1

Why Graphics Processing Units

Perri Needham¹, Andreas W. Götz² and Ross C. Walker¹,²

¹San Diego Supercomputer Center, UCSD, La Jolla, CA, USA
²Department of Chemistry and Biochemistry, UCSD, La Jolla, CA, USA

1.1 A Historical Perspective of Parallel Computing

The first general-purpose electronic computers capable of storing instructions came into existence in 1950. That is not to say, however, that the use of computers to solve electronic structure problems had not already been considered, or realized. From as early as 1930 scientists used a less advanced form of computation to solve their quantum mechanical problems, albeit a group of assistants simultaneously working on mechanical calculators but an early parallel computing machine nonetheless [1]. It was clear from the beginning that solutions to electronic structure problems could not be carried forward to many-electron systems without the use of some computational device to lessen the mathematical burden. Today’s computational scientists rely heavily on the use of parallel electronic computers.

Parallel electronic computers can be broadly classified as having either multiple processing elements in the same machine (shared memory) or multiple machines coupled together to form a cluster/grid of processing elements (distributed memory). These arrangements make it possible to perform calculations concurrently across multiple processing elements, enabling large problems to be broken down into smaller parts that can be solved simultaneously (in parallel).

The first electronic computers were primarily designed for and funded by military projects to assist in World War II and the start of the Cold War [2]. The first working programmable digital computer, Konrad Zuse’s Z3 [3], was an electromechanical device that became operational in 1941 and was used by the German aeronautical research organization. Colossus, developed by the British for cryptanalysis during World War II, was the world’s first programmable electronic digital computer and was responsible for the decryption of valuable German military intelligence from 1944 onwards. Colossus was a purpose-built machine to determine the encryption settings for the German Lorenz cipher and
read encrypted messages and instructions from paper tape. It was not until 1955, however, that the first general-purpose machine to execute floating-point arithmetic operations became commercially available, the IBM 704 (see Figure 1.1).

A common measure of compute performance is floating point operations per second (FLOPS). The IBM 704 was capable of a mere 12,000 floating-point additions per second and required 1500–2000 ft\(^2\) of floor space. Compare this to modern smartphones, which are capable of around 1.5 GIGA FLOPS [5] thanks to the invention in 1958 and a subsequent six decades of refinement of the integrated circuit. To put this in perspective, if the floor footprint of an IBM 704 was instead covered with modern-day smartphones laid side by side, the computational capacity of the floor space would grow from 12,000 to around 20,000,000,000,000 FLOPS. This is the equivalent of every person on the planet carrying out roughly 2800 floating point additions per second. Statistics like these make it exceptionally clear just how far computer technology has advanced, and, while mobile internet and games might seem like the apex of the technology’s capabilities, it has also opened doorways to computationally explore scientific questions in ways previously believed impossible.

Computers today find their use in many different areas of science and industry, from weather forecasting and film making to genetic research, drug discovery, and nuclear weapon design. Without computers many scientific exploits would not be possible.

While the performance of individual computers continued to advance the thirst for computational power for scientific simulation was such that by the late 1950s discussions had turned to utilizing multiple processors, working in harmony, to address more complex scientific problems. The 1960s saw the birth of parallel computing with the invention of multiprocessor systems. The first recorded example of a commercially available multiprocessor (parallel) computer was Burroughs Corporation’s D825, released in 1962, which had four processors that accessed up to 16 memory modules via a cross switch (see Figure 1.2).
This was followed in the 1970s by the concept of single-instruction multiple-data (SIMD) processor architectures, forming the basis of vector parallel computing. SIMD is an important concept in graphics processing unit (GPU) computing and is discussed in the next chapter.

Parallel computing opened the door to tackling complex scientific problems including modeling electrons in molecular systems through quantum mechanical means (the subject of this book). To give an example, optimizing the geometry of any but the smallest molecular systems using sophisticated electronic structure methods can take days (if not weeks) on a single processor element (compute core). Parallelizing the calculation over multiple compute cores can significantly cut down the required computing time and thus enables a researcher to study complex molecular systems in more practical time frames, achieving insights otherwise thought inaccessible. The use of parallel electronic computers in quantum chemistry was pioneered in the early 1980s by the Italian chemist Enrico Clementi and co-workers [7]. The parallel computer consisted of 10 compute nodes, loosely coupled into an array, which was used to calculate the Hartree–Fock (HF) self-consistent field (SCF) energy of a small fragment of DNA represented by 315 basis functions. At the time this was a considerable achievement. However, this was just the start, and by the late 1980s all sorts of parallel programs had been developed for quantum chemistry methods. These included HF methods to calculate the energy and nuclear gradients of a molecular system [8–11], the transformation of two-electron integrals [8, 9, 12], the second-order Møller–Plesset perturbation theory [9, 13], and the configuration interaction method [8]. The development of parallel computing in quantum chemistry was dictated by developments in available technologies. In particular, the advent of application programming interfaces (APIs) such as the message-passing interface (MPI) library [14] made parallel computing much more accessible to quantum chemists, along with developments in hardware technology driving down the cost of parallel computing machines [10].

While finding widespread use in scientific computing until recently, parallel computing was reserved for those with access to high-performance computing (HPC) resources. However, for reasons discussed in the following, all modern computer architectures exploit parallel technology, and effective parallel programming is vital to be able to utilize the computational power of modern devices. Parallel processing is now standard across all devices fitted with modern-day processor architectures. In his 1965 paper [15], Gordon E. Moore first observed that the number of transistors (in principle, directly related to performance) on integrated circuits was doubling every 2 years (see Figure 1.3).
Figure 1.3 Microprocessor transistor counts 1971–2011. Until recently, the number of transistors on integrated circuits has been following Moore’s Law [16], doubling approximately every 2 years.

