Chapter 31
Multi-GPU Programming

- 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
Suppose I have a node with more than one GPU accelerator. The kraken machine, for example, has four Nvidia Tesla K40c GPU cards, as well as 88 CPU cores (44 dual-hyperthreaded cores). Each Tesla card has 2,880 GPU cores. How can I utilize all 11,520 GPU cores on this node?

One way is to use the massively parallel approach introduced in Chapter 18 in the context of a cluster parallel computer. I can run four separate GPU accelerated parallel programs on kraken at once, each program running on a separate GPU.

But what I really might like to do is run one parallel program and scale it up to use all the GPUs on the node. To illustrate this approach, let’s change the GPU parallel π estimating program so it can run on multiple GPUs.

The simplest way to do this is to combine the multicore paradigm from Chapter 8 with the GPU accelerated paradigm from Chapter 30 (Figure 31.1). The program consists of multiple threads. There is one thread for each GPU on the node (not one thread for each CPU core). The N loop iterations (dart throws) are partitioned among the threads, in the manner with which we are familiar. Each thread runs a computational kernel on its own GPU—in fact, the exact same kernel as in Chapter 30. The kernel computes the number of darts within the circle quadrant for the thread’s portion of the loop iterations. These kernel results become the semifinal counts for each thread—C₀, C₁, and so on. The threads’ semifinal counts are sum-reduced, again in the manner with which we are familiar, to produce the final count C.

The second version of the GPU parallel π estimating program, class PiGpu2 (Listing 31.1), begins the same as the first version, with the command line arguments seed and N (lines 13–14) and the kernel function interface (lines 18–24). It’s the same kernel function interface as the previous version. In addition, there is a reduction variable of type LongVbl named count (line 15); this will hold the final count after the per-thread semifinal counts are reduced together.

The task main program (line 26) begins by obtaining the command line arguments and initializing the count global variable to do a sum reduction (line 36). Next the program sets up a parallel thread team. The parallelDo() method on line 39 creates the thread team, where the first argument is the number of threads in the team, namely the number of GPU accelerators, and the second argument is the parallel section object containing the code each team thread will execute. The number of GPUs is determined by calling the Gpu.allowedDeviceCount() method, which returns the number of GPUs this process is allowed to use. As we will see later, the default is to use all the GPUs on the node; but this can be overridden with an option on the pj2 command line.

Each parallel team thread now calls the run() method on its own copy of the parallel section object defined in the anonymous inner class starting on line 40. The thread first creates its own per-thread reduction variable,
Chapter 31. Multi-GPU Programming

Figure 31.1. Estimating $\pi$ with a single GPU and with multiple GPUs

Listing 31.1. PiGpu2.java (part 1)

```java
package edu.rit.gpu.example;
import edu.rit.gpu.Kernel;
import edu.rit.gpu.Gpu;
import edu.rit.gpu.GpuLongVbl;
import edu.rit.pj2.LongChunk;
import edu.rit.pj2.Section;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.LongVbl;
public class PiGpu2
    extends Task
    {
        private static long seed;
        private static long N;
        private static LongVbl count;

        // Kernel function interface.
        private static interface PiKernel
            extends Kernel
            {
                public void computeRandomPoints
                    (long seed,
                    long N);
            }
```
thrCount, linked to the global reduction variable count (line 44). The thread next obtains a GPU object (line 47). Under the hood, the gpu.gpu() method returns a different GPU object, representing a different GPU accelerator, to each calling thread. Thus, each thread ends up working with its own separate GPU. The rest of the run() method is almost identical to the single-GPU π estimating program in Chapter 30, except for two things.

The first difference is that before launching the GPU kernels, the N loop iterations must be partitioned among the parallel team threads. This is done by calling the LongChunk.partition() method on line 57. The method partitions the total index range (0 through N–1) into as many equal-sized chunks as there are threads in the team (threads()) and returns the chunk associated with the current thread’s rank in the team (rank()). The length of this chunk (length()) is the number of iterations (thrN) the current thread, that is, the current thread’s GPU kernel, will perform.

The second difference is in the arguments passed to the GPU kernel method on lines 67–68. Each thread’s kernel must generate a different sequence of random values; so the seed passed to the kernel is the seed from the command line plus one million times the current thread’s rank. Thus, the seed for thread rank 0 is just seed; the seed for thread rank 1 is seed + 1,000,000; the seed for thread rank 2 is seed + 2,000,000; and so on. Inside the kernel, each GPU thread in turn adds its own rank to this seed, and the result is used to initialize each GPU thread’s pseudorandom number generator (PRNG). Thus, each GPU thread’s PRNG is initialized with a different seed and generates a different sequence of random values. (This assumes that the kernel will have fewer than one million GPU threads, which seems reasonable.) Also, the number of iterations the kernel will perform is specified as thrN, the per-thread number—not N, the total number.

