CHAPTER

Parallel programming and computational thinking 17

CHAPTER OUTLINE

We have so far concentrated on the practical experience of parallel programming, which consists of features of the CUDA programming model, performance and numerical considerations, parallel patterns, and application case studies. We will now switch gears to more abstract concepts. We will first generalize parallel programming into a computational thinking process that decomposes a domain problem into well-defined, coordinated work units that can each be realized with efficient numerical methods and well-studied algorithms. A programmer with strong computational thinking skills not only analyzes but also transforms the structure of a domain problem: which parts are inherently serial, which parts are amenable to high-performance parallel execution, and the domain-specific tradeoffs involved in moving parts from the former category to the latter. With good problem decomposition, the programmer can select and implement algorithms that achieve an appropriate compromise between parallelism, computational efficiency, and memory bandwidth consumption. A strong combination of domain knowledge and computational thinking skills is often needed for creating successful computational solutions to challenging domain problems. This chapter will give the reader more insight into parallel programming and computational thinking in general.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00001-7

17.1 GOALS OF PARALLEL COMPUTING

Before we discuss the fundamental concepts of parallel programming, it is important for us to first review the three main reasons why people pursue parallel computing. The first goal is to solve a given problem in less time. For example, an investment firm may need to run a financial portfolio scenario risk analysis package on all its portfolios during after-trading hours. Such an analysis may require 200 hours on a sequential computer. However, the portfolio management process may require that analysis be completed in 4 hours in order to be in time for major decisions based on that information. Using parallel computing may speed up the analysis and allow it to complete within the required time window.

The second goal of using parallel computing is to solve bigger problems within a given amount of time. In our financial portfolio analysis example, the investment firm may be able to run the portfolio scenario risk analysis on its current portfolio within a given time window using sequential computing. However, the firm is planning on expanding the number of holdings in its portfolio. The enlarged problem size would cause the running time of analysis under sequential computation to exceed the time window. Parallel computing that reduces the running time of the bigger problem size can help accommodate the planned expansion to the portfolio.

The third goal of using parallel computing is to achieve better solutions for a given problem and a given amount of time. The investment firm may have been using an approximate model in its portfolio scenario risk analysis. Using a more accurate model may increase the computational complexity and increase the running time on a sequential computer beyond the allowed window. For example, a more accurate model may require consideration of interactions between more types of risk factors using a more numerically complex formula. Parallel computing that reduces the running time of the more accurate model may complete the analysis within the allowed time window.

In practice, parallel computing may be driven by a combination of these three goals.

It should be clear from our discussion that parallel computing is primarily motivated by increased speed. The first goal is achieved by increased speed in running the existing model on the current problem size. The second goal is achieved by increased speed in running the existing model on a larger problem size. The third goal is achieved by increased speed in running a more complex model on the current problem size. Obviously, the increased speed through parallel computing can be used to achieve a combination of these goals. For example, parallel computing can reduce the run time of a more complex model on a larger problem size.

It should also be clear from our discussion that applications that are good candidates for parallel computing typically involve large problem sizes and high modeling complexity. That is, these applications process a large amount of data, require a lot of computation in each iteration, and/or perform many iterations on the data, or both. Applications that do not process large problem sizes or incur high modeling

complexity tend to complete within a small amount of time and do not offer much motivation for increased speed. In order for a problem to be solved with parallel computing, the problem must be formulated in such a way that the large problem can be decomposed into sub-problems that can be safely solved at the same time. Under such formulation and decomposition, the programmer writes code and organizes data to solve these sub-problems concurrently.

In Chapters 14 and 15, Application case study—non-Cartesian MRI and Application case study—molecular visualization and analysis, we presented two problems that are good candidates for parallel computing. The magnetic resonance imaging (MRI) reconstruction problem processes a large amount of *k*-space sample data. Each *k*-space sample data is also used many times for calculating its contributions to the reconstructed voxel data. For a reasonably high resolution reconstruction, each sample data is used a very large number of times. We showed that a good decomposition of the F^HD problem in MRI reconstruction forms sub-problems that each calculate the value of an F^HD element. All of these subproblems can be solved in parallel with each other. We use a massive number of CUDA threads to solve these sub-problems.

Similarly, the electrostatic potential calculation problem involves the calculation of the contribution of a large number of atoms to the potential energy of a large number of grid points. Fig. 15.10 further shows that the electrostatic potential calculation problem should be solved with a massively parallel CUDA device only if there are 400 or more atoms. In reality, this is not a very restricting requirement. A realistic molecular system model typically involves at least hundreds of thousands of atoms and millions of energy grid points. The electrostatic charge information of each atom is used many times in calculating its contributions to the energy grid points. We showed that a good decomposition of the electrostatic potential calculation problem forms sub-problems that each calculates the energy value of a grid point. All the sub-problems can be solved in parallel with each other. We use a massive number of CUDA threads to solve these sub-problems.

The process of parallel programming can typically be divided into four steps: problem decomposition, algorithm selection, implementation in a language, and performance tuning. The last two steps were the focus of previous chapters. In the next two sections, we will discuss the first two steps with more generality as well as depth.

