Leveraging Parallel Processing to Accelerate Large-Scale Simulations on GPUs

Brad Suchoski, Heidi Gurung, Prasith Baccam
Innovative Emergency Management, Inc. (dba IEM)
Bel Air, MD
bradley.suchoski@iem.com, heidi.gurung@iem.com, sid.baccam@iem.com

Steve Stage
IEM
Baton Rouge, LA
steve.stage@iem.com

ABSTRACT

Simulations, data analysis, and artificial intelligence (AI) can be computationally intensive, but you don't necessarily need a supercomputer to tackle your problems. Initially designed to make video games look better, graphics processing units (GPUs) are now being used for powerful research. Our presentation will provide background on the key differences between GPUs and traditional central processing units (CPUs), describe different types of problems that are best for each type, and give examples of our own work with GPU processing. Our first example is source term reconstruction, where we try to use detector data and weather data to calculate the most likely release location, quantity, and duration. We also discuss how we use GPU processing on a Markov Chain Monte Carlo (MCMC) method to tackle problems that were previously thought to be too computationally intensive and slow. We used the MCMC technique to develop epidemiological models that provide projections for future COVID-19 projections to help decision makers. Our projections were done for approximately 400 jurisdictions, and running 4 million simulations per jurisdiction, our GPU MCMC technique completed over 1.5 billion simulations in under 20 minutes using a cloud computing solution. Finally, we coded a GPU implementation of a plume dispersion model. Running on a single-threaded CPU, 300,000 plume simulations were completed in 319 seconds. Executing 12 million plume simulations on 4 GPUs was completed in 52 seconds, approximately 247 times faster than the single-threaded CPU simulations. While GPUs are not a silver bullet that can solve all problems faster, we discuss the types of problems that are suitable for GPU processing in the hope to inspire others to use this technology to solve challenging problems.

ABOUT THE AUTHORS

Brad Suchoski, M.S. is a computational scientist at IEM whose work focuses on using graphics processing units (GPUs) to accelerate simulations and to improve research. Mr. Suchoski was a pioneer in using CUDA, a parallel computing platform and programming model developed by NVIDIA, to produce 50x to 100x speedup by leveraging GPU processing. He continues to focus on using GPUs in high-performance computing, artificial intelligence, and machine learning.

Heidi Gurung, Ph.D. is a data scientist at IEM with a focus on bioinformatics with research in the application of machine learning to DNA analysis, epidemiological models, and infectious diseases. Dr. Gurung has extensive experience in development and research of machine learning approaches to regression and classification problems in various fields, including epidemiology and sound (music) as well as experience software development, software design, implementation, evaluation, and system administration.

Steve Stage, Ph.D. is a senior scientist at IEM with a degree in meteorology and expertise in agent fate and plume dispersion modeling. Dr. Stage has developed numerous mathematical models that include the effects of a terrorist chemical or biological attack, which can be used to evaluate the effectiveness of sensor networks, vaccination, medical assets, and other resources in reducing the impact of the attack.

Prasith "Sid" Baccam, Ph.D. is the manager of the Emerging Technologies group at IEM with a dual degree in applied mathematics and immunology. Dr. Baccam's work focuses on modeling and simulation of disease outbreaks and medical consequences following hypothetical bioterrorism or emerging infectious diseases. He develops key simulations and decision support tools, specializing in public health, disaster response, and medical countermeasures to enhance data-driven decision making and improve modeling assumptions.

Leveraging Parallel Processing to Accelerate Large-Scale Simulations on GPUs

Brad Suchoski, Heidi Gurung, Prasith Baccam
Innovative Emergency Management, Inc. (dba IEM)
Bel Air, MD
bradley.suchoski@iem.com, heidi.gurung@iem.com, sid.baccam@iem.com

Steve Stage
IEM
Baton Rouge, LA
steve.stage@iem.com

INTRODUCTION

Reviewing papers presented at past I/ITSEC conferences that involved graphics processing units (GPUs), we found that most articles focused on using GPUs for rendering images to ensure that vegetation, terrain, reflections, illumination, shadows, and textures were as realistic as possible. The use of GPUs has pushed the boundaries of mixed, augmented, and virtual reality – blurring the line between real life and computer graphic simulations. The authors of this paper, however, leverage the computational power of GPUs for non-specialized computations, also known as general-purpose computing on GPUs or GPGPU computing. While "non-specialized computations" may sound simplistic and mundane, GPGPU computing can be used as part of high-performance computing to advance and benefit computationally intensive research.

Early GPUs were specifically designed and developed to offload the graphics processing function from the central processing unit (CPU). While GPUs were originally meant as a specialized hardware component for computing graphics and rendering images, modern GPUs are programmable processors with thousands of processing cores that can run simultaneously with massive parallelism capable of real-time processing and analysis of enormous datasets. GPGPU processing is fundamentally a software concept that requires specialized programming, hardware, and GPUs to facilitate massive parallel processing to substantially speed up high-volume calculations. In our personal experience, we have been able to achieve acceleration of simulations by approximately 50, 100, and 250 times through GPGPU programming.

