Chapter 23
GPU Parallel Reduction

- 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
Let’s try a more substantial GPU parallel program: our old friend, the \( \pi \) estimating program. Recall that the program throws \( N \) darts that land at random locations in the unit square, counts how many darts \( C \) fall within the inscribed circle quadrant, and estimates \( \pi \approx 4C/N \). In a parallel version, the loop iterations (darts) are partitioned among \( K \) threads; each thread throws \( N/K \) darts and increments its own semifinal count; at the end, the semifinal counts are sum-reduced together, yielding the final total count from which to compute the result.

I will use this same approach for the GPU parallel \( \pi \) estimating program. But I have to be careful about how I partition the computation and about how I perform the reduction.

In the GPU parallel outer product program in Chapter 21, I set up the computational grid to match the structure of the output matrix. There was one thread for each matrix element. The threads were arranged into two-dimensional blocks, and the blocks were arranged into a two-dimensional grid, mirroring the two-dimensional array (matrix) being computed. Each thread computed one and only one matrix element.

In contrast, the GPU parallel \( \pi \) estimating program has no such structure. The computation’s output is a single value, \( C \), the total number of darts that landed inside the circle quadrant. So other considerations will dictate how I set up the computational grid.

I want to use the full parallelism available on the GPU. So the grid will be one-dimensional, and the number of blocks in the grid will be the same as the number of multiprocessors in the GPU. (As we will see, the Parallel Java 2 Library lets you query how many multiprocessors the GPU has.) Thus, when I run the kernel, each block in the grid will be assigned to its own multiprocessor, and all the blocks will run fully in parallel. Each block in turn will be one-dimensional, and the number of threads in the block will be the maximum possible. The maximum threads per block is a characteristic of the CUDA compute capability; for my compute-capability-2.0 GPU, that’s 1024 threads. So if the GPU has \( K \) multiprocessors, the grid will have 1024\( \cdot \)\( K \) threads; and each thread will do \( N/(1024\cdot K) \) loop iterations.

The kernel’s output is the final total count \( C \). This will be stored in the GPU’s global memory, so the CPU can retrieve it. Following the parallel reduction pattern, each thread in the kernel will maintain its own per-thread counter. Reducing the per-thread counters together will take place in two
package edu.rit.gpu.example;
import edu.rit.gpu.Kernel;
import edu.rit.gpu.Gpu;
import edu.rit.gpu.GpuLongVbl;
import edu.rit.gpu.Module;
import edu.rit.pj2.Task;
public class PiGpu
    extends Task
{
    // Kernel function interface.
    private static interface PiKernel
        extends Kernel
    {
        public void computeRandomPoints
            (long seed,
             long N);
    }

    // Task main program.
    public void main
        (String[] args)
        throws Exception
    {
        // Validate command line arguments.
        if (args.length != 2) usage();
        long seed = Long.parseLong (args[0]);
        long N = Long.parseLong (args[1]);

        // Initialize 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 count = module.getLongVbl ("devCount");

        // Generate n random points in the unit square, count how many
        // are in the unit circle.
        count.item = 0;
        count.hostToDevice();
        PiKernel kernel = module.getKernel (PiKernel.class);
        kernel.setBlockDim (1024);
        kernel.setGridDim (gpu.getMultiprocessorCount());
        kernel.computeRandomPoints (seed, N);

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

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 " +
            "edu.rit.gpu.example.PiGpu <seed> <N>");
        System.err.println ("<seed> = Random seed");
    }

    Listing 23.1. PiGpu.java (part 1)
stages. Within each block, the threads in the block will use the multiprocessor’s fast shared memory to add the per-thread counters together, yielding one semifinal count in shared memory for each block. The per-block counts will in turn be added together, yielding the final count in global memory.

Listing 23.1 is the Java main program, class PiGpu. The main program doesn’t do much; it sets up the GPU kernel, runs the kernel, retrieves the kernel’s result, and prints the answer. All the real work happens in parallel in the kernel.

Class PiGpu begins by declaring the GPU kernel interface (lines 11–17). The kernel function is named `computeRandomPoints()`, and it takes two arguments, the seed for the pseudorandom number generator (PRNG) and the number of darts to throw, \( N \), both of Java type `long`.