17.2 PROBLEM DECOMPOSITION

Finding parallelism in large computational problems is often conceptually simple but can be challenging in practice. The key is to identify the work to be performed by each unit of parallel execution, which is a thread, so that the inherent parallelism of the problem is well utilized. For example, in the electrostatic potential map calculation problem, it is clear that all atoms can be processed in parallel and all energy grid points can be calculated in parallel. However, one must take care when decomposing the calculation work into units of parallel execution, which will be referred to as

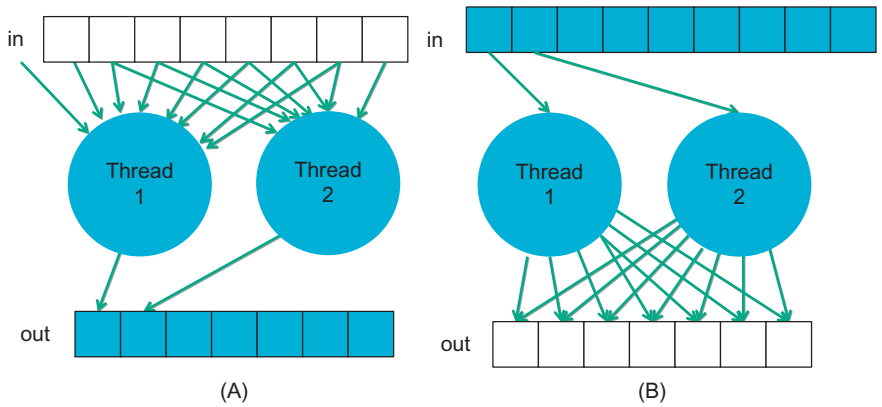

FIGURE 17.1

(A) Gather and (B) scatter based thread arrangements.

threading arrangement. As we discussed in Section 15.2, the decomposition of the electrostatic potential map calculation problem can be atom-centric or grid-centric. In an atom-centric threading arrangement, each thread is responsible for calculating the effect of one atom on all grid points. In contrast, a grid-centric threading arrangement uses each thread to calculate the effect of all atoms on a grid point.

While both threading arrangements lead to similar levels of parallel execution and same execution results, they can exhibit very different performance in a given hardware system. The grid-centric arrangement has a memory access behavior called *gather*, where each thread gathers or collects the effect of input atoms into a grid point. Fig. 17.1A illustrates the gather access behavior. Gather is a desirable thread arrangement in CUDA devices because the threads can accumulate their results in their private registers. Also, multiple threads share input atom values, and can effectively use constant memory caching or shared memory to conserve global memory bandwidth.

The atom-centric arrangement, on the other hand, exhibits a memory access behavior called *scatter*, where each thread scatters or distributes the effect of an atom into grid points. The scatter behavior is illustrated in Fig. 17.1B. This is an undesirable arrangement in CUDA devices because the multiple threads can write into the same grid point at the same time. The grid points must be stored in a memory that can be written by all the threads involved. Atomic operations must be used to prevent race conditions and loss of value during simultaneous writes to a grid point by multiple threads. These atomic operations are typically slower than the register accesses used in the atom-centric arrangement. Understanding the behavior of the threading arrangement and the limitations of hardware allows a parallel programmer to steer toward the more desired gather-based arrangement.

A real application often consists of multiple modules that work together. Fig. 17.2 shows an overview of major modules of a molecular dynamics application. For each

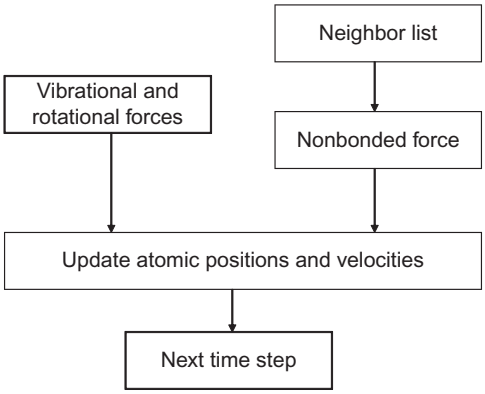

FIGURE 17.2

Major tasks of a molecular dynamics application.

atom in the system, the application needs to calculate the various forms of forces, e.g., vibrational, rotational, and nonbonded, that are exerted on the atom. Each form of force is calculated by a different method. At the high level, a programmer needs to decide how the work is organized. Note that the amount of work can vary dramatically between these modules. The nonbonded force calculation typically involves interactions among many atoms and incurs much more calculation than the vibrational and rotational forces. Therefore, these modules tend to be realized as separate passes over the force data structure.

The programmer needs to decide if each pass is worth implementing in a CUDA device. For example, he/she may decide that the vibrational and rotational force calculations do not involve sufficient amount of work to warrant execution on a device. Such a decision would lead to a CUDA program that launches a kernel that calculates nonbonded force fields for all the atoms while continuing to calculate the vibrational and rotational forces for the atoms on the host. The module that updates atomic positions and velocities may also run on the host. It first combines the vibrational and rotational forces from the host and the nonbonded forces from the device. It then uses the combined forces to calculate the new atomic positions and velocities.