The objective of this paper is to introduce the reader to the powerful potential of GPGPU programming with the hope of inspiring them to use this technique to solve challenging problems. To assist in this effort, we discuss different types of problems that are well-suitable for GPGPU processing. GPGPU programming, however, is not a silver bullet that can accelerate all problems. Subsequently, we also describe problems that are not well-suited for GPGPU programming. While some problems may initially appear ill-suited to GPGPU programming, creative thinking and high-level algorithm changes may allow you to transform the problems to take advantage of GPUs.

An understanding of the differences between CPUs and GPUs requires us to touch on the architecture of these different processing units. Figure 1 illustrates the main components of CPUs and GPUs: dynamic random access memory (DRAM); Cache for data caching; Control unit designed for flow control; and arithmetic logic unit (ALU) for data processing. A computer core consists of the control unit that has one or more ALUs and registers (very fast storage, not shown in Figure 1). CPU cores and GPU cores, however, differ in their ability to process instructions and data, so it is not fair to compare just the number of cores in CPUs and GPUs. The differences between CPUs and GPUs are highlighted in their architecture and explain the strengths and weaknesses of both processing units, as shown in Table 1. A more detailed description of CPU and GPU differences is provided in a subsequent section of this paper.

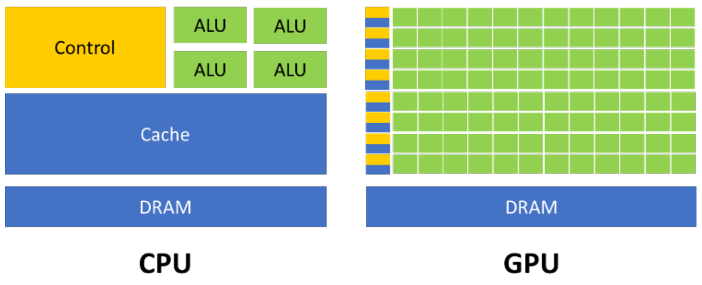

Figure 1. CPU and GPU Architecture

Table 1. Comparison of CPU vs GPU advantages

	CPU	GPU
Strength:	Serial processing	Parallel processing
Optimized for:	Low latency (fast speed)	High throughput
Compute Cores:	Few "heavyweight" cores	Many "lightweight" cores
Cores Capable of:	Large diverse instruction set	Small highly optimized instruction set
Memory:	Large memory; hardware-managed cache	Smaller memory; hardware- and software-managed cache (shared memory)

In a 2005 I/ITSEC paper focused on GPUs, the authors estimated peak computational performance and found that a CPU could complete approximately 6 billion floating-point operations per second (gigaflops or GFLOPS) while a GPU could complete approximately 48 GFLOPS (Verdesca et al., 2005). With advances in computational power, most CPUs today have 2-8 cores while GPUs may have hundreds or thousands of cores. Examining the top-end gaming CPU, the Ryzen Threadripper PRO 5995WX, has 64 cores running 128 threads at a maximum speed of 4.5 GHz, which is capable of performing approximately 9.22 trillion floating-point operations per second (teraflops or TFLOPS) (CPU benchmarks, 2022). The top-end gaming GPU, the NVIDIA GeForce RTX 3080, has 8,704 cores running at a maximum speed of 1.71 GHz and is capable of performing approximately 29.76 TFLOPS (James, 2022).

In the example with the top-end processors, the GPU is about three times faster than the CPU. In practice, however, the GPU is much faster since achieving peak utilization on a CPU is extremely difficult while it is relatively more straightforward on a GPU. The clock rate for the CPU, 4.5 GHz, is four times faster than the GPU clock rate, 1.71 GHz, highlighting the latency advantage of CPUs. The GPU has 136 times more cores than the CPU, and along with the other factors listed in Table 1, demonstrates why it better suited for parallel processing and high throughput.

COMPARISON OF GPUs VS CPUs

Hardware Differences

Both modern day CPUs and GPUs consist of multiple cores and can perform some degree of parallel computing where progress is made toward completing multiple compute tasks concurrently. The key difference between CPU and GPU processors is that this type of parallel processing became common in CPU design only in the last decade or two while GPUs were designed from the start with this type of parallel processing in mind. Today's multi-core CPUs consist of a comparatively small number of complex cores, each of which is highly optimized to complete single independent serial tasks in as little time as possible. GPUs, on the other hand, consist of an exceptionally large number of simple cores that are designed to run cooperatively to complete a single problem that can be broken down into millions or more individual simple tasks.

Table 2 provides an overview of the number of cores on each type of processor. Nvidia uses the term "core" slightly differently than its closest analogous component on a CPU. Both types of processors support vectorized instruction sets where multiple data lanes are computed by a single instruction. On CPUs, these vectorized instructions sets are optional and include instructions that are dependent on their processors. For example, the AVX2/AVX-512 instructions are used on x86 processors, NEON instructions are used on ARM CPUs, and AltiVec instructions are used on IBM Power processors. On GPUs, only vectorized instructions are used and are the core instruction sets. The number of data lanes provided by each instruction set varies and is listed in Table 2.

The most similar component on a GPU to a CPU "core" is the Symmetric Multiprocessor (SM) on Nvidia GPUs and the Compute Unit (CU) on AMD GPUs. While there are several significant differences between these components, those differences are beyond the scope of the current discussion. What is important is that each Core/SM/CU can run one or more instructions, each of which is connected to multiple data lanes so that multiple pieces of data can be computed concurrently in parallel. The total number of FP32/INT32 lanes per chip is what Nvidia refers to as CUDA Cores (AMD calls them Stream Processors), and it is listed in the right-most column of Table 2.