Since this observation was announced, the semiconductor industry has preserved this trend by ensuring that chip performance doubles every 18 months through improved transistor efficiency and/or quantity. In order to meet these performance goals the semiconductor industry has now improved chip design close to the limits of what is physically possible. The laws of physics dictate the minimum size of a transistor, the rate of heat dissipation, and the speed of light.

“The size of transistors is approaching the size of atoms, which is a fundamental barrier” [17].

At the same time, the clock frequencies cannot be easily increased since both clock frequency and transistor density increase the power density, as illustrated by Figure 1.4. Processors are already operating at a power density that exceeds that of a hot plate and are approaching that of the core of a nuclear reactor.

In order to continue scaling with Moore’s Law, but keep power densities manageable, chip manufacturers have taken to increasing the number of cores per processor as opposed to transistors per core. Most processors produced today comprise multiple cores and so are parallel processing machines by definition. In terms of processor performance, this is a tremendous boon to science and industry; however, the increasing number of cores brings with them increased complexity to the programmer in order to fully utilize the available compute power. It is becoming more and more difficult for applications to achieve good scaling with increasing core counts, and hence they fail to exploit this technology. Part of this problem stems from the fact that CPU manufacturers have essentially
taken serial (scalar) architectures and their associated instructions sets and attempted to extend them to a multicore regime. While this is well tolerated for single-digit core counts, once the number of cores hits double digits the complexity of having to keep track of how the data are distributed across multiple independent cores, which requires communication between local cache memories, quickly leads to performance bottlenecks. This complexity has, to date, limited the number of cores and, ultimately, the performance available in traditional CPUs. In the next section we discuss an alternative to multicore programming that utilizes the massively parallel (SIMD) architecture of GPUs.

1.2 The Rise of the GPU

The concept of using a specialized coprocessor to supplement the function of the central processing unit originated in the 1970s with the development of math coprocessors. These would handle floating-point arithmetic, freeing the CPU to perform other tasks. Math coprocessors were common throughout the 1980s and early 1990s, but they were ultimately integrated into the CPU itself. The theme of using a coprocessor to accelerate specific functions continued however, with video accelerators being the most common example. These GPUs would free the CPU from much of the complex geometric math required to render three-dimensional (3D) images, and their development was catalyzed by the computer gaming industry’s desire for increasingly realistic graphics.

Beyond the math coprocessor of the 1980s, coprocessors have been routinely used in scientific computation as a means of accelerating mathematically intensive regions of code. More recent examples include ClearSpeed’s accelerator boards, which combined hundreds of floating point units on a single PCI card, and field-programmable gate arrays (FPGAs), which can be reconfigured to accelerate a specific computational problem. These accelerator technologies, while exploited by some, have not found widespread use in the general scientific computing arena for a number of reasons, the main one being their cost. However, one coprocessor technology that has succeeded in being adopted by the wider scientific computing community is the GPU. The reasons for this are many, but probably the overriding reason for their widespread adoption is that they are ubiquitous and cheap. GPUs have been an integral part of personal computers for decades. Ever since the introduction of the Voodoo graphics chip by 3DFX in 1996, the entertainment industry has been the major driving force for the development of GPUs in order to meet the demands for increasingly realistic computer games. As a result of the strong demand from the consumer electronics industry, there has been significant industrial investment in the stable, long-term development of GPU
technology. Nowadays, GPUs have become cheap and ubiquitous and, because their processing power has substantially increased over the years, they have the potential, when utilized efficiently, to significantly outperform CPUs (see Figure 1.5).

GPUs are thus very attractive hardware targets for the acceleration of many scientific applications including the subject of this book, namely electronic structure theory. GPUs with significant computing power can be considered standard equipment in scientific workstations, which means that they either are already available in research labs or can be purchased easily with new equipment. This makes them readily available to researchers for computational experimentation. In addition, many compute clusters have been equipped with large numbers of high-end GPUs for the sole purpose of accelerating scientific applications. At the time of writing, the most powerful supercomputer that is openly accessible to the scientific community draws the majority of its peak computational performance from GPUs.

The nature of GPU hardware, however, originally made their use in general-purpose computing challenging because it required extensive three-dimensional (3D) graphics programming experience. However, as discussed in the following, the development of APIs for general-purpose scientific computing has reduced this complexity such that an extensive range of scientific problems is making use of GPU acceleration in an economically efficient manner. GPU-accelerated computing is now a broad but still advancing area of research and development that is capable of providing scalable supercomputing power at a fraction of the cost and, potentially, a fraction of the software development effort (depending on the application).

GPUs were originally designed for computer graphics, which require repeated evaluation of pixel values and little else. For example, branching is not required in rendering 3D images and therefore in place of sophisticated cache hierarchies, branch prediction hardware, pipeline controllers, and other complex features, GPUs instead have hundreds to thousands of simplistic cores. GPUs are fundamentally parallel, with each core computing as an individual thread, making them ideal for data decompositions with few dependencies between threads. For fine-grained parallel workloads, GPUs can thus provide a substantial amount of computing power. As can be seen from Figure 1.5, they provide, largely by the difference in memory bandwidth, a means to improve the performance of a suitable code far beyond the limits of a CPU.

