        GpuDoubleMatrix c = gpu.getDoubleMatrix (N, N);
        for (int i = 0; i < N; ++ i)
        {
            a.item[i] = prng.nextDouble()*20.0 - 10.0;
            b.item[i] = prng.nextDouble()*20.0 - 10.0;
        }
        a.hostToDev();
        b.hostToDev();

        // Compute outer product.
        Module module = gpu.getModule
            ("edu/rit/gpu/example/OuterProductGpu.cubin");

Listing 29.2. OuterProductGpu.java (part 1)
wise, the main program creates an array to hold the input \textbf{B} vector (line 46). To create the output \textbf{C} matrix, the main program calls the GPU object’s \texttt{getDoubleMatrix()} method (line 47), passing in the number of rows and columns. The main program must create the arrays and matrix this way, rather than constructing the arrays and matrix directly, so that the arrays’ and matrix’s contents can be transferred between the CPU and the GPU. Also note that the arrays’ and matrix’s data types are those of the previously declared kernel method’s arguments.

For this example, the main program initializes the \textbf{A} and \textbf{B} vectors with random elements in the range $-10.0$ to $+10.0$ (lines 48–52), storing the values into the actual arrays, which are the \texttt{item} fields in the \texttt{a} and \texttt{b} objects. The main program then transfers the input vectors from the CPU to the GPU by calling the \texttt{hostToDevice()} method on the \texttt{a} and \texttt{b} objects (lines 53–54).

With the input data set up and transferred to the GPU’s memory, the main program proceeds to the kernel. The main program first obtains a \textit{module object} by calling the GPU object’s \texttt{getModule()} method (lines 57–58). The argument is the name of the CUDA module file that contains the compiled kernel function. The module file name is stated \textit{relative to the Java class path}. Because class \texttt{OuterProductGpu} is in package \texttt{edu.rit.gpu.example}, the compiled Java class file \texttt{OuterProductGpu.class} is in directory \texttt{edu/rit/gpu/example} under the top level directory of the Java class path. The compiled CUDA module file \texttt{OuterProductGpu.cubin} must be specified as “\texttt{edu/rit/gpu/example/OuterProductGpu.cubin}”. If you are not using Java packages, all the compiled .class and .cubin files would typically be in the top level directory of the Java class path, and you would not need to include the directory name.

To configure and invoke the kernel, the main program obtains a \textit{kernel object} by calling the module object’s \texttt{getKernel()} method (lines 59–60), specifying the class of the kernel interface declared earlier. The \texttt{getKernel()} method returns a reference to a \textit{proxy object} that implements the kernel interface. Thus, you can call the kernel method on the kernel object. You can also call other methods on the kernel object, which are declared in the Kernel superinterface.

The main program specifies the block and grid dimensions for the kernel by calling the kernel object’s \texttt{setBlockDim()} and \texttt{setGridDim()} methods (lines 61–62). Each block is two-dimensional and consists of $NT \times NT$ threads, where $NT$ was declared earlier to be 32 (line 25). Thus, each block will have the maximum possible number of threads, namely 1,024. The grid is also two-dimensional, with each dimension consisting of $\text{ceil}(N/NT)$ blocks. (The ceiling function is computed with the expression $(N + NT - 1)/NT$; if $N$ is not a multiple of $NT$, this rounds up to the next higher integer.) Thus, the entire grid will have enough threads to compute every matrix element, possibly with extra threads if $N$ is not a multiple of $NT$ as shown in Figure 29.6.
OuterProductKernel kernel =
    module.getKernel (OuterProductKernel.class);
kernel.setBlockDim (NT, NT);
kernel.setGridDim ((N + NT - 1)/NT, (N + NT - 1)/NT);
kernel.setCacheConfig (CacheConfig.CU_FUNC_CACHE_PREFER_L1);
long t2 = System.currentTimeMillis();
kernel.outerProduct (a, b, c, N);
long t3 = System.currentTimeMillis();