Table 2. Comparison of the number of cores, symmetric multiprocessors (SM), and compute units (CU) on various flagship CPUs and GPUs.

Processor/Chip	Cores/SMs/CUs	Max FP32/INT32 Lanes per "Core"	Max Total FP32/INT32 Lanes
Intel Xeon Platinum 8380 CPU	40	16	640
AMD EPYC 7773X CPU	64	8	512
Nvidia GA100 GPU Chip	128	64	8,192
Nvidia GH100 GPU Chip	144	128	18,432
AMD Instinct MI250X GPU	220	64	14,080

We can see from the table that the total number of data lanes available on GPUs is roughly an order of magnitude greater than that of CPUs. In addition to these physical cores, many modern processors support some sort of Simultaneous Multithreading (SMT) whereby there are multiple sets of registers connected to and share a single physical cores to create multiple logical cores. Most readers will likely be most familiar with Intel's hyper-threading, which is their proprietary implementation of SMT on Intel CPUs that provides two logical cores per each single physical core. IBM introduced SMT8 on its POWER8 architecture that provides eight logical cores per single physical core. Many other CPU vendors provide similar SMT implementations on their chips.

While SMT is an enhancement available on some CPU chips, it is an essential feature used by all modern GPU architectures used to hide the long latency times associated with main memory access. Modern GPUs have a much higher maximum main memory bandwidth than their CPU counterparts, reaching as high as a few terabytes per second (TBps) on GPUs as opposed to a few hundred gigabytes per second (GBps) on CPU. However, GPUs also have a higher memory latency time reaching into the hundreds of clock cycles as compared to tens of cycles on CPUs from the time a memory load/store instruction is issued until it is retired. To cope with these high latency times, GPUs use SMT to keep a pipeline of several threads active per physical core and interleaves each thread's access to the underlying physical hardware components as needed.

Unlike CPUs, GPUs do not use a defined set of registers and a set number of logical cores per physical core. They instead have a register file, and the total number of threads active is limited by the register file size and the number of registers per thread required for the currently running kernel. If the compute kernel being run is too complex and requires many registers, then the hardware will not be able to hide the memory latency and results in degraded performance. There is a point beyond which performance will be limited by other factors such as memory bandwidth or compute capacity, and further increasing the number of threads will not result in improved performance. This sweet spot has various complex dependencies on the nature of the code being run, but it is typically somewhere between 2 and 32 active threads per core.

All these factors imply that fully utilizing a modern GPU requires a minimum of several tens of thousands up to hundreds of thousands of individual threads or tasks to be active at any given time. Furthermore, there is a limit to the complexity of each thread's task. More complex tasks tend to increase register usage of the task, and thus limit the number of concurrent threads and the GPU's ability to effectively hide the latency of memory operations. Thus, the types of problems that are ideal for execution on GPU are ones where the inner most loops of the algorithm are simple, have no data dependencies, and execute tens of thousands to hundreds of thousands of iterations.

Software Differences

Due to the considerable differences in architecture between CPUs and GPUs, writing software that effectively leverages the strengths of each type of processor can be challenging. At its foundation, any programming model for writing effective GPU software needs to be able to execute, at a minimum, many thousands to millions of individual tasks in parallel. Shared-memory parallel programming models have existed for CPUs for many decades. Popular examples of these include the POSIX threads and Threading Building Blocks (TBB) libraries, or the directives-based OpenMP programming model. POSIX threads, TBB, and similar libraries tend to promote coarse-grained, task-based parallelism where a problem is decomposed into a small number of large complex tasks. This type of parallelism is ideal for execution on CPUs, but it results in large complex tasks and does not usually scale to the number of threads required to fully utilize a GPU. Starting with version 4.0, OpenMP does support offloading blocks of code for execution on GPU, and some simple algorithms, if carefully written, can be made to scale up to the sizes required by

GPU. OpenMP does, however, include features for coarse-grained, task-based parallelism, and so it is not guaranteed that software written with it will be able to utilize a GPU.

To contend with these issues, Nvidia revolutionized the GPGPU industry in 2007 with the introduction of the Compute Unified Device Architecture (CUDA) parallel computing platform. CUDA provides extensions to C, C++, and Fortran programming languages that give direct access to the GPU's virtual instruction set, and for the first time allows programmers to write general purpose GPU kernels and programs easily and efficiently. The platform is built on the Single Instruction, Multiple-Thread (SIMT) programming paradigm where a single device function called a kernel is written, and that same function is executed on thousands to millions of individual pieces of data in parallel. The programming model is similar to Single Program, Multiple Data (SPMD) programming models, which are popular in distributed memory CPU platforms such as MPI. CUDA differs from SPMD because CUDA uses shared memory, operates at the function level instead of program level, and was designed from the ground up to scale to the level of parallelism required to efficiently utilize a GPU. It does this by using a hierarchy of threads that define rules for when and how the threads can synchronize their access to the shared memory.