The scientific community has leveraged this processing power in the form of general-purpose (GP) computation on GPUs, sometimes referred to as GPGPU. GPGPUs were specifically designed to target HPC applications incorporating additional features into GPUs, such as caching technology, and support for double-precision arithmetic. The success of early GPGPU efforts was such that these features are now considered standard in almost all GPUs, and indeed many modern computer games...
actually make use of the GPGPU functionality to compute game physics. The traditional approach to computing on a GPU is as an accelerator, or coprocessor, working in conjunction with the CPU. The GPU executes specific functions within the code (known as kernels), which are deemed suitable for parallelization on a GPU. A GPU operates a SIMD instruction set, whereby an instruction is issued to many cores in a single instance. The lightweight nature of a GPU core makes it ideally suited to parallelize compute-intensive code with few data dependencies because, once the instruction is issued, cores cannot diverge along different code paths. If branching occurs on cores issued with the same instruction, some cores will stall until the execution paths converge. This artifact of GPU hardware is what sets them apart from CPU architectures. It enables calculation of multiple data elements in one instance, provided the code design allows it. However, that is not to say that CPUs do not make use of SIMD, but rather GPUs implement it on a much larger scale. For example, the latest GPU in Nvidia’s Tesla range, Tesla K40, boasts 2880 GPU cores and has the capacity to launch multiple threads per core, which can be quickly switched in and out with minimal overhead, whereas the latest CPU in the Intel® Core™ product family, Intel® Core™ i7-4930K processor, comprises six cores capable of spawning two threads per core. The CPU makes use of SIMD to carry out multiple operations per thread; in this case, each thread is a 256-bit SIMD instruction in the form of AVX. In principle, the CPU can therefore execute 96 (256-bit/32-bit × 12 threads) single-precision floating-point operations per clock cycle, although in reality it is effectively half of this since the two threads per core are time-sliced. A GPU therefore can potentially process hundreds, if not thousands, of data elements in a single instance as opposed to a mere 48 on a CPU. However, it cannot be emphasized enough how important the GPU code design is in order to take advantage of this potential performance, often requiring a rethinking of the underlying mathematical algorithms. The purpose of this book is to highlight such rethinking within the framework of electronic structure theory.

1.3 Parallel Computing on Central Processing Units

Before entering into the world of GPU programming it is important to cement some key concepts that lie at the heart of parallel programming. Parallel programming requires substantially more consideration than serial programming. When parallelizing computer code it cannot be presumed that all parallel processing elements have access to all the data in memory or indeed that memory updates by one element will automatically propagate to the others. The locality of data to a processing element is dependent on the setup of the computer system being used and the parallel programming memory model, which is the first concept to be discussed in this section. Following this are the language options available to the programmer that are best suited to a specific programming model and also a discussion of the different types of parallelism to consider when breaking down a problem into smaller parallel problems. It is also beneficial to be aware of the factors that may affect the performance of a parallel program and how to measure the performance and efficiency of a parallel program, which will be the final topics discussed. For a more detailed exposition than presented in the following, we refer the reader to the excellent text book on HPC by Hager and Wellein, which also covers aspects of computer architecture and serial code optimizations in addition to parallelization [20].

1.3.1 Parallel Programming Memory Models

In parallel computing, there are two main memory models: the shared memory model and the distributed memory model. Each model brings with it different implications at both the hardware and software level. In hardware, the shared memory model is exactly that – multiple processors share a single address space in memory. The distributed memory model refers to segregated memory, where
each processor has its own private address space, be it a physically private memory space or a portion of a larger memory.

In shared memory all processors have access to all addresses in memory making communication between processors potentially very fast. However, problems arise when more than one processor requires access to the same memory address, resulting in potential race conditions, a situation – discussed in more detail later in this chapter – where the result of a computation is a function of which of the threads computes the answer first, and communication overhead in keeping the data stored in different memory caches coherent. In distributed memory all processors have access only to their own memory address space and so communication between processors becomes the bottleneck. If a process requires data that is not stored in its private memory space the process first has to find which processor's private memory the data is stored in before it can be transferred via an interconnect. Consequently, shared memory systems are deemed more suited to fewer processor counts, as performance falls off with increasing numbers of processors as a result of explosion of cache coherency overhead. Neither memory model is ideal. In practice, most modern-day parallel computers use a mixture of the two types of memory model. Many scientific computing clusters utilize a hybrid memory model, with shared memory nodes coupled together with an interconnect providing a distributed memory model.

1.3.2 Parallel Programming Languages

An Application Programming Interface (API) fundamentally is a means of communicating with a computer. Communication with a computer is layered and can be easily imagined as a metaphorical onion. At the very core of the onion is binary. Zeros and 1's represent the switch states of transistors on a piece of silicon. Binary allows these switch states to represent real numbers. The next layer of the onion is machine code. A small number of very basic operations instruct the CPU what to do with the real numbers. The layer after that is assembly, which is similar to machine code except with slightly more human-friendly operations enabling manipulation of the real numbers with more ease. And so it goes on. The larger the onion, that is, the more the layers of communication, the easier it is for a programmer to express his/her computational problem because with each added layer the language becomes closer and closer to human communication. These language layers are known as APIs and can be deemed high-level (outer layers of a large onion) or lower level (layers close to the core). The two most widely used APIs for expressing parallel CPU code are MPI (Message Passing Interface) [14] and OpenMP (Open Multi-Processing) [21]. MPI is a library called upon from existing CPU code. The library is language-independent, which means it can be called from code written in any compatible programming language, that is, C, C++, or Fortran. MPI enables point-to-point and collective communication between processors in a system and is primarily designed for use with a distributed memory system, but it can also be used within a shared memory setting. OpenMP is primarily a set of compiler directives that can be called from within C/C++ and Fortran code. OpenMP is designed primarily for shared memory programming, although OpenMP 4.0 supports a “target directive” for running across devices that do not share memory spaces and exhibits a multithreaded paradigm to distribute work across processors. OpenMP is considered a higher level parallel programming API than MPI, as the runtime environment and the compiler do most of the work. The programmer can do as little as specify which sections of code require parallelizing through the use of a directive and let the API do the rest of the work. MPI requires more thought, as the programmer is responsible for keeping track of data in memory and explicitly passing data between processors. However, one benefit from this added degree of difficulty when using MPI is that it forces the programmer to think about data locality and so can lead to more efficient, higher performing code.