// Print results.
c.devToHost();
System.out.printf("a[%d] = %.5f\n", 0, a.item[0  ]);
System.out.printf("a[%d] = %.5f\n", 1, a.item[1  ]);
System.out.printf("a[%d] = %.5f\n", N-2, a.item[N-2]);
System.out.printf("a[%d] = %.5f\n", N-1, a.item[N-1]);
System.out.printf("b[%d] = %.5f\n", 0, b.item[0  ]);
System.out.printf("b[%d] = %.5f\n", 1, b.item[1  ]);
System.out.printf("b[%d] = %.5f\n", N-2, b.item[N-2]);
System.out.printf("b[%d] = %.5f\n", N-1, b.item[N-1]);
System.out.printf("c[%d][%d] = %.5f\n", 0, 0, c.item[0  ][0  ]);
System.out.printf("c[%d][%d] = %.5f\n", 0, 1, c.item[0  ][1  ]);
System.out.printf("c[%d][%d] = %.5f\n", 0, N-2, c.item[0  ][N-2]);
System.out.printf("c[%d][%d] = %.5f\n", 0, N-1, c.item[0  ][N-1]);
System.out.printf("c[%d][%d] = %.5f\n", 1, 0, c.item[1  ][0  ]);
System.out.printf("c[%d][%d] = %.5f\n", 1, 1, c.item[1  ][1  ]);
System.out.printf("c[%d][%d] = %.5f\n", 1, N-2, c.item[1  ][N-2]);
System.out.printf("c[%d][%d] = %.5f\n", 1, N-1, c.item[1  ][N-1]);
System.out.printf("c[%d][%d] = %.5f\n", N-2, 0, c.item[N-2][0  ]);
System.out.printf("c[%d][%d] = %.5f\n", N-2, 1, c.item[N-2][1  ]);
System.out.printf("c[%d][%d] = %.5f\n", N-2, N-2, c.item[N-2][N-2]);
System.out.printf("c[%d][%d] = %.5f\n", N-2, N-1, c.item[N-2][N-1]);
System.out.printf("c[%d][%d] = %.5f\n", N-1, 0, c.item[N-1][0  ]);
System.out.printf("c[%d][%d] = %.5f\n", N-1, 1, c.item[N-1][1  ]);
System.out.printf("c[%d][%d] = %.5f\n", N-1, N-2, c.item[N-1][N-2]);
System.out.printf("c[%d][%d] = %.5f\n", N-1, N-1, c.item[N-1][N-1]);

// Print running times.
long t4 = System.currentTimeMillis();
System.out.printf("%d msec pre\n", t2 - t1);

Listing 29.2. OuterProductGpu.java (part 2)
Because the kernel does not use the multiprocessor’s shared memory, I
want to configure the multiprocessor’s fast RAM to consist of more L1 cache
and less shared memory. Increasing the amount of L1 cache will reduce the
time required to access the arrays and matrix in the GPU’s global memory.
The main program configures the shared memory and L1 cache for the kernel
by calling the kernel object’s setCacheConfig() method (line 63), specify-
ing that L1 is preferred over shared memory.

Finally, after all this setup, the main program launches the kernel by call-
ing the kernel method on the kernel object (line 65), passing in the requisite
arguments. At this point the main program blocks waiting for the kernel to
finish execution. When the kernel method returns, the computation on the
GPU is complete. The main program transfers the output matrix from the
GPU to the CPU by calling the devToHost() method on the c object (line
69). The main program prints selected elements of the input A and B vectors
and the output C matrix, obtaining the elements from the item fields of the a,
b, and c objects. After printing the running times for various sections of the
program, the task terminates.

The OuterProductGpu task class specifies two additional pieces of infor-
mation: that it requires one CPU core (lines 133–136) and one GPU accelera-
tor (lines 139–142). When the task is executed on a node or a cluster where
the Parallel Java 2 Tracker is running, the Tracker ensures that the task runs
on a node that has one idle CPU core and one idle GPU accelerator. The
Tracker tells the task which GPU to use, and the $\text{gpu.gpu()}$ method on line
40 returns a GPU object referring to this GPU.