The main program needs to access the variable in GPU global memory that will end up holding the kernel’s result. To gain access to this variable, the main program first obtains a GPU object (lines 30–31) and obtains the compiled GPU module (lines 34–35). The main program then calls the `getLongVbl()` method on the module object, specifying the name of a certain variable in the GPU module, `devCount`. (We’ll see the GPU variable’s declaration when we look at the code for the kernel.) The `getLongVbl()` method returns a `GpuLongVbl` object, which is stored in the main program’s `count` variable. The `count` object has a field of type `long`; this field (on the CPU) mirrors the `devCount` variable (on the GPU).

The main program is now ready to run the computation. First, it initializes the count to 0 and uploads the CPU variable to the GPU variable (lines 40–41); this initializes the count on the GPU. One-time initializations like this have to be done by the CPU main program, not by the GPU kernel function. If this were done in the kernel function, every thread in the grid would initialize the count, which would be incorrect. Next, the main program gets the kernel object; configures the grid to have 1024 threads per block and \( K \) blocks, where \( K \) is the number of multiprocessors on the GPU as returned by the `getMultiprocessorCount()` method; and calls the kernel method (lines 42–45).

When the kernel method returns, the computation has finished, and the
count on the GPU has been set to \( C \), the total number of darts that landed inside the circle quadrant. The main program downloads the count object from the GPU to the CPU (line 48). The count object’s item field now contains \( C \), and the program uses that to print the results (lines 49–50).

To finish off the Java code, the PiGpu task class overrides the coresRequired() method (lines 64–67) to state that the task requires just one CPU core—because there is no multithreaded parallelism on the CPU side. The PiGpu task class also overrides the gpusRequired() method (lines 70–73) to state that the task requires one GPU accelerator; if this is omitted, the Tracker would assume the task requires no GPUs, the Tracker might assign the task to a node that has no GPU, and the program would fail.

Turning to the GPU side of the program, Listing 23.2 has the GPU kernel function as well as several variable declarations, written in C with CUDA. Line 1 includes the Random.cu source file, which defines a typedef prng_t for a PRNG as well as several functions for seeding the generator and extracting random numbers. (I’m not going to describe what’s inside Random.cu.) Line 4 states that the grid is assumed to have \( NT = 1024 \) threads per block. (It’s up to the main program to configure the grid properly.)

Line 7 declares the devCount variable, which will hold the final count \( C \). The CUDA keyword __device__ (“underscore underscore device underscore underscore underscore”) says that the variable is to be located in the GPU’s global memory. The devCount variable is located in the global memory for two reasons: so that all the threads in all the blocks can access the count for purposes of reduction, and so that the count can be mirrored in the CPU main program.

Line 10 declares an array variable named shrCount. In each block, this array will hold the per-thread counts for purposes of reduction. The CUDA

Listing 23.1. PiGpu.java (part 2)
keyword __shared__ ("underscore underscore shared underscore underscore") says that the array is to be located in the multiprocessor’s fast shared memory. Because each multiprocessor has its own separate shared memory, each block in the grid gets its own separate shrCount array. The array needs only as many elements as there are threads in a block, namely NT elements— not as many elements as there are threads in the whole grid. The shrCount array is located in the shared memory so that all the threads in one block can access the array during the reduction. There is no need (and in fact no way) for threads in one block to access the shrCount array in a different block.

Lines 17–19 declare the kernel function, computeRandomPoints(). The arguments are the PRNG’s seed and the number of loop iterations N, both of type unsigned long long int in C. (In the Java main program, the corresponding kernel function arguments are of type long; see PiGpu.java lines 14–16.)

The kernel function begins by determining several quantities (lines 26–28). thr is the index of the currently executing thread within the block, obtained from the threadIdx.x pseudo-variable. size is the number of threads in the entire grid, which is the number of blocks in the grid (the gridDim.x pseudo-variable) times the number of threads per block. The N loop iterations need to be partitioned among this many threads. rank is the unique rank of the currently executing thread within the entire grid, computed from the block’s x coordinate in the grid and the thread’s x coordinate in the block.

When each thread executes the kernel function, the thread gets its own per-thread counter in the count local variable and its own per-thread PRNG of type prng_t in the prng local variable. These variables are declared on lines 22–23 and initialized on lines 31–32. The PRNG is initialized differently in each thread (seed plus rank); this ensures that each thread will generate different random dart locations. The per-thread counter is initialized to 0. We’ve now seen all the counter variables that will be involved in the reduction: the per-thread counts (line 22), the per-block counts in shared memory (line 10), and the final count in global memory (line 7).