Other APIs used in parallel programming are high-performance Fortran (HPF), POSIX threads, and Unified Parallel C (UPC), to name a few; however, they are beyond the scope of this textbook.
1.3.3 Types of Parallelism

One key consideration when parallelizing code is how to distribute the work across processors. The two main types of decomposition are task parallelism and data parallelism. Task parallelism involves deconstructing code into separate, independent tasks/calculations able to be computed on different processors concurrently. Each processor is responsible for executing its own set of instructions using the same or different data as other processes. Figure 1.6 shows a simple example of a four-task program. Task B is dependent on Task A, therefore they are executed sequentially. Task C is independent of both A and B, therefore they can be executed concurrently; Task D is dependent on the completion of Tasks B and C, therefore it is executed on their completion.

Data parallelism involves decomposing a dataset across processors, and each processor executing the same instructions but on different data. Data parallelism is synonymous with SIMD architecture. There are many different strategies for data parallelism, for example, static where the data are decomposed into chunks and distributed across processors at compile time and each processor carries out all operations on the same chunk of memory initially allocated, and dynamic parallelism where a load balancer hands out work on a per-request basis. A more detailed description of data decomposition strategies is beyond the scope of this chapter and so will not be discussed further.

Similar to the parallel memory models, a combination of both types of parallelism can often be the most fruitful where performance is concerned. It can be beneficial to employ both types of parallelism, for example, decomposing code into tasks across multiple compute nodes and further decomposing the data for each task across multiple processors within a compute node. By doing this, multiple levels of parallelism can be exploited, making better use of the available computer architecture.

The ease and extent by which an algorithm can be broken up into separate tasks/calculations is known as the granularity. Fine-grained parallelism describes algorithms that can be subdivided into many small calculations. These types of algorithm are said to be more easily parallelized but can require more synchronization/communication between parallel processes. On the contrary, coarse-grained parallelism is the subdivision of an algorithm into fewer larger tasks/calculations. These types of algorithm do not benefit from as much parallelism, that is, scale to small processor

![Directed acyclic graph demonstrating the decomposition of four tasks across two processors](image)

**Figure 1.6** Directed acyclic graph demonstrating the decomposition of four tasks across two processors
counts, but can benefit from fewer overheads as a result of reduced communication. Algorithms that can be subdivided into many small tasks (similar to fine-grained parallelism) yet do not carry large communication overheads (similar to coarse-grained parallelism) are said to be *embarrassingly* parallel (although the term naturally parallel might be a better description) and are the easiest applications to parallelize.

### 1.3.4 Parallel Performance Considerations

#### 1.3.4.1 Speedup

Before parallelizing code, it is important to realistically predict the potential for performance improvement. A common measure is speedup ($S$), which is given by

$$S(n) = \frac{T(1)}{T(n)}, \quad (1.1)$$

where $T(1)$ is the execution time of the original algorithm and $T(n)$ is the execution time of the new algorithm on $n$ processors/cores/threads.

#### 1.3.4.2 Amdahl’s Law and Gustafson’s Law

Amdahl’s and Gustafson’s laws provide a guideline for speedup after parallelizing code. Effectively a simplification of a concept, familiar to all chemists, known as a rate-determining step, they show that the serial portion of a parallel program is always the limiting factor in speedup. There are slight subtleties between the two laws; Amdahl presumes a fixed problem size, whereas Gustafson presumes increasing problem size with increasing numbers of processors. Amdahl’s Law, named after the computer architect Amdahl [22], was introduced in 1967 and is defined by

$$T(n) = T(1) \left( \beta_A + \frac{1}{n} (1 - \beta_A) \right), \quad (1.2)$$

$$S(n) = \frac{T(1)}{T(1) \left( \beta_A + \frac{1}{n} (1 - \beta_A) \right)} = \frac{1}{\beta_A + \frac{1}{n} (1 - \beta_A)}, \quad (1.3)$$

where $\beta_A$ is the fraction of the algorithm that is non-parallelizable, that is, the fraction of the total run time $T(1)$ that the serial program spends in the non-parallelizable part. As can be seen from Amdahl’s equation, as the number $n$ of processors increases, the speedup becomes limited by the serial code fraction $\beta_A$. However, Amdahl’s Law is rather simplistic, as it makes assumptions such as the number of processes used throughout the parallel portions of the code is a constant; the parallel portion achieves linear speedup (i.e., the speedup is equal to the number of processes); and the parallel portion scales perfectly, to name a few. However, the most important assumption made was that the serial and parallel workloads remain constant when increasing the number of processors. This is true for many scientific problems, particularly those involving integrals over time.