To compare the speed of the GPU with that of the CPU, I wrote a single-
threaded CPU-only version of the vector outer product program in Java, class
dedu.rit.gpu.example.OuterProductSeq. I ran the CPU-only program on one
core of kraken and the GPU program on one GPU accelerator of kraken.
The kraken machine has two Intel Xeon E5-2696 processors, each with 22
dual-hyperthreaded CPU cores, running at a 2.2 GHz clock rate; kraken also
has four 2,880-core Nvidia Tesla K40c GPU accelerators, running at a 0.875
GHz clock rate. Here are the programs’ running times in milliseconds for
vectors of various lengths $n$. Both the total running times and the running
times for just the outer product calculation are listed.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Total Time</th>
<th>Calculation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>1000</td>
<td>34</td>
<td>637</td>
</tr>
<tr>
<td>2000</td>
<td>47</td>
<td>755</td>
</tr>
<tr>
<td>4000</td>
<td>92</td>
<td>1202</td>
</tr>
<tr>
<td>8000</td>
<td>494</td>
<td>1958</td>
</tr>
<tr>
<td>16000</td>
<td>1745</td>
<td>5052</td>
</tr>
</tbody>
</table>
Chapter 29. GPU Massively Parallel

The data shows that the GPU program’s total running time is longer than the CPU-only program’s. This is because the GPU program has to transfer two large vectors and a very large matrix between the CPU memory and the GPU memory—something the CPU-only program doesn’t have to do. On the other hand, the parallelizable calculation runs 15 to 22 times faster on the GPU than on the CPU.

This example program, calculating a vector outer product, is not one that utilizes a GPU accelerator to best advantage. The calculation is trivial and doesn’t take that long even on a single CPU core. I used this example primarily to introduce GPU and CUDA programming concepts. In the next few chapters we’ll do some more serious computations on the GPU.

Under the Hood

Let’s dissect the command that compiles the kernel function’s kernel source file:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \   --ptxas-options="-v" -o OuterProductGpu.cubin \ OuterProductGpu.cu
```

Listing 29.2. OuterProductGpu.java (part 3)
The CUDA compiler can produce two kinds of compiled output. One alternative is a “PTX” file. This contains instructions for a device-independent virtual machine called PTX. PTX is analogous to the device-independent Java Virtual Machine (JVM), and a PTX file is analogous to a Java class file. The Java compiler translates a Java source file into an intermediate Java class file. When the Java program is executed, the just-in-time compiler in the JVM translates the class file into machine code for the particular CPU where the JVM is running. In the same way, the CUDA compiler can translate a CUDA source file into an intermediate PTX file. When the GPU program is executed, the CUDA driver finishes the translation process and converts the PTX file into machine code for the particular GPU running the program. While this run-time translation imposes some overhead when the GPU program starts up, a PTX file can be executed on any kind of CUDA-capable GPU.

The other alternative is for the CUDA compiler to produce a “CUDA binary” file. The CUDA compiler translates the CUDA source file all the way down to machine instructions for a particular kind of GPU. This eliminates the overhead at program startup; but the price is that the CUDA binary file can be executed only on the particular kind of GPU for which it was compiled.

If portability across different kinds of GPUs is important, generate PTX files. If you’re going to be using a specific GPU all the time, generate CUDA binary files and eliminate the startup overhead. That’s what I usually do. The “-cubin” option tells the CUDA compiler to produce a CUDA binary file. (See the CUDA documentation for information about other options.)

Various CUDA-capable GPUs have different compute capabilities. A compute capability is denoted by a major version number and a minor version number, such as “compute capability 1.0.” The compute capability specifies features that are supported, such as the number of registers in a multi-processor, the maximum number of threads in a block, the maximum size of the multiprocessor’s shared memory, the existence of certain specialized machine instructions, and so on. (See the CUDA documentation for a complete list of the compute capabilities and their supported features.) Each higher compute capability supports all the features of the lower compute capabilities and extends existing features or adds new features. The CUDA compiler needs to know the compute capability for which the source file is being compiled, so the compiler can flag as errors any features not supported by that compute capability. The “-arch compute_20” option specifies compute capability 2.0, which is the compute capability a Tesla C2075 GPU card supports. (This is the older GPU card in my desktop PC.)

Before launching a GPU kernel that was compiled assuming a certain compute capability, you need to verify that the GPU you’re using in fact supports that compute capability. If it doesn’t, the kernel function might fail if it
tries to utilize a feature the GPU doesn’t support. This is the purpose of the `ensureComputeCapability()` method call on line 41. The arguments are the major and minor version numbers, 2 and 0 in this case (compute capability 2.0). If the GPU supports the given compute capability or any higher compute capability, the method just returns. Otherwise, the method throws an exception that will abort the program.

When generating a CUDA binary file with the "-cubin" option, the CUDA compiler needs to know the GPU machine language into which to translate the source file. The "-code sm_20" option specifies the machine language associated with compute capability 2.0. (Again, see the CUDA documentation for a list of the GPU machine languages.)

The "-ptxas-options="-v"" option tells the CUDA compiler to turn on verbose output from the PTX assembler that is generating the machine code in the CUDA binary file. This verbose output includes some important information. Here is what the `nvcc` command prints:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \ 
   --ptxas-options="-v" -o OuterProductGpu.cubin OuterProductGpu.cu
ptxas info: 0 bytes gmem
ptxas info: Compiling entry function 'outerProduct' for 'sm_20'
ptxas info: Function properties for outerProduct
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info: Used 10 registers, 48 bytes cmem[0]
```