CUDA is a closed platform only capable of running and executing code on Nvidia GPUs. As CUDA continued to rise in popularity, similar platforms from other vendors were developed that allowed execution of general-purpose code on a wide range of GPUs, digital signal processors (DSPs), and field-programmable gate arrays (FPGAs). One of the earliest and most popular frameworks was OpenCL. Originally developed by Apple and now maintained by the Khronos Group, OpenCL is an open standard defining a C-like kernel-based programming language similar to CUDA. It has implementations from and can run on hardware from AMD, ARM, IBM, Intel, Nvidia, Samsung, and Xilinx among others. Although this portability across hardware vendors is desirable, the performance of code written in OpenCL does not always carry over from one platform to another. For example, a 2011 study by D-Wave Systems Inc found that Nvidia's OpenCL implementation was 16%-63% slower than CUDA when comparing two nearly identical implementations of a Monte Carlo simulation application (Karimi, Dickson, & Hamze, 2010).

Other directives-based standards such as OpenACC and OpenMP provide extensions to C++ and Fortran for portable execution of code on GPUs, but they often suffer from similar performance bottlenecks to OpenCL. To address these issues, AMD released their Radeon Open Compute platforM (ROCm) in 2016. Included as part of ROCm is AMD's Heterogeneous-Computing Interface for Portability (HIP), a dialect of C++ specifically designed to be similar to CUDA and to facilitate the porting of existing CUDA code to run on AMD GPUs without loss of performance. The platform even provides a tool that can automate much of the code conversion from CUDA to HIP and vice-versa.

Types of problems that are NOT well-suited for GPUs

Because of the hardware and software differences between the types of processors, there are several classes of problems that are better suited to each one. The most significant and major differentiating factor between these classes of problems is that CPUs are generally optimized toward flexibility and the ability to efficiently multitask between a few mid-range computationally intensive tasks, while GPUs are typically optimized specifically toward parallel processing and high throughput on exceptionally large, highly computationally intensive tasks. For example, a problem such as database creation, querying, or updating that involves the processor spending much of its time waiting on file or networking I/O operations to complete would not be a good candidate for GPU acceleration. That is not to say that any problem that involves file or network I/O is a poor candidate. Once a dataset is loaded into memory from a database query, performing operations on that data such as extracting features, transforming, or summarizing the dataset are often highly parallelizable, very computationally intensive tasks that are good candidates for GPU acceleration provided the dataset can fit into GPU memory.

Another category of potentially poor candidates for GPU acceleration are single tasks or small, real-time problems, with the keyword here being small in terms of number of threads. GPUs were initially developed specifically for the real-time problem of rendering 3-D graphics at high frame rates. They are only efficient at that problem because of the enormous number of pixels, which can reach into the tens of millions per frame at modern resolutions. If the number of pixels were smaller, then GPUs would actually be a poor choice compared to CPUs. This is because GPUs typically have a lower base clock rate compared to CPUs, as well as longer latencies associated with accessing main memory. Also, most modern GPU chips lack many of the hardware level optimizations often present on CPUs such as instruction level parallelism, out-of-order execution, branch prediction, speculative execution, and memory prefetching that allow CPUs to excel at executing single tasks. GPUs, on the other hand, are designed to dedicate more transistors to providing as many ALUs as possible to maximize throughput as opposed to these optimizations. Because of all these considerations, for any individual thread of execution the time between when the thread begins its work and when it finishes it will always be greater on GPU than on CPU. A GPU's advantage comes in its ability to keep many more threads at a time in flight so that more threads per unit time can complete their work on GPUs as compared to CPUs. There is a cutoff point in the number of concurrent executing threads below which CPUs will outperform GPUs.

Types of problems that ARE well-suited for GPUs

Many modern scientific and data analysis problems use enormous sets of data and require computation of the exact type that GPUs were designed for. Consequently, there is no shortage of existing real-world examples of applications that have benefitted from GPU acceleration. Some examples include:

- Deep learning: neural networks
- Drug design: structure-based drug design
- Seismic imaging: oil and gas industry to detect oil reservoirs
- Automotive design: physics based and fluid flows
- Astrophysics: interaction of N-body simulations
- Image processing: medical (CT, MRI, x-ray)), satellite, etc.
- Weather forecasting

A growing number of military and defense projects based on GPGPU technology have already been deployed, including systems that use the advanced processing capabilities for radars, image recognition, classification, motion detection, encoding, etc. Private industry has made this possible by designing ruggedized high performance computing (HPC) computers with embedded GPUs. Adhering to SWaP (size, weight, and power) principles, these ruggedized HPC computers are designed to be small, light, and powerful enough so they can be deployed to "in theater" locations and used on ground-based vehicles as well as shipboard and aircraft applications. These GPU-based systems utilize parallel-processing for analyzing video, imagery, and sensor data collected from unmanned aerial vehicles (UAVs), eliminating the need for off-site processing, which would result in valuable time lost. The parallel-processing can, for example, reduce the time required to analyze 24 hours of HD (1080p) video captured from a UAV to under one hour or process satellite images to map a 200 square mile city in detail in less than 20 seconds (Travis, 2018). While these in theater applications are very important to the defense industry, we describe in this paper some examples of GPGPU applied to general research topics that may help inspire others to identify computationally intensive processes that can benefit from parallel processing with GPUs, some of which may eventually end up in applications in the field.