However, some applications are problem-size-bound rather than time-bound, such as weather modeling and graphics rendering, and given more processors, the size of the problem is usually increased to fill the void. Gustafson attempted to remedy this shortcoming in 1988 by defining parallel speedup by a new equation [23], sometimes referred to as “scaled speedup”:

$$T(1) = (n - \beta_G(n)(n - 1))T(n), \quad (1.4)$$

$$S(n) = n - \beta_G(n)(n - 1), \quad (1.5)$$

where $\beta_G$ is again the non-parallelizable fraction, but now defined as the fraction of the total time $T(n)$ that the parallel program spends in serial sections if run on $n$ processors. The two different definitions
of speedup in terms of processor count \( n \) and serial code fraction \( \beta \) can lead to confusion. However, one has to keep in mind that for a given problem size, \( \beta_G \) depends on the number of processors, as indicated in Eqs. (1.4) and (1.5). Equations (1.3) and (1.5) are actually equivalent with the following relation between the different definitions of the serial code fraction, as can be derived from Eqs. (1.2) and (1.4):

\[
\beta_A = \frac{1}{1 + \frac{(1-\beta_G)n}{\beta_G}}
\]  

(1.6)

In the application of Eq. (1.5), one typically assumes that \( \beta_G \) remains constant with increasing processor number. In other words, Gustafson’s Law states that exploiting the parallel power of a large multiprocessor system requires a large parallel problem. Gustafson’s Law thus allows for expansion of the parallel portion of an algorithm but still makes all the same assumptions as Amdahl’s Law. Either way, the serial code will ultimately become the bottleneck as the core count increases. It is thus essential for effective parallel programming that all serial aspects of the compute portion of the code are removed. If a section of the algorithm is serial, even though representing only a tiny fraction of the total compute time, it is necessary to rethink the approach to remove this serial bottleneck if one wishes to scale across the thousands of cores in GPUs. For example, if 99% of code is parallel and it is run on 100 cores with ideal scaling, the remaining 1% that is serial will take over 50% of the total compute time. If one then runs the same calculation on 1000 cores, again assuming ideal scaling, the serial portion now consumes over 90% of the total compute time.

1.3.4.3 Race Conditions

Coding in parallel requires the programmer to consider the possible order of execution for multiple processes at any one time. It is important to be aware of some possible scenarios that may happen during execution with multiple processes.

Scenario 1: Process A relies on the result of a calculation carried out by process B. This is known as a dependency, more specifically a data dependency. Process A cannot continue to execute until process B has calculated the result. If it does, the answer will be incorrect since it does not have valid data from B. A solution would be to introduce synchronization between the processes prior to the operation that carries the dependency, or, alternatively, design a new parallel algorithm that eliminates the dependency.

Scenario 2: Process A and process B are both adding a number to the same variable \( X \) in memory. Process A and process B both take the current value of \( X \) as 5 and begin to add their result to the variable. Process A does \( X = 5 + 2 \), whereas process B does \( X = 5 + 1 \). Depending on which process is quicker at putting the new value of \( X \) back in memory, the result will be either 7 or 6. However, the desired value of \( X \) in memory is actually 8. This is known as a race condition. One solution would be to make use of a lock, which ensures that only one process can access any one variable at any time, thereby preventing a race condition by creating mutual exclusion of variable \( X \). The use of a lock is often discouraged however, since it introduces serialization into the code. Various tricks can be used to circumvent this, such as the lock-free, warp-synchronized, approach used by the GPU version of the AMBER MD software discussed later (see Section 1.5.1).

1.3.4.4 Communication Metrics

While it might be possible to perfectly parallelize an algorithm, this does not necessarily mean that the algorithm will be executed at the theoretical limit of the hardware’s performance. The reason for this is that all algorithms ultimately require some form of communication between each process (unless embarrassingly parallel). Some might require the reduction of a set of calculations, for example, a summation executed in parallel, while others might require comparison between values or the sharing of variables describing the system’s state, for example, the coordinates of a set of atoms, between
there are two major aspects to communication that determine how costly, in terms of time, the sharing of data between processes is. Bandwidth is a measure of the amount of data that can be communicated in a given timeframe, a.k.a. bit rate, that is, gigabytes per second. When measuring performance, the actual bandwidth (known as the throughput) can be compared with the theoretical bandwidth capability of the hardware to gauge the efficiency of communicating data. Latency is another metric to measure the performance of data movements; however, it is more concerned with the time it takes to communicate data rather than the amount of data it can communicate. Throughput and latency essentially measure the same thing: data communication efficiency. In simple terms, latency is the time taken to establish a communication channel, and bandwidth is the amount of data per unit time that can be sent along the channel. Algorithms that rely on large numbers of small packets of data will typically be latency-bound, while algorithms that send large packets of data infrequently will typically be bandwidth-bound. Since the design of parallel hardware typically involves a balancing act between bandwidth and latency, some algorithms will naturally perform better on certain types of hardware than others.

1.4 Parallel Computing on Graphics Processing Units

This section follows much the same order of concepts/theory as the previous section; however, it does so in the context of GPUs. First to be discussed is the memory model of GPUs. Following this will be a discussion of the available APIs and their compatible hardware, code suitability, and, finally, a discussion of scalability, performance and cost effectiveness.

1.4.1 GPU Memory Model

The memory model of a system equipped with GPU coprocessors does not easily fall into either of the distinct categories shared or distributed, as both the CPU(s) and the GPU(s) have their own memory spaces. The data required by a GPU in order to accelerate offloaded code from the CPU has to be accessible to the GPU. This involves the transferring of data from the CPU to the GPU, similar to the transfer of data from process to process in a distributed memory system. Once the data are on the GPU, the memory model is quite different. The main memory on a GPU, referred to as global memory, is accessible and visible to all cores/threads. This reflects the shared memory model. However, as a GPU has multiple memory locations (each with varying capacity, bandwidth, and latency depending on the proximity to the compute cores), the shared memory model is not consistent as one steps through the memory hierarchy. The memory hierarchy of a GPU will be discussed in more detail in Chapter 2. Taking advantage of the memory hierarchy is one of the key goals when writing GPU code and is crucial in achieving good performance. A GPU offers a hybrid memory platform giving the programmer control of data locality.