The kernel is now ready to partition the loop iterations among the threads in the grid, and have each thread execute a subset of the loop iterations. The for loop on line 35 partitions the iterations using what amounts to a leapfrog schedule. Each thread’s initial loop index is the same as the thread’s rank. Going to the next iteration, each thread increases its loop index by the total number of threads in the grid. Suppose the grid consists of 14 blocks of 1024 threads each, for a total of 14336 threads, and suppose N is 100000. Then thread rank 0 performs loop indexes 0, 14336, 28672, 43008, 57344, 71680, and 86016; thread rank 1 performs loop indexes 1, 14337, 28673, 43009, 57345, 71681, and 86017; and so on. Each thread ends up performing an equal (or nearly equal) number of loop iterations; and because the running time of each loop iteration is the same, the load is balanced. In the loop body
// Number of threads per block.
#define NT 1024

// Overall counter variable in global memory.
__device__ unsigned long long int devCount;

// Per-thread counter variables in shared memory.
__shared__ unsigned long long int shrCount [NT];

// Device kernel to compute random points.
// Called with a one-dimensional grid of one-dimensional blocks, NB
// blocks, NT threads per block. NT must be a power of 2.
// seed Pseudorandom number generator seed.
// N Number of points.
extern "C" __global__ void computeRandomPoints
(unsigned long long int seed,
 unsigned long long int N)
{
    int thr, size, rank;
    unsigned long long int count;
    prng_t prng;

    // Determine number of threads and this thread's rank.
    thr = threadIdx.x;
    size = blockDim.x*NT;
    rank = blockIdx.x*NT + thr;

    // Initialize per-thread prng and count.
    prngSetSeed (&prng, seed + rank);
    count = 0;

    // Compute random points.
    for (unsigned long long int i = rank; i < N; i += size)
    {
        double x = prngNextDouble (&prng);
        double y = prngNextDouble (&prng);
        if (x*x + y*y <= 1.0) ++ count;
    }

    // Shared memory parallel reduction within thread block.
    shrCount[thr] = count;
    __syncthreads();
    for (int i = NT/2; i > 0; i >>= 1)
    {
        if (thr < i)
            shrCount[thr] += shrCount[thr+i];
    }
    __syncthreads();

    // Atomic reduction into overall counter.
    if (thr == 0)
        atomicAdd (&devCount, shrCount[0]);
}

Listing 23.2. PiGpu.cu
(lines 37–39), a random \((x, y)\) dart location is extracted from the per-thread PRNG, and the per-thread counter is incremented if the dart falls inside the circle quadrant. Because each thread is updating its own per-thread PRNG and counter, no thread synchronization is needed at this point.

After finishing the loop iterations, the kernel is ready to perform the reduction. The first stage of reduction is to add up all the per-thread counts within the block. This is done in parallel using a reduction tree, like the one we encountered in the multicore parallel \(\pi\) estimating program in Chapter 4 (Figure 23.1). There, the code to do the reduction was hidden inside the reduction variable classes in the Parallel Java 2 Library, and the parallel for loop did the reduction automatically. CUDA has nothing analogous to a reduction variable class, so here I have to code the reduction tree myself (lines 43–50). The reduction makes use of the multiprocessor’s fast shared memory, which all the threads in the block can access.

Each thread first stores its per-thread counter variable \((\text{count})\) into one of the elements of the shared memory array \((\text{shrCount}[\text{thr}])\); the array index is the same as the thread’s index within the block. Before proceeding, each thread has to wait until all the threads in the block have stored their per-thread counters. In other words, at this point the threads need to wait at a barrier. The \text{__syncthreads()} function (“underscore underscore syncthreads”), a special CUDA function, performs the barrier synchronization. The barrier is implemented in hardware and is very fast.

Now the shared memory parallel reduction can commence. The reduction proceeds in a number of rounds; each iteration of the loop on lines 45–50 is one round. The loop index \(i\) is the stride for the current round; the stride starts at half the number of threads in the block, namely 512; the stride is halved at each loop iteration; so the strides for the rounds are 512, 256, 128, \ldots, 4, 2, 1; when the stride hits 0, the loop stops. The stride is halved by right-shifting it one bit position \((i >>= 1)\), which is equivalent to dividing it by 2. During each round, each thread whose index is less than the stride adds its own array element \((\text{shrCount}[\text{thr}])\) together with the array element whose index is the stride higher \((\text{shrCount}[\text{thr}+i])\), and stores the result back into its own array element. These additions are all done in parallel. Again, after the additions there is a barrier, to ensure that all threads have completed their additions before proceeding to the next round of reduction.