The portion of work done by the device will ultimately decide the application level speedup achieved by parallelization. For example, assume that the nonbonded force calculation accounts for 95% of the original sequential execution time and it is accelerated by 100× using a CUDA device. Further assume that the rest of the application remains on the host and receives no speedup. The application level speedup is $1/(5\% + 95\%/100) = 1/(5\% + 0.95\%) = 1/(5.95\%) = 17\times$. This is a demonstration of Amdahl's Law: the application speedup due to parallel computing is limited by the sequential portion of the application. In this case, even though the sequential portion of the application is quite small (5%), it limits the application level speedup to 17× even though the nonbonded force calculation has a speedup of 100×. This example

illustrates a major challenge in decomposing large applications: the accumulated execution time of small activities that are not worth parallel execution on a CUDA device can become a limiting factor in the speedup seen by the end users.

Amdahl's Law often motivates task-level parallelization. Although some of these smaller activities do not warrant fine-grained massive parallel execution, it may be desirable to execute some of these activities in parallel with each other when the data set is large enough. This could be achieved by using a multi-core host to execute such tasks in parallel. Alternatively, we could try to simultaneously execute multiple small kernels, each corresponding to one task. The previous CUDA devices did not support such parallelism but the new generation devices such as Kepler do.

An alternative approach to reducing the effect of sequential tasks is to exploit data parallelism in a hierarchical manner. For example, in a Message Passing Interface (MPI) [MPI 2009] implementation, a molecular dynamics application would typically distribute large chunks of the spatial grids and their associated atoms to nodes of a networked computing cluster. By using the host of each node to calculate the vibrational and rotational force for its chunk of atoms, we can take advantage of multiple host CPUs and achieve speedup for these lesser modules. Each node can use a CUDA device to calculate the nonbonded force at higher levels of speedup. The nodes will need to exchange data to accommodate forces that go across chunks and atoms that move across chunk boundaries. We will discuss more details of joint MPI-CUDA programming in Chapter 18, Programming a heterogeneous cluster. The main point here is that MPI and CUDA can be used in a complementary way in applications to jointly achieve a higher level of speed with large data sets.

17.3 ALGORITHM SELECTION

An algorithm is a step-by-step procedure where each step is precisely stated and can be carried out by a computer. An algorithm must exhibit three essential properties: definiteness, effective computability, and finiteness. Definiteness refers to the notion that each step is precisely stated; there is no room for ambiguity as to what is to be done. Effective computability refers to the fact that each step can be carried out by a computer. Finiteness means that the algorithm must be guaranteed to terminate.

Given a problem, we can typically come up with multiple algorithms to solve the problem. Some require fewer steps of computation than others; some allow a higher degree of parallel execution than others; some have better numerical stability than others, and some consume less memory bandwidth than others. Unfortunately, there is often not a single algorithm that is better than others in all the four aspects. Given a problem and a decomposition strategy, a parallel programmer often needs to select an algorithm that achieves the best compromise for a given hardware system.

In our matrix–matrix multiplication example, we decided to decompose the problem by having each thread compute the dot product for an output element. Given this decomposition, we presented two different algorithms. The algorithm in Section 4.2 is a straightforward algorithm where every thread simply performs

an entire dot product. Although the algorithm fully utilizes the parallelism available in the decomposition, it consumes too much global memory bandwidth. In Section 4.4, we introduced tiling, an important algorithm strategy for conserving memory bandwidth. Note that the tiled algorithm partitions the dot products into phases. All threads involved in a tile must synchronize with each other so that they can collaboratively load the tile of input data into the shared memory and collectively utilize the loaded data before they move on to the next phase. As we showed in Fig. 4.16, the tiled algorithm requires each thread to execute more statements and incur more overhead in indexing the input arrays than the original algorithm. However, it runs much faster because it consumes much less global memory bandwidth. In general, tiling is one of the most important algorithm strategies for matrix applications to achieve high performance.

As we demonstrated in Sections 5.5 and 15.3, we can systematically merge threads to achieve a higher level of instruction and memory access efficiency. In Section 5.5, threads that handle the same columns of neighboring tiles are combined into a new thread. This allows the new thread to access each `M` element only once while calculating multiple dot products, reducing the number of address calculations and memory load instructions executed. It also further reduces the consumption of global memory bandwidth. The same technique, when applied to the DCS kernel in electrostatic potential calculation, further reduces the number of distance calculations while achieving a similar reduction in address calculations and memory load instructions.

One can often invent even more aggressive algorithm strategies. An important algorithm strategy, referred to as *cutoff binning*, can significantly improve the execution efficiency of grid or particle algorithms by sacrificing a small amount of accuracy. This is based on the observation that many grid or particle calculation problems are based on physical laws where numerical contributions from particles or samples that are far away from a grid point or particle can be collectively treated with an implicit method at much lower computational complexity. This is illustrated for the electrostatic potential calculation in Fig. 17.3. Fig. 17.3A shows the direct summation algorithms discussed in Chapter 15, Application case study—molecular visualization and analysis. Each grid point receives contributions from all atoms. While this is a very parallel approach and achieves excellent speedup over CPU-only execution for moderate-sized energy grid systems, as we showed in Section 15.5, it does not scale well to very large energy-grid systems where the number of atoms increases proportional to the volume of the system. The amount of computation increases with the square of the volume. For large volume systems, such an increase makes the computation excessively long even for massively parallel devices.