After each thread’s kernel method returns, the thread downloads the kernel’s count and stores it in the thread’s own per-thread thrCount variable (lines 72–73). After all the threads have finished, the per-thread counts are automatically sum-reduced into the global count variable, which is used to print the answer (lines 78–79).

Although the PiGpu2 program runs with a team of multiple threads, most of the time the threads are blocked waiting for the GPU kernel method to return. There is no need to tie up a whole core for each thread. Accordingly, the coresRequired() method is overridden to specify that the program needs only one core (lines 93–96). When the Tracker schedules the program to run, the Tracker needs to find a node with only one idle core; all the threads will share this core.

On the other hand, the PiGpu2 program wants to use all the GPU accelerators on the node, and the gpusRequired() method is overridden to specify this (lines 100–103). When the Tracker schedules the program to run, the Tracker needs to find a node all of whose GPUs are idle. You can specify that
Chapter 31. Multi-GPU Programming

Listing 31.1. PiGpu2.java (part 2)
the program use a particular number of GPUs by including the “gpus=” option on the pj2 command line.

To study the PiGpu2 program’s weak scaling performance, I ran the program on one to four GPUs on the kraken machine. The scale factor $K$ was the number of GPU accelerators (not the number of CPU cores). I ran the program with problem size $N =$ one, two, five, ten, and twenty trillion darts with one GPU. As I increased $K$, I also increased $N$ in the same proportion. Here are examples of the commands I used:

$\$ java pj2 debug=makespan gpus=1 edu.rit.gpu.example.PiGpu2 142857 1000000000000
$\$ java pj2 debug=makespan gpus=2 edu.rit.gpu.example.PiGpu2 142857 2000000000000

Figure 31.2 plots the running times, sizeups, and efficiencies I observed. The fitted running time model is

\[ T = (0.784 + 3.73 \times 10^{-13} N) + (0.180) \cdot K + (1.06 \times 10^{-10} N) / K \quad (31.1) \]

The program’s sequential fraction ranges from 0.0228 for the smallest problem size down to 0.00395 for the largest problem size. Once again, we see that as the amount of computation increases, the overhead due to the fixed sequential portion diminishes, resulting in higher efficiencies. Each parallel team thread takes 0.180 seconds to do its one-time initialization, mostly setting up the thread’s GPU. Each dart throw takes $1.06 \times 10^{-10}$ seconds, for a computation rate of 9.43 billion darts per second.

Below are the estimates for $\pi$ calculated by the program for various problem sizes $N$. Note how the estimate improves—$\Delta$, the relative difference between the program’s estimate and the actual value of $\pi$, trends downward—as the number of darts increases. Programs like this must do enormous numbers of iterations to get accurate answers, which makes such programs attractive candidates for parallelization.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\pi$ Estimate</th>
<th>$\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4 \times 10^{12}$</td>
<td>3.141592930</td>
<td>$8.80 \times 10^{-8}$</td>
</tr>
<tr>
<td>$8 \times 10^{12}$</td>
<td>3.141593293</td>
<td>$2.04 \times 10^{-7}$</td>
</tr>
<tr>
<td>$2 \times 10^{13}$</td>
<td>3.141592801</td>
<td>$4.69 \times 10^{-8}$</td>
</tr>
<tr>
<td>$4 \times 10^{13}$</td>
<td>3.141592583</td>
<td>$2.25 \times 10^{-8}$</td>
</tr>
<tr>
<td>$8 \times 10^{13}$</td>
<td>3.141592653</td>
<td>$1.88 \times 10^{-10}$</td>
</tr>
</tbody>
</table>
Points to Remember

• To run a single parallel program on multiple GPUs, dedicate a separate thread to each GPU.
• Call Gpu.allowedDeviceCount() to determine the number of GPU accelerators the program is allowed to use.
• Use the parallelDo statement to set up a parallel thread team with one thread for each GPU.
• Each thread gets its own Gpu object, runs its own computational kernel, and retrieves its own kernel’s result.
• Reduce the per-thread results together using reduction variables with the multithreaded reduction pattern.
• Override the coresRequired() method to specify that the task requires one core. All the parallel team threads will share this core.
• Override the gpusRequired() method to specify that the task requires all the GPUs on the node.
Figure 31.2. PiGpu2 weak scaling performance metrics