REAL-WORLD EXAMPLES

Example 1: Physics-based agent fate modeling

Our first example comes from an application IEM developed to accelerate computations of transport and diffusion in atmospheric dispersion applications. The inspiration for this work was our earlier study of source term reconstruction, where we used detector data and weather data to calculate the most likely release location, quantity, and duration. In this application, the end users were interested in collecting and analyzing statistics on the result of up to millions of individual runs of the Second-order Closure Integrated PUFF (SCIPUFF) model. The SCIPUFF model is a Gaussian puff method first introduced in 1980 (Bass, 1980), which is a numerical technique used to solve the dispersion equations for time-dependent, three-dimensional concentration fields resulting from the release of a chemical into the atmosphere. It does this by representing the concentration field at any given time as a collection of independent puffs that moves through the atmospheric velocity field and grows in time according to second-order turbulence closure theories (Donaldson, 1973; Lewellen, 1977).

The original SCIPUFF model provided many advanced options for modeling cases that include effects from special materials such as dense gases, aerosols, particulate materials as droplets, nuclear weapons materials, and other radiological materials. It can also model several environmental factors such as observed wind velocity fields as input, terrain profiles, and dispersion over a vegetative or urban canopy layer. While there is no fundamental reason preventing these effects from being incorporated into a GPU-accelerated solution, simple releases of non-radiological buoyant gases with a constant wind velocity field over flat ground with no canopy cover was sufficient for this specific project. The fundamental governing equations for the concentration of an individual puff at a point in space **x,** with no surface or inversion layer reflections, is given by

$$c(x) \ = \ \frac{Q}{(2\pi)^{\frac{3}{2}}\left(Det(\sigma)^{\frac{1}{2}}\right)} \exp\left[-\frac{1}{2}\sigma_{ij}^{-1}(x_i - \overline{x_i})(x_j - \overline{x_j})\right]$$

where Q is the mass released in the puff, σ is the three-dimensional dispersion coefficient matrix, and $\overline{x}$ is the puff centroid location. The dispersion coefficient matrix and puff centroid location are time-dependent variables where updated equations for their values can be derived by using the Reynolds averaging technique to solve the advection-diffusion equation for the first three spatial moments of the concentration equation given by

$$\frac{\partial c}{\partial t} + \nabla(uc) = k\nabla^2 c \ + \ S$$

In cases with complex wind velocity fields **u,** or a high degree of spatial dependence in the diffusivity parameter k, an entire release can be represented as a series of infinitesimally small puff releases over time, and the total concentration is calculated as the integral over the contribution from each of these series of puffs. Numerically, this situation is ideal for GPU acceleration because this can be approximated by summing up the contribution of a large but finite set of independent puffs, where the contribution from each puff is the same independent and sufficiently simple computation just performed on the different data for each puff.

For sufficiently large and complex models, this parallelism alone could be enough to saturate a GPU and result in higher performance than its CPU counterpart. For our use case, however, we are also interested in running several thousand or up to millions of individual scenarios each with varying input parameters and collecting statistics on the output. If performed concurrently, this additional level of parallelism drastically reduces the problem size limit where a GPU will outperform a CPU. Table 3 shows the average wall time, throughput, average time, and relative speedup of the plume dispersion runs executed on Linux and Windows machines utilizing CPUs and GPUs. The CPU implementation running on a Linux workstation completed 300,000 scenarios in 56.1 seconds (12 cores, 24 threads), while the 4xGPU version completed 12 million simulations in 52.0 seconds, a speedup of 43.2 times faster. The CPU implementation running on a Windows workstation completed 300,000 scenarios in 125.6 seconds (6 cores, 12 threads), while the 1xGPU version completed 12 million simulations in 54.6 seconds, a speedup of 23.0 times faster. Comparing the single-threaded Linux CPU and the Linux 4xGPU, we see a speedup of 247.2x.

Table 3. Speedup of plume dispersion modeling using GPUs
(relative to multi-threaded CPU using same operating system)

Configuration	# Scenarios (Thousands)	Wall time (sec)	Throughput (Scenarios/sec)	Avg. Time (µs/Scenario)	Speedup
Linux CPU Single Thread	300	319	940	1,063	0.2
Linux CPU 12 Cores 24 Threads	300	56.1	5,347	187	1.0
Linux 1xGPU	300	24.6	121,195	82	22.8
Linux 2xGPU	6,000	32.9	182,370	5.5	34.1
Linux 4xGPU	12,000	52.0	230,769	4.3	43.2
Windows CPU Single Thread	300	311.6	963	1,039	0.4
Windows CPU 6 Cores 12 Threads	300	125.6	2,389	419	1.0
Windows 1xGPU	3,000	54.6	54,945	18.2	23.0

Windows workstation configuration: 1x Intel Xeon E5-1650 v4 CPU with 32GB RAM; 1x GTX Titan Xp GPU
Linux workstation configuration: 1x Intel Core i9-9920x CPU with 128GB RAM; 4x RTX 2080 Ti GPU

Example 2: COVID-19 case projections

The next example application of GPU processing to accelerate a scientific computing problem is IEM's COVID-19 case projection model. This example is interesting because it is not a straightforward use case for GPU processing. Several design decisions about the algorithm had to be made up front, and the level of parallelism required to benefit from GPU processing is only attained when the effects of those decisions on the algorithm as a whole are carefully considered. In this section, we discuss what some of those decisions were as well as why they were made and how they affect GPU performance. The discussion that follows does not cover a number of important contributing factors to GPU performance such as thread hierarchy and synchronization, shared memory, branch divergence, or coalesced memory access to name just a few. It should, however, provide some high-level insight to the alternate way a problem needs to be thought about and approached when targeting GPU processing.