In practice, we know that each grid point needs to receive contributions from atoms that are close to it. The atoms that are far away from a grid point will have negligible contribution to the energy value at the grid point because the contribution is inversely proportional to the distance. Fig. 17.3B illustrates this observation with a circle drawn around a grid point. The contributions to the grid point energy from atoms outside the circle (*maroon* (dark gray in print versions)) are negligible. If we can devise an algorithm where each grid point only receives contributions from

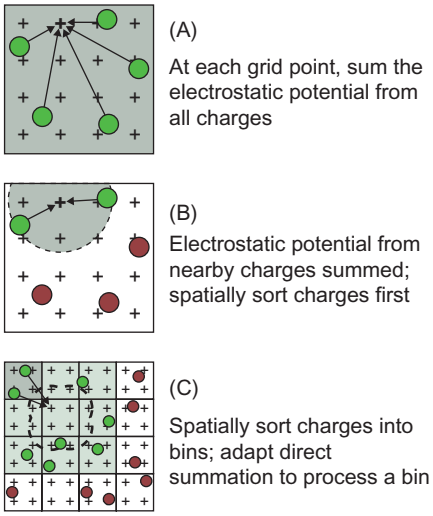

FIGURE 17.3

Cutoff summation algorithm. (A) Direct summation, (B) cutoff summation, and (C) cutoff summation using direct summation kernel.

atoms within a fixed radius of its coordinate (*green* (light gray in the print versions)), the computational complexity of the algorithm would be reduced, becoming linearly proportional to the volume of the system. This would make the computation time of algorithm linearly proportional to the volume of the system. Such algorithms have been used extensively in sequential computation.

In sequential computing, a simple cutoff algorithm handles one atom at a time. For each atom, the algorithm iterates through the grid points that fall within a radius of the atom's coordinate. This is a straightforward procedure since the grid points are in an array that can be easily indexed as a function of their coordinates. However, this simple procedure does not carry easily to parallel execution. The reason is what we discussed in Section 17.2: the atom-centric decomposition does not work well due to its scatter memory access behavior. However, as we discussed in Chapter 8, Parallel patterns – prefix-sum, it is important that a parallel algorithm matches the work efficiency of an efficient sequential algorithm.

Therefore, we need to find a cutoff binning algorithm based on the grid-centric decomposition: each thread calculates the energy value at one grid point. Fortunately, there is a well-known approach to adapting direct summation algorithm, such as the one in Fig. 15.9, into a cutoff binning algorithm. Rodrigues et al. presents such an algorithm for the electrostatic potential problem [RSH 2008].

The key idea of the algorithm is to first sort the input atoms into bins according to their coordinates. Each bin corresponds to a box in the grid space and it contains all atoms whose coordinates falls into the box. We define a "neighborhood" of bins for a grid point to be the collection of bins that contain all the atoms that can contribute to

the energy value of a grid point. If we have an efficient way of managing neighborhood bins for all grid points, we can calculate the energy value for a grid point by examining the neighborhood bins for the grid point. This is illustrated in Fig. 17.3C. Although Fig. 17.3 shows only one layer (2D) of bins that immediately surround that containing a grid point as its neighborhood, a real algorithm will typically have multiple layers (3D) of bins in a grid's neighborhood. In this algorithm, all threads iterate through their own neighborhood. They use their block and thread indices to identify the appropriate bins. Note that some of the atoms in the surrounding bins may not fall into the radius. Therefore, when processing an atom, all threads need to check if the atom falls into its radius. This can cause some control divergence among threads in a warp.

The main source of improvement in work efficiency comes from the fact that each thread now examines a much smaller set of atoms in a large grid system. This, however, makes constant memory much less attractive for holding the atoms. Since thread blocks will be accessing different neighborhoods, the limited-size constant memory will unlikely be able to hold all the atoms that are needed by all active thread blocks. This motivates the use of global memory to hold a much larger set of atoms. To mitigate the bandwidth consumption, threads in a block collaborate in loading the atom information in the common neighborhood into the shared memory. All threads then examine the atoms out of shared memory. The reader is referred to Rodrigues et al. [RSH 2008] for more details of this algorithm.

One subtle issue with binning is that bins may end up with different numbers of atoms. Since the atoms are statistically distributed in space, some bins may have lots of atoms and some bins may end up with no atom at all. In order to guarantee memory coalescing, it is important that all bins are of the same size and aligned at appropriate coalescing boundaries. In order to accommodate the bins with the largest number of atoms, we would need to make the size of all other bins the same size. This would require us to fill many bins with dummy atoms whose electrical charge is 0, which causes two negative effects. First, the dummy atoms still occupy global memory and shared memory storage. They also consume data transfer bandwidth to the device. Second, the dummy atoms extend the execution time of the thread blocks whose bins have few real atoms.