A kernel function typically declares local variables. The vector outer product kernel function declared the local variables `row` and `col`. The compiler places local variables in registers in the multiprocessor’s register bank. The compiler also uses registers to hold intermediate values of expressions. The last line of output above tells us that the compiled kernel function uses 10 registers. So each thread in a block, executing the kernel function, needs its own separate group of 10 registers. I configured the kernel with 1,024 threads per block, so each block requires 10,240 registers. A compute capability 2.0 GPU has 32,768 registers in each multiprocessor. So all is well.

If you configure a kernel with too many threads, such that a block requires more registers than a multiprocessor has, the kernel will fail to launch, and the Java main program will throw an exception. It’s important to be aware of your kernel function’s register usage. You have to make sure that the number of registers per thread times the number of threads per block does not exceed the number of registers per multiprocessor. You might have to reduce the number of threads per block to get them to fit.

Finally, the "-o OuterProductGpu.cubin" option gives the name of the output CUDA binary file the compiler is to generate, and "OuterProductGpu.cu" gives the name of the CUDA source file to compile.
Points to Remember

• A CUDA-capable GPU consists of a number of multiprocessors plus a global memory.
• Each multiprocessor has a number of cores, a bank of high-speed registers, a fast shared memory, and a fast L1 cache memory. The multiprocessors can be configured with more shared memory and less L1 cache, or vice versa.
• A CUDA kernel consists of a grid plus a kernel function.
• A grid consists of a number of blocks in a one-, two-, or three-dimensional arrangement.
• A block consists of a number of threads in a one-, two-, or three-dimensional arrangement.
• Each thread in the grid executes the same kernel function.
• Each thread in the grid has a unique identity, determined by the grid dimensions (gridDim), block dimensions (blockDim), block index (blockIdx), and thread index (threadIdx).
• A GPU parallel program consists of a task main program written in Java using the Parallel Java 2 Library, plus a kernel function written in C (or C++ or Fortran) written using CUDA.
• The GPU computational arc: Input data flows from the CPU memory to the GPU memory, calculations take place on the GPU, output data flows from the GPU memory to the CPU memory.
• In the Java main program, declare the kernel method in a kernel interface that extends interface edu.rit.gpu.Kernel.
• Use Java classes in package edu.rit.gpu, such as GpuDoubleArray and GpuDoubleMatrix, to set up variables mirrored in the CPU memory and the GPU memory. Call methods on these objects to transfer data from the CPU to the GPU or vice versa.
• To execute a kernel, use Java classes in package edu.rit.gpu to get a module object; get a kernel object that implements the kernel interface; configure the grid in the kernel object; and call the kernel method on the kernel object.
• Define the static coresRequired() and gpusRequired() methods in the task class.
• Be aware of the GPU’s compute capability, and compile the kernel source file appropriately.
• Be aware of the kernel function’s register usage, the number of registers the threads in a block require, and the maximum number of registers available in a multiprocessor.