The model takes as input a time-series of some combination of COVID-19 cases, deaths, or hospitalizations for a particular county, state, or country. Using those data, the model produces prediction interval ranges for those time-series on dates in the future. It does this by starting with an epidemiological compartment model of the jurisdiction's population. Well known examples of compartment models that the reader might be familiar with include the Susceptible, Infected, Recovered (SIR) and Susceptible, Exposed, Infected, Recovered (SEIR) models.

These models divide a population into discrete categories such as Susceptible or Infected. It then defines rules for the rate that individuals will transition from one compartment to another, leading to a system of Ordinary Differential Equations (ODEs) that describes the population of each compartment over time. The equations shown below are an example of one of the most basic epidemiological compartment models, the SIR model.

$$\begin{aligned} S'(t) &= -\beta(t)\frac{S(t)I(t)}{N} \\ I'(t) &= \beta(t)\frac{S(t)I(t)}{N} - \gamma I(t) \\ R'(t) &= \gamma I(t) \end{aligned}$$

Once an underlying epidemiological model is chosen, the challenge is then to determine the parameter values of the epidemiological model that best describe the recently reported case, death, or hospitalization data for a given jurisdiction. Once those parameters are determined, then the system of ODEs can be solved for times into the future, and the solution can be used as a forecast for future cases, hospitalizations, or deaths.

Markov Chain Monte Carlo (MCMC) methods are one of the most commonly used algorithms for such parameter estimation Bayesian inferencing problems. The Metropolis-Hastings (MH) algorithm is an MCMC method where a sequence or chain of parameter samples are selected by first drawing a random jump from a proposal probability distribution. A new proposal point is given by applying this jump to the current chain location, and the new proposal point is either accepted as the next sample in the chain or rejected based on the calculated loss value of the proposal point. In our case, the loss function involves using the proposal point parameters to numerically solve the underlying epidemiological model system of ODEs and using mean square error (MSE) or a similar metric to compare the simulated number of new cases to the actual reported number.

While the MH algorithm is powerful, it presents several challenges to be effectively used on modern CPUs or GPUs. The most obvious challenge is that MH is an inherently sequential algorithm. At each iteration of the algorithm, the new proposal point is determined by using the location of the previous point in the chain, and that iteration must be completed in order and cannot be parallelized. For some problems, this might not be an obstacle. Often, the MH algorithm is used for problems where the underlying simulation required for a single loss function calculation is computationally intensive enough to fully utilize the available compute resources, but this was not the case for the system of ODEs we were solving. Some research has been done on algorithms to parallelize numerical ODE solvers, but the most common and effective methods still are sequentially calculating each timestep after the previous one has completed. At each timestep, even the most complex compartment models will typically have fewer than 100 compartments and at most a few hundred terms for the derivatives, which is far fewer than the tens of thousands to hundreds of thousands that would be required to fully utilize even a single GPU.

To take advantage of all the cores, another source of parallelism had to be introduced. For our COVID-19 projections, one source of parallelism was the fact that we were always running our analysis for more than one jurisdiction at a time. For instance, individual states and counties in the Unites States or individual countries of the European Union were often run concurrently. Since the different jurisdictions had no effect on one another, they could be run independently. This type of high-level, coarse-grained, task-based parallelism would have been sufficient for some modern CPU systems, but it was still an insufficient number of tasks for GPU systems. Furthermore, the size and complexity of each task would be far too large to effectively utilize a GPU, which benefit from low-level, fine-grained, data-level parallelism. The register usage of a GPU kernel as large and complex as solving a system of ODEs would be so high that it would limit the GPU's ability to use SMT to hide the memory and instruction latencies and would degrade the performance.

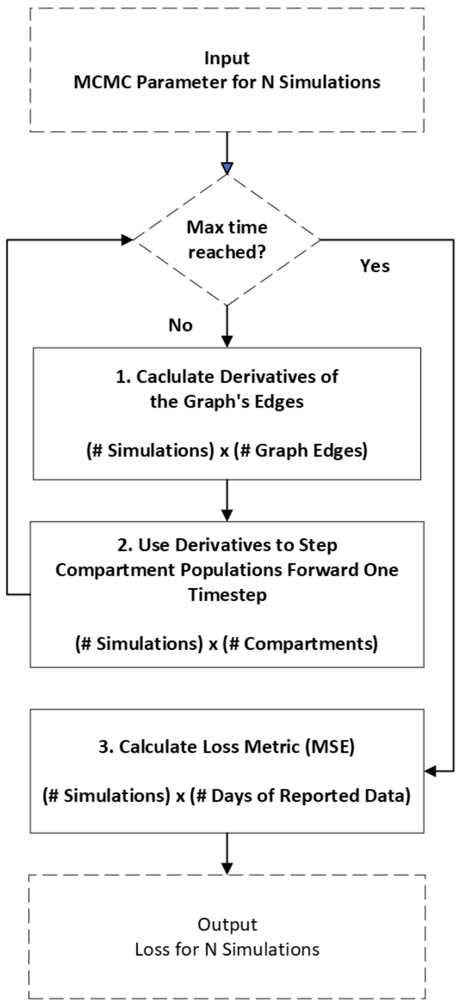
Input
MCMC Parameter for N Simulations
Max time reached?
Yes
No
1. Caclulate Derivatives of the Graph's Edges
(# Simulations) x (# Graph Edges)
2. Use Derivatives to Step Compartment Populations Forward One Timestep
(# Simulations) x (# Compartments)
3. Calculate Loss Metric (MSE)
(# Simulations) x (# Days of Reported Data)
Output
Loss for N Simulations