A well-known solution is to set the bin size at a reasonable level, typically much smaller than the largest possible number of atoms in a bin. The binning process maintains an overflow list. When processing an atom, if the atom's home bin is full, the atom is added to the overflow list instead. After the device completes a kernel, the result grid point energy values are transferred back to the host. The host executes a sequential cutoff algorithm on the atoms in the overflow list to complete the missing contributions from these overflow atoms. As long as the overflow atoms account for only a small percentage of the atoms, the additional sequential processing time of the overflow atoms is typically shorter than that of the device execution time. One can also design the kernel so that each kernel invocation calculates the energy values for a subvolume of grid points. After each kernel completes, the host launches the next kernel and processes the overflow atoms for the completed kernel. Thus, the host will be processing the overflow atoms while the device executes the next kernel. This

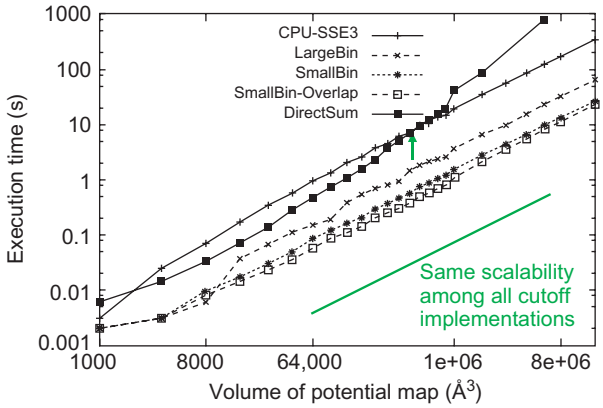

FIGURE 17.4

Scalability and performance of different algorithms for calculating electrostatic potential map.

approach can hide most, if not all, the delays in processing overflow atoms since it is done in parallel with the execution of the next kernel.

Fig. 17.4 shows a comparison of scalability and performance of the various electrostatic potential map algorithms. Note that the CPU–SSE3 curve is based on a sequential cutoff algorithm. For a map with small volumes, around 1000 Å^3 the host (CPU with SSE) executes faster than the DCS kernel shown in Fig. 17.4. This is because there is not enough work to fully utilize a CUDA device for such a small volume. However, for moderate volumes, between 2000 Å^3 and 500,000 Å^3, the Direct Summation kernel performs significantly better than the host due to its massive parallelism. As we anticipated, the Direct Summation kernel scales poorly when the volume size reaches about 1,000,000 Å^3, and runs longer than the sequential algorithm on the CPU! This is due to the fact that the algorithm complexity of the DCS kernel is higher than the sequential cut-off algorithm and thus the amount of work done by kernel grows much faster than that done by the sequential algorithm. For volume size larger than 1,000,000, the amount of work is so large that it swamps the hardware execution resources.

Fig. 17.4 also shows the running time of three binned cutoff algorithms. The `LargeBin` algorithm is a straightforward adaptation of the DCS kernel for cutoff. The kernel is designed to process a sub-volume of the grid points. Before each kernel launch, the CPU transfers all atoms that are in the combined neighborhood of all the grid points in the sub-volume. These atoms are still stored in the constant memory. All threads examine all atoms in the joint neighborhood. The advantage of the kernel is its simplicity. It is essentially the same as the Direct Summation kernel with a relatively large, pre-selected neighborhood of atoms. Note that the `LargeBin` approach performs reasonably well for moderate volumes and scales well for large volumes.

The SmallBin algorithm allows the threads running within the same kernel to process different neighborhoods of atoms. This is the algorithm that uses global memory and shared memory for storing atoms. The algorithm achieves higher efficiency than the LargeBin algorithm because each thread needs to examine a smaller number of atoms. For moderate volumes, around 8000 Å^3, the LargeBin algorithm slightly outperforms SmallBin. The reason is that the SmallBin algorithm does incur more instruction overhead for loading atoms from global memory into shared memory. For a moderate volume, there is limited number of atoms in the entire system. The ability to examine a smaller number of atoms does not provide sufficient advantage to overcome the additional instruction overhead. However, the difference is so small at 8000 Å^3 that the SmallBin algorithm is still a clear win across all volume sizes. The SmallBin-Overlap algorithm overlaps the sequential overflow atom processing with the next kernel execution. It provides a slight but noticeable improvement in running time over SmallBin. The SmallBin-Overlap algorithm achieves a 17× speedup an efficiently implemented sequential CPU–SSE cutoff algorithm and maintains the same scalability for large volumes.

17.4 COMPUTATIONAL THINKING

Computational *thinking* is arguably the most important aspect of parallel application development [Wing 2006]. We define computational thinking as the thought process of formulating domain problems in terms of computation steps and algorithms. Like any other thought processes and problem-solving skill, computational thinking is an art. As we mentioned in Chapter 1, Introduction, we believe that computational thinking is best taught with an iterative approach where students bounce back and forth between practical experience and abstract concepts.

The electrostatic potential map kernels used in Chapter 15, Application case study—molecular visualization and analysis, and this chapter serve as good examples of computational thinking. In order to develop an efficient parallel application that solves the electrostatic potential map problem, one must come up with a good high-level decomposition of the problem. As we showed in Section 17.2, one must have a clear understanding of the desirable (e.g., gather in CUDA) and undesirable (e.g., scatter in CUDA) memory access behaviors to make a wise decision.

Given a problem decomposition, parallel programmers face a potentially overwhelming task of designing algorithms to overcome major challenges in parallelism, execution efficiency, and memory bandwidth consumption. There is a very large volume of literature on a wide range of algorithm techniques that can be hard to understand. It is beyond the scope of this book to have a comprehensive coverage of the available techniques. We did discuss a substantial set of techniques that have broad applicability. While these techniques are based on CUDA, they help the readers build up the foundation for computational thinking in general. We believe that humans understand best when we learn from the bottom up. That is, we first learn the concepts in the context of a particular programming model, which provide us with

solid footing before we generalize our knowledge to other programming models. An in-depth experience with the CUDA model also enables us to gain maturity, which will help us learn concepts that may not even be pertinent to the CUDA model.