Thus, in the first round with a stride of 512, thread 0 adds array element 512 into array element 0, thread 1 adds element 513 into element 1, \ldots, thread 511 adds element 1023 into element 511; threads 512–1023 do nothing. In the second round with a stride of 256, thread 0 adds array element 256 into array element 0, thread 1 adds element 257 into element 1, \ldots, thread 255 adds element 511 into element 255; threads 256–1023 do nothing. And so on. This implements exactly the reduction tree shown in Figure 23.1 (extended upwards to accommodate 1024 elements). Because each thread is up-
dating a different array element, the threads do not need to synchronize with each other during the additions; the threads only need to synchronize at the barrier at the end of each round; this minimizes the synchronization overhead. When the reduction loop exits, the per-block count—the sum of all the per-thread counts—ends up stored in shrCount[0].

This shared memory parallel reduction code is specifically designed for the case where the number of threads in the block is a power of 2. If, in some other program that does reduction, the number of threads is not a power of 2, the code would have to be altered a little. The initial stride would be the smallest power of 2 greater than or equal to half the number of threads in the block. The if statement in the loop body would check that the thread index is less than the stride and that the thread index plus the stride is less than the number of threads in the block.

Now the second stage of reduction happens. Each thread must add its own per-block count into the global count variable, devCount. This is done by a single thread in each block, namely thread rank 0 (line 53). Multiple threads, one in each block, are attempting to update the global variable concurrently; therefore, these threads do need to synchronize with each other. I get the necessary synchronization by using an atomic operation to do the up-

![Figure 23.1. Sum-reduce parallel reduction tree](image-url)
date (line 54). The \texttt{atomicAdd()} function, a special CUDA function, adds its second argument (the per-block count, \texttt{shrCount[0]}) to the variable referred to by the first argument (the global count, \texttt{devCount}) and stores the result back into that variable. Furthermore, the \texttt{atomicAdd()} function ensures that only one thread at a time performs the operation. “Atomic” means “not divisible” or “not interruptible”—a thread doing an atomic add on a variable will not be interrupted by another thread trying to do an atomic add on that variable until the first add is complete. Various atomic operations are available in different CUDA compute capabilities; \texttt{atomicAdd()} of unsigned long long integer variables is supported in compute capability 2.0 GPUs like mine. (See the CUDA documentation for a list of the supported atomic operations.) This is another reason to verify the GPU’s compute capability before proceeding with the program.

When all the threads in all the blocks in the grid have finished executing the kernel function, the global \texttt{devCount} variable contains the sum of all the per-block counts, which is the sum of all the per-thread counts, which is the total number of darts that landed inside the circle quadrant, \( C \). At this point, back in the CPU main program, the kernel method returns, and the CPU main program downloads \( C \) from the GPU and prints the result. Done!

I compared the total running times of the single-threaded CPU-only version of the \( \pi \) estimating program (\texttt{PiSeq}) with the GPU parallel version (\texttt{PiGpu}) on the \texttt{kraken} machine for various problem sizes, using commands like this:

\$ java pj2 edu.rit.pj2.example.PiSeq 142857 1000000
\$ java pj2 edu.rit.gpu.example.PiGpu 142857 1000000

Here are the running times I observed, in milliseconds. I also listed the estimate for \( \pi \) that the \texttt{PiGpu} program computed.

\begin{center}
\begin{tabular}{cccc}
\textbf{\( N \)} & \textbf{CPU} & \textbf{GPU} & \textbf{Ratio} & \textbf{\( \pi \) Estimate} \\
\hline
1\times10^6 & 50 & 96 & 0.5 & 3.142316000 \\
1\times10^7 & 181 & 87 & 2.1 & 3.141395200 \\
1\times10^8 & 1233 & 116 & 10.6 & 3.141559320 \\
1\times10^9 & 11415 & 277 & 41.2 & 3.141609640 \\
1\times10^{10} & 113127 & 1985 & 57.0 & 3.141612167 \\
1\times10^{11} & 1129808 & 19088 & 59.2 & 3.141598614 \\
1\times10^{12} & 11300994 & 190098 & 59.4 & 3.141594103 \\
\end{tabular}
\end{center}