Figure 2. Steps involved in solving SIR or SEIR models

The task of solving this ensemble of ODE systems, one for each jurisdiction, needed to be sub-divided into a sequence of much smaller, fine-grained task batches. We shifted our focus from parallelizing from the top down, which is common for CPU processing, to parallelizing from the bottom up, starting with the lowest level loops. Figure 2 gives a high-level overview of the steps that were involved in solving a simple SIR or SEIR model using the forward Euler method. The steps for the models used in practice are much more nuanced, but this simple example illustrates the important points. The steps for each simulation are the same. Step 1 is to iterate over all of the graph edges (i.e., the terms in the ODE) and to calculate the derivative value using the current compartment populations. Step 2 is to step forward in time by iterating over the compartments and update their populations using the calculated derivatives and timestep. This process is repeated until the desired end time is reached. Step 3 is then to iterate over simulated and reported cases and calculate the MSE or other metric of simulated cases compared to reported cases.

On CPU, a single process or thread would be responsible for this entire sequence of calculations for a single simulation. On GPU, we use a more fine-grained division of work, all the way down to where the responsibility of each individual thread involves little to no iteration of loops. Each of the solid border boxes in Figure 2 represents a single GPU compute kernel executed by a 2-D grid of threads where each dimension of the grid encapsulates the iterations of a loop. For example, the two dimensions of kernel 1 iterate over all the simulations and all the graph edges while the dimensions of kernel 2 iterate over all the simulations and all the compartments, etc. Each thread of kernel 1 is responsible for only the calculation of a single term in the derivative graph for a single simulation then exits. The division of work in this manner satisfies both requirements for efficient GPU coding by having a large number of threads that are each performing tasks with low complexity.

Even with the lowest level loops bundled together into separate compute kernels, running several hundred jurisdictions concurrently does not provide enough parallelism to fully utilize a single GPU when using simple underlying epidemiological models. Simple epidemiological models will have on the order of 10 compartments/edges, which means the kernels in Figure 2 will have up to a few thousand threads, not the hundreds of thousands required. This means yet another source of parallelism needs to be introduced to effectively use our GPUs. The additional parallelism comes in the form of modifications to the MCMC algorithm. Recall that the MCMC algorithm collects a sequence of samples in a chain, collected iteratively. When the algorithm first starts, there is a "burn-in" period where the samples collected from the chain have not yet converged to the desired stationary distribution. Preliminary testing on our code indicated that this "burn-in" period could be as many as several thousand to over 10,000 iterations of the MCMC algorithm.

The next two optimizations to our algorithm focus on the addition of parallelizable work to each iteration of the MCMC algorithm to reduce the total number of MCMC iterations required to reach convergence. The first of these optimizations was accomplished by using a modification to the Metropolis Hastings algorithm known as the Multiple-try Metropolis (MTM) algorithm (Lui, Liang, & Wong, 2000). The idea for the MTM algorithm is similar to that of traditional MH except that at each iteration, the MTM algorithm selects and independently tries multiple proposal points instead of just a single point. This means that for each iteration of the MTM algorithm, each jurisdiction will require solving multiple epidemiological models instead of just one. Importantly, each of these epidemiological models can be solved and their corresponding likelihoods computed in parallel. Lui, Liang, and Wong showed that once the likelihood of each of the proposals for a single iteration is calculated, that a single proposal point can be selected as the next chain sample in such a way that the final chain converges to the stationary distribution.

The second optimization is the introduction of an Interacting Particle Method (IPM). In IPM, we simultaneously run multiple independent copies of the MTM algorithm that interact with one another only during the proposal selection process. In the traditional MH and MTM algorithms, the proposal points are drawn randomly from some proposal distribution, and the algorithms will converge faster the more closely this proposal distribution matches the stationary distribution. The idea of IPM is that the current state of the multiple independent chains will act as a small sample of the stationary distribution, and we can use them to construct a proposal distribution that is more similar to the stationary distribution, and this accelerates the convergence rate.

With the addition of MTM and IPM to our algorithm, each of the jurisdictions we run in our analysis now runs multiple complete epidemiological models per MCMC iteration. Typical values that we currently use in our analyses are 128 tries in the MTM and 32 chains in the IPM, which results in each jurisdiction running a total of 4,096 epidemiological models per iteration. This is now large enough that running even just a few jurisdictions provides the kernels with enough work to completely utilize a single GPU, and our larger analyses require us to split up the computational work among several GPUs.