There is a myriad of skills needed for a parallel programmer to be an effective computational thinker. We summarize these foundational skills as follows:

- Computer architecture: memory organization, caching and locality, memory bandwidth, Single Instruction, Multiple Thread (SIMT) vs Single Program, Multiple Data (SPMD) vs. Single Instruction, Multiple Data (SIMD) execution, and floating-point precision vs. accuracy. These concepts are critical in understanding the tradeoffs between algorithms.
- Programming models and compilers: parallel execution models, types of available memories, array data layout, and thread granularity transformation. These concepts are needed for thinking through the arrangements of data structures and loop structures to achieve better performance.
- Algorithm techniques: tiling, cutoff, scatter-gather, binning, and others. These techniques form the toolbox for designing superior parallel algorithms. Understanding of the scalability, efficiency, and memory bandwidth implications of these techniques is essential in computational thinking.
- Domain knowledge: numerical methods, precision, accuracy, and numerical stability. Understanding these ground rules allows a developer to be much more creative in applying algorithm techniques.

Our goal for this book is to provide a solid foundation for all the four areas. The reader should continue to broaden his/her knowledge in these areas after finishing this book. Most importantly, the best way of building up more computational thinking skills is to keep solving challenging problems with excellent computational solutions.

17.5 SINGLE PROGRAM, MULTIPLE DATA, SHARED MEMORY AND LOCALITY

At this point, it is worth saying a few words about some different parallel programming models, specifically Shared Memory vs. Message Passing. You may be familiar with these concepts from other studies, or you may encounter them later. We have focused on shared memory parallel programming, because this is what CUDA (and OpenMP, OpenCL) is based on. Also, most if not all future massively parallel microprocessors are expected to support shared memory at the chip level. The programming considerations of the message passing model are quite different; however, you will find similar concepts for almost every technique you learned in parallel programming.

In either case, you will need to be aware of space-time constraints. Data locality (or lack thereof) in time of access/use and data locality in access patterns can have profound effects on performance. Data sharing, whether intentional or not, can be a double-edged sword. Excessive data sharing can drastically reduce the performance advantages of parallel execution, so it is important not to overshare. Localized

sharing can improve memory bandwidth efficiency without creating conflicts and contention. Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory and data. Also important is efficient use of on-chip, shared storage and datapaths. Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization.

You can think of sharing as appearing in one of four modes: Many:Many, One:Many, Many:One, or One:One. This is true for both data sharing, and synchronization. You may think of synchronization as "control sharing." An example is barriers—barriers may cause some threads to wait until other threads catch up. This may be good in that execution is aligned, but it is also bad in that waiting is a lost opportunity for work. Atomic operations may reduce waiting, but perhaps at the cost of serialization. It is important to be aware of which work items are truly independent and not introduce false/unnecessary synchronizations into your program.

Program models and data organization drive parallel programming coding styles. One very important program model is SPMD. In this model, all PEs (processor elements) execute the same program in parallel, but program instance has its own unique data. Consequently, each PE uses a unique ID to access its portion of data, but each different PE can follow different paths through the same code. This is essentially the CUDA Grid model (also OpenCL, MPI). SIMD is a special case of SPMD where the threads move in lock-step—in CUDA execution, SIMD WARPs are used for efficiency.

SPMD programming also drives algorithm structures and coding styles. Due to the prevalence of massively parallel processors, this is currently the dominant coding style of scalable parallel computing. MPI code is mostly developed in SPMD style, so it is often used for parallel programming in multicore CPUs as well. Many OpenMP codes are also created in SPMD style, or utilize loop parallelism. This style is particularly suitable for algorithms based on task parallelism and geometric decomposition. A powerful advantage of this approach is that tasks and their interactions are visible in one piece of source code, and there is no need to correlate multiple sources and algorithm fragments to understand what is going on.

Many SPMD programs look similar, as almost all SPMD programs have the same typical program phases. These are:

1. Initialize—establish localized data structure and communication channels.
2. Uniquify—each thread acquires a unique identifier, typically ranging from 0 to *N*-1, where *N* is the number of threads. Both OpenMP and CUDA have built-in support for this.
3. Distribute data—decompose global data into chunks and localize them, or sharing/replicating major data structures using thread IDs to associate subsets of the data to threads.
4. Compute—run the core computation! Thread IDs are used to differentiate the behavior of individual threads. Use thread ID in loop index calculations to split loop iterations among threads—beware of the potential for memory/data

divergence. Use thread ID or conditions based on thread ID to branch to their specific actions—beware of the potential for instruction/execution divergence.
5. Finalize—reconcile global data structure, and prepare for the next major iteration or group of program phases.

You will see this pattern a myriad of times in your journeys through parallel programing, and it will become second nature for you to organize the solution of your problem in this fashion.