1.4.2 GPU APIs

At present there are two main APIs available to program GPUs: OpenCL and CUDA. A higher level GPU programming option, OpenACC, is also available, which is a collection of directives and programs, logically equivalent to OpenMP, that allows the compiler to analyze sections of code and automatically map loops onto parallel GPU cores. In this textbook, most examples and concepts will be discussed in terms of the CUDA programming paradigm, with some chapters making use of OpenCL. CUDA is freeware-supported, at the time of writing, only by Nvidia GPU hardware, whereas OpenCL is a cross-platform standard available and compatible on many different coprocessor architectures, not just GPUs. CUDA is a series of extensions to the C, C++, and Fortran
Why Graphics Processing Units

programming languages. This has the benefit of reducing the programming effort, as it is not formally required that algorithms be rewritten in order to run on a GPU, only modified to include the CUDA syntax. However, it is worth noting that the majority of algorithms need to be modified in order to execute efficiently on the many-core architecture of a GPU. Therefore, redesigning algorithms may be unavoidable for many applications to get the most performance out of a GPU as highlighted in the following. OpenCL offers a lower level set of extensions to the C programming language, providing the programmer with more control of the hardware. The programming model is very similar to CUDA’s, once the terminology is stripped away. For instance, in the CUDA programming model, a thread is the sequence of instructions to be executed by a CUDA core, whereas in the OpenCL programming model a thread is termed a work-item. There are many pros and cons to both GPU API options; however, a discussion of these is beyond the scope of this textbook, and the choice of API is left to the programmer’s preference. For ease of writing, the main focus of this chapter and the next will be CUDA, given its wide use in GPU quantum chemistry codes.

1.4.3 Suitable Code for GPU Acceleration

As GPUs were originally designed for applications involving graphics rendering, the lack of sophisticated hardware limited the number of applications suitable for acceleration using the historical GPU hardware design. The first programmers to attempt general-purpose applications on a GPU had the arduous task of representing their calculations as triangles and polygons. Although modern GPU hardware design has become more forgiving to the general-purpose application, it is still necessary to ensure that an algorithm is suited to the hardware in addition to considering Amdahl’s or Gustafson’s Law in order to achieve a performance gain.

Usually, only a selection of the entire code will be suitable for GPU execution, as illustrated in Figure 1.7. The rest of the code is executed on the CPU as normal, although the cost of transferring data between CPU and GPU often means that it is desirable to execute as much of the compute-intensive code as possible on the GPU.

Careful consideration of the desired code for porting is required before any development proceeds. Code that has been poorly assessed with respect to the hardware could see no performance improvement or even a performance decrease after porting to GPU, compared with running the entire code on a CPU. One aspect to consider is the SIMT (single-instruction, multiple-thread) architecture of GPUs and how that reflects on code suitability. Threads on a GPU are spawned in blocks (and on Nvidia hardware executed in batches of 32 termed warps), which are logically mapped to the physical GPU

![Figure 1.7 Illustration of a GPU working in conjunction with a CPU as work is offloaded and computed on a GPU concurrently with CPU execution [24]](image-url)
cores. The threads are designed to be lightweight and are scheduled with no overhead. The hardware does not execute the threads in a block independently. Instead, all threads execute the same instructions, and conditional branching can thus be detrimental to performance because it holds up an entire block of threads. Because of the nature of the threads working concurrently within a thread block, it is beneficial to reduce the number of idle threads by creating many more threads than there are physical compute cores on the GPU. Communication between threads is also limited and can be a source of poor performance. Therefore, fine-grained parallel problems, such as those that are data-parallel with few dependencies between data elements and few divergent branches, most successfully exploit a GPU’s hardware.

Another consideration when assessing a code’s suitability for GPU acceleration is memory access latency and bandwidth. Memory access latency is a large overhead when porting to a GPU, as the GPU cannot access memory on the CPU at the same speed as the local GPU memory, and vice versa. Data have to be explicitly transferred, or some form of unified addressing has to be used, both of which introduce overheads. The bandwidth between the GPU and the CPU memory is much lower than the bandwidth between CUDA cores and GPU memory. For example, the theoretical global memory bandwidth for the Nvidia Tesla K40 card is 288 GB/s, and the theoretical unidirectional bandwidth between the CPU and the GPU is only 16 GB/s (PCIe Gen 3 × 16). Memory access through the PCIe interface also suffers from latencies on the order of several thousand instruction cycles. As a result, transferring data between the CPU and the GPU can cause a bottleneck in performance. Arithmetic-intensive code is required in order to hide the memory access latency with calculations. Well-suited code will have a high degree of arithmetic intensity, a high degree of data reuse, and a small number of memory operations to reduce the overhead of data transfers and memory accesses.

An additional method of hiding the memory access latency is to run code of a large problem size. In order to reduce the start-up costs of allocating and freeing memory when transferring data from CPU to GPU, it is beneficial to send one large data packet as opposed to several smaller data packets. This means that the performance of code with a large problem size will be affected less by the start-up costs of CPU–GPU data transfer, as a proportion of the overall execution time, compared to code with a smaller problem size incurring the same start-up costs.

