Chapter 24
Multi-GPU Programming

► 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
Suppose I have a node with more than one GPU accelerator. The kraken machine, for example, has four Nvidia Tesla C2075 GPU cards, as well as 80 CPU cores (40 dual-hyperthreaded cores). Each Tesla card has 448 GPU cores. How can I utilize all 1,792 GPU cores on this node?

One way is to use the massively parallel approach introduced in Chapter 13 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 \( \pi \) estimating program so it can run on multiple GPUs.

The simplest way to do this is to combine the multicore paradigm from Chapter 4 with the GPU accelerated paradigm from Chapter 23 (Figure 24.1). 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 23. 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—\( C_0, C_1 \), 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 \( \pi \) estimating program, class PiGpu2 (Listing 24.1), 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 24. Multi-GPU Programming

Figure 24.1. Estimating $\pi$ with a single GPU and with multiple GPUs

Listing 24.1. PiGpu2.java (part 1)
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 23, 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 par-
titions 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 se-
quence 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 re-
sult 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 reason-
able.) 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 ker-
nel’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 re-
turn. 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 83–86). 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 overrid-
den to specify this (lines 90–93). 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 the program use a particular number of GPUs by including the
// Task main program.
public void main
(String[] args)
throws Exception
{
    // Validate command line arguments.
    if (args.length != 2) usage();
    seed = Long.parseLong (args[0]);
    N = Long.parseLong (args[1]);

    // Set up global counter variable.
    count = new LongVbl.Sum (0);

    // Run one CPU thread for each GPU on the node.
    parallelDo (Gpu.allowedDeviceCount(), new Section()
    {
        public void run() throws Exception
        {
            // Set up per-thread counter variable.
            LongVbl thrCount = threadLocal (count);

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

            // Set up GPU counter variable.
            Module module = gpu.getModule
            ("edu/rit/gpu/example/PiGpu.cubin");
            GpuLongVbl devCount = module.getLongVbl ("devCount");

            // Determine how many of the N points this thread will
            // compute.
            long thrN = LongChunk.partition (0, N - 1, threads(),
            rank()) .length() .longval();

            // Generate thrN random points in the unit square,
            // count how many are in the unit circle.
            devCount.item = 0;
            devCount.hostToDev();
            PiKernel kernel = module.getKernel (PiKernel.class);
            kernel.setBlockDim (1024);
            kernel.setGridDim (gpu.getMultiprocessorCount());
            kernel.computeRandomPoints (seed + 1000000L*rank(),
            thrN);

            // Get per-thread count, automatically reduced into
            // global count.
            devCount.devToHost();
            thrCount.item = devCount.item;
        }
    });

    // Print results.
    System.out.printf ("pi = 4*%d/%d = %.9f%n", count.item, N,
    4.0*count.item/N);
}

Listing 24.1. PiGpu2.java (part 2)
“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 $N = 100$ billion, $200$ billion, $500$ billion, one trillion, and two 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:

```bash
$ java pj2 debug=makeSpan gpus=1 edu.rit.gpu.example.PiGpu2 142857 100000000000
$ java pj2 debug=makeSpan gpus=2 edu.rit.gpu.example.PiGpu2 142857 200000000000
```

Figure 24.2 plots the running times and efficiencies I observed. The fitted running time model is

$$T = 1.28 + 9.39 \times 10^{-15} N + 0.180 K + 1.90 \times 10^{-10} N/K$$  \hspace{1cm} (24.1)$$

The program’s sequential portion takes 1.28 seconds (plus a negligible term proportional to $N$); this yields a sequential fraction ranging from 0.06 for the smallest problem size down to 0.003 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 $1.90 \times 10^{-10}$ seconds, for a computation rate of 5.26 billion darts per second.

Below are the estimates for $\pi$ calculated by the program for various problem sizes $N$. Note how the 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>$4 \times 10^{11}$</td>
<td>3.141591085</td>
<td>$4.99 \times 10^{-7}$</td>
</tr>
<tr>
<td>$8 \times 10^{11}$</td>
<td>3.141589897</td>
<td>$8.77 \times 10^{-7}$</td>
</tr>
<tr>
<td>$2 \times 10^{12}$</td>
<td>3.141591885</td>
<td>$2.45 \times 10^{-7}$</td>
</tr>
<tr>
<td>$4 \times 10^{12}$</td>
<td>3.141593160</td>
<td>$1.61 \times 10^{-7}$</td>
</tr>
<tr>
<td>$8 \times 10^{12}$</td>
<td>3.141593379</td>
<td>$2.31 \times 10^{-7}$</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.
// Specify that this task requires one core. (Parallel team
// threads will share the core.)
protected static int coresRequired()
{
    return 1;
}

// Specify that this task requires all GPU accelerators on the
// node.
protected static int gpusRequired()
{
    return ALL_GPUS;
}
• 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.