17.6 STRATEGIES FOR COMPUTATIONAL THINKING

A good goal for effective use of computing is making science better, not just faster. This requires re-examining prior assumptions and really thinking about how to apply the big hammer of massively parallel processing. Put another way, there will probably be no Nobel Prizes or Turing Awards awarded for "just recompile" or using more threads with the same computational approach! Truly important scientific discoveries will more likely come from fresh computational thinking. Consider this an exhortation to use this bonanza of computing power to solve new problems in new ways.

As a strategy for attacking computation-hungry applications, we can consider a three-step approach:

1. Tune core software for hardware architecture.
2. Innovate at the algorithm level.
3. Restructure the mathematical formulation.

This breakdown leads to three options, in increasing order of difficulty, complexity, and not surprisingly, potential for payoff. Let us call these good, better, and best!

The "good" approach is simply to "accelerate" legacy program codes. The most basic approach is simply to recompile and run on a new platform or architecture, without adding any domain insight or expertise in parallelism. This approach can be improved by using optimized libraries, tools, or directives, such as CuBLAS, CuFFT, Thrust, Matlab, OpenACC, etc. This is very good and rewarding work for domain scientists—minimal Computer Science knowledge or programming skills are required. We can categorize this approach as only choosing to attack part of step 3 above.

The "better" approach involves rewriting existing codes using new parallelism skills to take advantage of new architectures, or creating new codes from scratch. We can benefit from new algorithmic techniques to increase execution efficiency. This is an opportunity for clever algorithmic thinking, and is good work for nondomain computer scientists, as minimal domain knowledge is required. We can categorize this approach as choosing only to attack part of step 2 above.

The "best" approach involves more deep and careful thought and a holistic approach involving all three steps above. We wish to not only map a known algorithm and computation to a parallel program and architecture, but also rethink the numerical methods and algorithms used in the solution. In this approach, there is the

potential for the biggest performance advantage and fundamental new discoveries and capabilities. It is, however, also much more difficult. This approach is interdisciplinary and requires both Computer Science *and* domain insight, but the payoff is worth the effort. It is truly an exciting time to be a computational scientist!

The order of operations should be: Think, Understand… and then and only then, Program.

First, think deeply about the problem you are trying to solve, and truly understand the overall structure of the problem. Apply mathematical techniques to find a solution to your problem, and then map the mathematical technique to an algorithmic approach for computational solution.

Plan the structure of computation in detail. Be aware of in/dependence, interactions, and bottlenecks, and plan the work phases accordingly. Plan the organization of data, both for input and output, as well as during the computation phases. Be explicitly aware of locality, opportunities for data sharing or privatization, and minimize global data access and movement. Finally, write some code! This is the easy part ☺—all of the hard work has been done in the planning stage.

Of course this description is an oversimplification and many other techniques and considerations are left for future studies. We could explore more complex data structures, more scalable algorithms and building blocks, or more scalable or hierarchical mathematical models. We can also consider thread-aware approaches to capitalize on more available parallelism. There will also be great opportunities in locality-aware approaches, since computing is becoming bigger, and everything is becoming further away both in space and time. All of these are beyond the scope of this book, but we encourage you to pursue them.

17.7 A HYPOTHETICAL EXAMPLE: SODIUM MAP OF THE BRAIN

Let us provide a hypothetical example of rethinking a well-known and well-solved problem—MRI. We have discussed this problem in detail in previous chapters, and have described many aspects of how to use parallel computing to efficiently convert the frequency samples from the MRI device into spatial information. We will start with a worthy goal: creating 3D images of sodium in the brain, using MRI. Normally, MRI images the density of water in human tissue, as measured by the resonance of hydrogen atoms. The density of water provides imaging of anatomy—what kind of tissue exists in the 3D volume. Why is sodium imaging desirable? First, sodium is one of the most regulated substances in human tissues—any significant shift in sodium concentration signals cell death. Real-time measurement of sodium density would enable study of brain-cell viability before anatomic changes occur as a result of stroke and cancer treatment. This would provide a drastic improvement in timeliness of treatment decisions. We would be able to determine if treatment is effective within critical minutes for stroke and days for oncology, saving and improving the quality of many lives. Fig. 17.5 shows what this might look like.

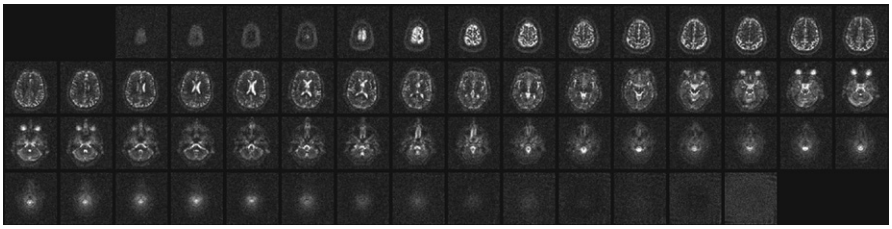

FIGURE 17.5

Sodium images of the brain. Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago.

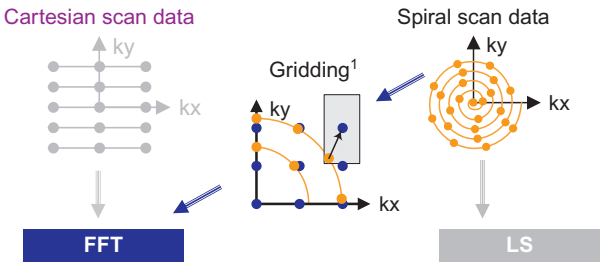

FIGURE 17.6

Classical gridded MRI reconstruction from spiral scan data.