Once the problem size becomes large enough that the fixed overhead no longer dominates the running time, the GPU program is 60 times faster than the CPU program on \texttt{kraken}. It took the CPU over three hours to compute one trillion darts. It took the GPU only three minutes. That’s a computation rate of over five billion darts per second. No human darts champion in any pub on the globe ever threw darts that fast.
Under the Hood

The GPU parallel \( \pi \) estimating program’s \texttt{main()} method, which runs on the CPU, declares a variable named \texttt{count} (PiGpu.java line 36). The kernel declares a variable named \texttt{devCount} (PiGpu.cu line 7) located in the GPU’s global memory. The \texttt{count} variable on the CPU “mirrors” the \texttt{devCount} variable on the GPU. Let’s look more closely at how mirrored variables work.

The \texttt{count} object was created by calling the \texttt{getLongVbl("devCount")} method on the module object obtained from the GPU object. As we have seen, \texttt{count} is an instance of class \texttt{GpuLongVbl}, with a public field named \texttt{item} of type \texttt{long}. The \texttt{getLongVbl()} method calls a CUDA function to get the address in the GPU’s global memory of the GPU variable named \texttt{devCount}. The address of \texttt{devCount} is stored in a private field of the \texttt{count} object. When the main program calls \texttt{count.hostToDev()} (line 41), the \texttt{hostToDev()} method in turn calls a CUDA function to copy eight bytes—the size of a \texttt{long} variable—from the address of the count object’s \texttt{item} field on the CPU to the address of \texttt{devCount} on the GPU. When the main program calls \texttt{count.devToHost()} (line 48), the \texttt{devToHost()} method in turn calls a CUDA function to copy eight bytes from the address of \texttt{devCount} on the GPU to the address of the count object’s \texttt{item} field on the CPU. In this way, I can keep the variable’s value in sync between the CPU and the GPU.

The \texttt{GpuDoubleArray} and \texttt{GpuDoubleMatrix} classes—which the outer product program in Chapter 22 used—work the same way, as do all of the GPU variable classes in package \texttt{edu.rit.gpu}. Each class stores the variable’s GPU global memory address in a private field. The \texttt{hostToDev()} and \texttt{devToHost()} methods call CUDA functions to copy the variable’s value from one side to the other. The Java methods use the Java Native Interface (JNI) to call the non-Java CUDA functions.

However, copying the mirrored variable does not happen automatically. You have to call the \texttt{hostToDev()} or \texttt{devToHost()} method explicitly. Why? Because it’s not necessarily true that every change to the variable’s value on one side needs to be immediately reflected on the other side. (We’ll see an example of this in Chapter 25.)

Here’s the command that compiles the PiGpu.cu source file, producing the PiGpu.cubin CUDA binary file:

```
$ nvcc -cubin -arch compute_20 -code sm_20 \n  --ptxas-options="-v" -o PiGpu.cubin PiGpu.cu
ptxas info: 8 bytes gmem, 4 bytes cmem[14]
ptxas info: Compiling entry function 'computeRandomPoints' for 'sm_20'
ptxas info: Function properties for computeRandomPoints
  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info: Used 20 registers, 8192 bytes smem, 48 bytes cmem[0], 24 bytes cmem[16]
```
The kernel function uses 20 registers, and there are 1,024 threads per block, so each block requires 20,480 registers; that fits within the 32,768 registers available in each multiprocessor. The `shrCount` array occupies 8,192 bytes of shared memory (1,024 array elements of 8 bytes each); that fits within the 48 kilobytes of shared memory available in each multiprocessor. So this kernel will have no trouble running on my compute-capability-2.0 GPU.

**Points to Remember**

- Configure the kernel grid to match the problem structure.
- If the problem has no inherent two-dimensional or three-dimensional structure, go with a one-dimensional grid of one-dimensional blocks.
- Use the multiprocessor’s shared memory to do parallel reduction within a block, following the reduction tree pattern.
- Using the `__syncthreads()` function, synchronize the threads at a barrier immediately before beginning the reduction tree and at the end of each round of the reduction tree.
- Use the GPU’s global memory to store the final result of the reduction.
- Only one thread in the block should do the reduction into the global memory variable.
- Use CUDA’s atomic operations to synchronize the threads in different blocks that are updating the global memory variable.