A desirable feature of a code, where not all compute-intensive regions have been ported to the GPU, is the ability to be able to overlap code execution on both the CPU and the GPU. One aim when parallelizing any code is to reduce the amount of idle processor time. If the CPU is waiting for kernels to be executed on a GPU, and there are CPU-intensive portions, then the performance of the code will suffer as the CPU spends time idling.

1.4.4 Scalability, Performance, and Cost Effectiveness

The GPU’s design makes it highly scalable, and in most cases effortlessly scalable, as the creation, destruction, and scheduling of threads are done behind the scenes in hardware. The programmer need not get bogged down with processor counts when planning a decomposition strategy. Algorithms can be decomposed according to the number of data elements, as opposed to the number of processors in a system, making applications effortlessly scalable to thousands of processor cores and tens of thousands of concurrent threads. The programmer need only concern himself/herself with the granularity of the decomposition strategy. The programming model abstracts multiple levels of parallelism from the hardware: that is, on the thread level, fine-grained data parallelism can be achieved, whereas on the kernel level a problem can be divided into more coarse-grained tasks. The ported GPU program will then scale to any number of processors at runtime. Not only does this result in a highly scalable programming/architectural model but it also lessens the difficulty for the programmer in designing scalable algorithms.
The main reason for using a GPU is to boost a program’s performance. This can be done using multiple CPUs (as mentioned earlier), however, the scope for performance improvement using a GPU is often far greater for many applications due to the larger memory bandwidth of GPUs. The potential for performance improvement is demonstrated in the next section using two applications that have seen large performance improvements after GPU acceleration (see Section 1.5).

The way the performance of GPU-accelerated programs is assessed is crucial in providing end users with realistic speedups. Obviously it is beneficial to compare the performance of a GPU with a single CPU, but what about multiple CPUs? As the traditional approach to parallelizing code is through multiple CPU cores, it seems only fair that a comparison of these two parallel implementations (GPU vs multicore) should be made in order to assess the real gain from using a GPU. One important factor to include in a performance comparison with multiple CPUs is the cost effectiveness of the improvement. The cost effectiveness of using a GPU depends largely on the type of application and the scalability of the algorithm to many processor counts. Buying time on supercomputers can be costly, and building your own supercomputer is unfeasible for the majority. A GPU puts the processing power of a compute cluster within the reach of anyone willing to spend the time modifying the code. Not only do GPUs provide supercomputing power to desktop computers, but they also do it affordably thanks to their energy efficiency. It is no coincidence that the top 10 computers on the Green500 list (a list that ranks the world’s supercomputers by energy efficiency) published in November 2013 contain Nvidia GPUs, as the performance per watt of a GPU is very desirable in comparison with a traditional CPU.

1.5 GPU-Accelerated Applications

Many real-word applications have already experienced massive performance improvements from GPU acceleration. Scientific domains benefiting from GPU technology include the defense industry, oil and gas, oceanography, medical imaging, and computational finance [25, 26]. This section showcases two GPU applications, in wildly different fields, not included in the later chapters of this book, providing additional motivation for parallelizing code on GPUs. The first to be discussed is the classical molecular dynamics (MD) software package Amber, and the second is the rendering and video-editing package Adobe Premiere(R) Pro Creative Cloud (CC).

1.5.1 Amber

Amber [27] is a software suite for molecular dynamics (MD) simulations for biomolecules, which can be used to study how biomolecules behave in solution and in cell membranes. It is used extensively, among others, in the drug discovery to determine the mechanism by which various drugs inhibit enzyme reactivity.

After porting and optimizing in CUDA for Nvidia hardware, the GPU implementation of the PMEMD molecular dynamics engine in the Amber software package is now the fastest simulation engine in the world on commodity hardware. For a simulation of the enzyme dihydrofolate reductase (DHFR), illustrated in Figure 1.8, containing approximately 23,500 atoms, it used to take, in 2009, one of the fastest supercomputers in the world to simulate ~50 ns in one day. That is one full day of computer simulation for a mere 50 billionth of a second in the life of a biomolecule. Today, 5 years on, with a single desktop containing four Nvidia Tesla Kepler K40 or similar GPUs, an aggregate throughput of over 1 μs/day can be achieved. The GPU code design takes advantage of a patented approach to lock-free computing via the use of warp-level parallelism [28], which applies algorithmic
tricks to eliminate the need for locks in parallel pairwise calculations. It also reduces the CPU–GPU data transfer overhead by carrying out the entire simulation on the GPU, communicating back to the CPU just for I/O, and uses PCIe peer-to-peer communication to eliminate CPU and motherboard chipset overhead when running across multiple GPUs. In order to achieve the best performance out of the Nvidia Kepler, Quadro, and GeForce series of GPUs, a mixed precision model, termed SPFP [29], was implemented to capitalize on the single-precision and integer atomic performance of the hardware without sacrificing simulation accuracy. The result is supercomputing capability that is accessible for every researcher at minimal cost – fundamentally advancing the pace of research involving MD simulation.

Example performance data for Amber v14 is shown in Figure 1.9, where the 90,906-atom blood clotting protein, Factor IX, has been simulated on various hardware. Amber 14 running on a single K40 GPU achieves a throughput of almost 40 ns/day, while using four K40 GPUs in tandem it can obtain a throughput on a single simulation of almost 74 ns/day. Compare this to the 5.6 ns/day achievable from running on two Intel Xeon E5-2670 CPUs (16 CPU cores), and the benefit of using GPUs to run Amber becomes obvious. A single GeForce GTX Titan Black (a gaming card that retails at around $1000) obtains over 42 ns/day and is thus capable of over 7× the performance of a single (16 cores) CPU node.