So, why is sodium imaging difficult? Unfortunately, sodium is much less abundant than water in human tissues, about 1/2000 the concentration. In addition, the magnetic resonance echo of sodium is much less strong than the echo from hydrogen/water. Thus, a very, very much larger number of samples would be required for good signal-to-noise ratio in the MRI. Another possible approach would be to choose mathematical techniques and algorithms for much higher quality signal reconstruction. This approach has been considered impractical due to the massive computation required. However, it is time to re-examine those assumptions. When MRI was first developed, computers were much slower, and MRI reconstruction required a long time on a roomful of computing equipment. Conventional MRI reconstruction is now real-time using desk-side hardware.

Non-Cartesian trajectories, such as the spiral trajectory shown here in Fig. 17.6, are becoming increasingly popular and common. These non-Cartesian scans are faster and less susceptible to artifacts than Cartesian scans. The spiral scan pattern is also a more efficient use of time on the MRI machine.

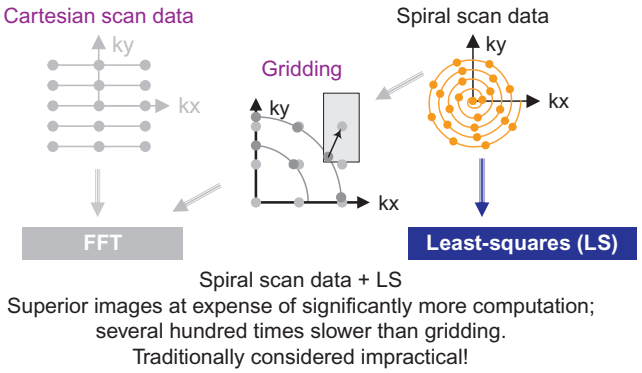

FIGURE 17.7

Least squares reconstruction of spiral scan data.

However, the FFT cannot be applied directly to the non-Cartesian scan data. One popular approach is to "grid" the data. That is, the non-Cartesian data (shown in orange (light gray in the print versions)) is interpolated onto a uniform grid (shown in blue (dark gray in the print versions)) using some sort of windowing function. The FFT can then be applied to the interpolated data. This technique introduces inaccuracies and satisfies no statistical optimality criterion, but is very fast and does produce better images than a Cartesian scan.

This is similar to the old joke about a man who has lost his keys in a dark corner of a parking lot. He is discovered looking for his keys under the streetlight. When asked why he is looking for his keys there instead of where he lost them, the reply is "because the light is better here!" While it may have been true that the gridding and FFT solution was a good choice for MRI processing at first, this is likely no longer the case.

Least-squares (LS) iterative reconstruction is a superior technique that operates directly on the nonuniform data using the LS optimality criterion. The combination of a non-Cartesian scan and the LS reconstruction produces images far superior to those obtained via Cartesian scans or gridding. Unfortunately, these superior images come at the expense of increasing the amount of computation by several orders of magnitude. For the LS reconstruction to be practical in clinical settings, it must be accelerated by a similar number of orders of magnitude.

Again, this is what we mean when we say that the GPU allows us to change the boundaries of science. The LS reconstruction algorithm isn't viable on the CPU. It's the GPU that makes the LS reconstruction practical, so that we don't have to use lossy and approximate techniques like gridding, just so that we can warp the problem into a convenient computational framework such as FFT (Fig 17.7).

Instead of simply applying brute force to the much harder problem of sodium imaging using MRI, we have achieved much more by re-examining the foundations of MRI and the related computations.

17.8 SUMMARY

In summary, we have discussed the main dimensions of algorithm selection and computational thinking. The key lesson is that given a problem decomposition decision, the programer will typically have to select from a variety of algorithms. Some of these algorithms achieve different tradeoffs while maintaining the same numerical accuracy. Others involve sacrificing some level of accuracy to achieve much more scalable running times. The cutoff strategy is perhaps the most popular of such strategies. Even though we introduced cutoff in the context of electrostatic potential map calculation, it is used in many domains including ray tracing in graphics and collision detection in games. Computational thinking skills allow an algorithm designer to work around the roadblocks and reach a good solution.

17.9 EXERCISES

1. Write a host function to perform binning of atoms. Determine the representation of the bins as arrays. Think about coalescing requirements. Make sure that every thread can easily find the bins it needs to process.

2. Write the part of the cut-off kernel function that determines if an atom is in the neighborhood of a grid point based on the coordinates of the atoms and the grid points.

3. Think about refactoring to change the organization of work. Take a scatter-gather kernel, and rewrite as scatter-scatter, or gather-gather, to improve locality. Which is better for CUDA execution?

REFERENCES

Message Passing Interface Forum. (2009). *MPI—a message passing interface standard version 2.2*. <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf> Accessed 04.09.2009.

Rodrigues, C. I., Stone, J., Hardy, D., & Hwu, W. W. (2008). GPU acceleration of cutoff-based potential summation. In: *ACM computing frontier conference 2008*, Italy, May.

Wing, J. (March 2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.