To test the speed of the GPU code, we ran a benchmark of our modified MTM algorithm using cumulative COVID-19 case timeseries data collected at the county level across the US (Johns Hopkins University, 2022). We ran our analysis on a selection of 385 individual counties, the aggregate cases for all 50 US states, 3 US territories, and the aggregate for the U.S. as a whole (total of 439 jurisdictions). We collected the average runtime for several analyses, with each analysis requiring solving a total of 899,072,000 complete SEIR simulations covering a 20-day time window at 2.4-hour timesteps. Table 4 gives an overview of three hardware configurations meant to be representative of a high-end developer's workstation, a CPU-optimized HPC Node, and a GPU-optimized HPC Node. Table 5 gives the average wall time of each analysis, as well as the speedups relative to the Workstation CPU configuration and the HPC Node CPU configuration. Relative to a workstation with a single CPU, the 4xGPU workstation resulted in a 36.3x speedup, and the 8xGPU HPC node resulted in an 84.8x speedup.

Table 4. Benchmark test hardware configuration

Configuration	CPU	Main Memory	GPUs
Workstation	1x Intel Core I-9 9920X	128 GB	4x Nvidia RTX 2080 Ti
CPU HPC Node	2x Intel Xeon Platinum 8275CL	192 GB	N/A
GPU HPC Node	2x Intel Xeon Platinum 8275CL	1,024 GB	8x Nvidia Tesla Z100-SXM4-40GB

Table 5. Benchmark timing test results

Configuration	Wall Time (sec)	Speedup (Rel to WS)	Speedup (Rel to HPC)
Workstation 1xCPU	8,469	1.0x	0.67x
Workstation 1xGPU	624	13.6x	9.06x
Workstation 4xGPU	233	36.3x	24.3x
HPC Node 2xCPU	5,654	1.5x	1.0x
HPC Node 1xGPU	387	21.9x	14.6x
HPC Node 8xGPU	100	84.8x	56.5x

CONCLUSION

We have provided a synopsis of the powerful potential of using GPGPU programming for conducting research as part of high-performance computing for computationally intensive problems. The first step in utilizing GPGPU programming is simply recognizing that you need it. Many people simply accept that their computations take a long time to complete because that's all they have known. If you rely on computations or simulations that take hours or days to complete, these are the types of problems that may benefit from GPUs and parallel processing. One cannot just take legacy computer code and run it on GPUs. That legacy code typically requires recoding to take advantage of the GPU hardware and facilitate massive parallel processing. We have provided two real-world examples that highlight the processes for achieving great acceleration, 36x, 85x, and 247x speedups, in our simulations.

We encourage readers to examine the daily problems and projects that they deal with and consider if they can benefit from GPGPU programming. One can begin using GPGPU for general research and perhaps can later try to deploy the application in theater, if applicable. Ruggedized HPC computers with embedded GPUs can facilitate GPGPU processing on ground-based vehicles as well as shipboard and aircraft applications. The GPU acceleration can be used to give our warfighters the tactical advantage when it really matters.

ACKNOWLEDGEMENTS

The purpose of our COVID-19 case projections was to provide decision makers with valuable information that they could use in their jurisdictions. We thank the numerous recipients of our COVID-19 projections who helped us shape and improve our reports to be more helpful.

REFERENCES

Bass, A. (1980). Modelling long range transport and diffusion. *Second Joint Conf. on Appl. Of Air Poll. Met.*, AMS & APCA, New Orleans, LA, 193-215.

CPU Benchmarks, (2022). AMD Ryzen Threadripper PRO 5995WX. Retrieved April 24, 2022, from https://www.cpubenchmark.net/cpu.php?cpu=AMD+Ryzen+Threadripper+PRO+5995WX&id=4764

Donaldson, C. du P. (1973). Atmospheric turbulence and the dispersal of atmospheric pollutants, *AMS Workshop on Micrometeorology*, ed. D. A. Haugen, Science Press, Boston, (pp. 313-390).

James, D. PC Gamer, (2022). The best graphics cards in 2022. Retrieved April 24, 2022, from https://www.pcgamer.com/the-best-graphics-cards/

Johns Hopkins University. (2022) COVID-19 Dashboard by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University. Retrieved daily from https://github.com/CSSEGISandData/COVID-19

Karimi, K., Dickson, N.G., & Hamze, F. (2010). A Performance Comparison of CUDA and OpenCL. *Computing Research Repository - CORR. arXiv*:1005.2581.

Lewellen, W. S. (1977). Use of invariant modeling. *Handbook of Turbulence*, ed. W. Frost and T. H. Moulden, Plenum Press, (pp. 237-280).

Liu, J.S., Liang, F., & Wong W.H. (2000). The multiple-try method and local optimization in metropolis sampling. *J Am Stat Assoc.* 95:121–34.

Travis, S. EE Times, (2018). Why high-performance computing is critical to military applications. Retrieved April 27, from https://www.eetimes.com/why-high-performance-computing-is-critical-for-military-applications/

Verdesca, M., Munro, J., Hoffman, M., Bauer, M., & Manocha, D. (2005). Using graphics processor units to accelerate OneSAF: a case study in technology transition. In *Proceedings of the 2005 Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*. Paper No. 2121. Arlington, VA: National Training and Simulation Association.