In addition to raw performance, there are also cost efficiency considerations for running Amber 14 on GPU(s), which can be demonstrated using the same FactorIX benchmark. The cost differential – considering for now just the acquisition costs and ignoring the running costs – between a high-end GPU workstation and the equivalent CPU-only cluster required to obtain the same performance is a factor of approximately 45×. This factor can be calculated, for example, by considering a GPU workstation consisting of a single node with four GTX Titan Black cards, costing approximately $7000, and the hardware specification of a CPU-only cluster required to obtain equivalent performance. The GPU workstation can achieve an aggregate throughput on four FactorIX simulations of 168 ns/day (42 ns/day per simulation). To obtain equivalent performance using
Figure 1.9 Computational performance of classical molecular dynamics simulations with the CPU and GPU versions of Amber on varying hardware configurations measured in nanosecond per day. The results are for a standard benchmark (Factor IX enzyme in explicit solvent, 90,906 atoms, NVE ensemble, 2 fs time step) [30]

CPU-based hardware, consisting of dual 8 core CPU nodes, requires a high-end QDR Infiniband interconnect between nodes and a total of 16 nodes per simulation (256 cores), since scaling across multiple nodes is never linear due to communication overhead, for a total cluster size of 64 nodes (for the four individual simulations). The cost of such a cluster, at the time of writing, is approximately $320,000.

Not only does using GPUs for MD simulations have a positive impact on acquisition costs but it also greatly reduces running costs, a substantial fraction of which, for large CPU clusters, is the cost of power. Saving in power also translates into more environmentally friendly computing. As mentioned in Section 1.4, a GPU is more energy efficient than a CPU, helping to promote greener science. To illustrate, compare a single Factor IX simulation running on a single Nvidia GeForce GTX Titan Black GPU against the same simulation running on a dual-socket Intel E5-2670 system. The Titan-Black GPU is rated at 250 W peak power consumption, but this does not include overhead in the node itself, such as case fans, hard drives, and the CPU idle power. Measuring the total node power consumption for the Factor IX simulation gives a power draw of approximately 360 W for a throughput of 42.06 ns/day. This equates to a power consumption of ~0.20 kWh/ns of simulation.

If, instead, one uses the Dual E5-2670 CPUs in this system, then the measured power consumption is approximately 359 W per node, with a single node yielding a throughput of 5.6 ns/day giving a power consumption of ~1.54 kWh/ns of simulation. In this example, the GPU version of AMBER provides a 7.7x improvement in power efficiency.

In conclusion, the GPU-accelerated Amber MD program enables faster simulations or longer simulation times than on a CPU at greater power efficiencies. It includes the majority of the MD functionality that is available on a CPU but at a fraction of the financial and environmental cost.
1.5.2 Adobe Premier Pro CC

Adobe Premier Pro CC is a GPU-accelerated software suite used primarily to edit videos offering interactive, real-time editing and rendering. Adobe teamed up with Nvidia to create high-speed functionality though the use of Nvidia GPU cards and the CUDA programming platform. Included in the Adobe Premier Pro CC are features such as debayering (process of converting a Bayer pattern color filter array image from a single sensor to a full RGB image), feathered masking (applying a mask to a frame), and master clips effects (allowing cutting, sticking, and reordering of frames). Figure 1.10 shows the performance data for rendering a high-definition (HD) video with the complex Mercury Playback Engine at a resolution of 720p on Nvidia Quadro GPUs, with respect to a Dual Intel Xeon E5 processor (16 cores in total). The cost of the graphics cards range from around $500 for a Quadro K2000, offering nearly a 5× speed-up, to $5000 for a Quadro K6000, capable of rendering over 16× faster.

What is even more impressive about the GPU-accelerated Adobe suite is the speed at which ray-traced 3D scenes can be rendered through their After Effects CC product. By equipping a desktop with a single Quadro K6000, the user can benefit from ~28× speedup when compared to using a Dual Intel Xeon E5 processor (16 cores in total) alone (see Figure 1.11). If the user happens to have an additional $5000 to spend on computer hardware, his or her 3D ray tracing workflow can be completed nearly 40× faster.

It is no surprise that a graphics rendering package such as the Adobe Premier Pro CC suite can achieve performance improvements of such a magnitude from GPU acceleration, given that the original design of a GPU was for such purposes. However, this serves as a great showcase of the capabilities of GPUs and their ability to accelerate applications in a way that is affordable to the masses. The purpose of this book is to explore in detail the successes that have been achieved and survey the current state of the art in accelerating electronic structure calculations on GPUs.

![Figure 1.10](image.png)

**Figure 1.10** Performance acceleration for Adobe Mercury Playback Engine on GPUs. System configuration: Adobe Premier Pro CC, Windows 7 – 64-bit, Dual Intel Xeon E5 2687 W 3.10 GHz CPUs (16 total cores). Test consists of HD video workflow with complex Mercury Playback Engine effects at 720p resolution. Results based on final output render time comparing noted GPU to CPU [31]
Figure 1.11 Performance data for Adobe After Effect CC Engine on GPUs. System configuration: Adobe After Effects CC, Windows 7 – 64-bit, Dual Intel Xeon E5 2687 W 3.10 GHz CPUs (16 total cores). Test consists of live After Effect CC scenes with 3D layer, comparing time to render ray-traced 3D scene on noted GPU versus CPU [32]

References

20 Electronic Structure Calculations on Graphics Processing Units


