Accelerating S3D: A GPGPU Case Study

Kyle Spafford[1], Jeremy Meredith[1], Jeffrey Vetter[1],
Jacqueline Chen[2], Ray Grout[2], and Ramanan Sankaran[1]

[1] Oak Ridge National Laboratory
{spaffordkl,jsmeredith,vetter,sankaranr}@ornl.gov
[2] Sandia National Laboratories
{jhchen,rwgrout}@sandia.gov

Abstract. The graphics processor (GPU) has evolved into an appealing choice for high performance computing due to its superior memory bandwidth, raw processing power, and flexible programmability. As such, GPUs represent an excellent platform for accelerating scientific applications. This paper explores a methodology for identifying applications which present significant potential for acceleration. In particular, this work focuses on experiences from accelerating S3D, a high-fidelity turbulent reacting flow solver. The acceleration process is examined from a holistic viewpoint, and includes details that arise from different phases of the conversion. This paper also addresses the issue of floating point accuracy and precision on the GPU, a topic of immense importance to scientific computing. Several performance experiments are conducted, and results are presented from the NVIDIA Tesla C1060 GPU. We generalize from our experiences to provide a roadmap for deploying existing scientific applications on heterogeneous GPU platforms.

1 Introduction

Strong market forces from the gaming industry and increased demand for high definition, real-time 3D graphics have been the driving forces behind the GPU's incredible transformation. Over the past several years, increases in the memory bandwidth and the speed of floating point computation of GPUs have steadily outpaced those of CPUs. In a relatively short period of time, the GPU has evolved from an arcane, highly-specialized hardware component into a remarkably flexible and powerful parallel coprocessor.

1.1 GPU Hardware

Originally, GPUs were designed to perform a limited collection of operations on a large volume of independent geometric data. These operations fell into to only two main categories (vertex and fragment) and were highly parallel and computationally intense, resulting in a highly specialized design with multiple cores and small caches. As graphical tasks became more diverse, the demand for flexibility began to influence GPU designs. GPUs transitioned from a fixed function design, to one which allowed limited programmability of its two specialized

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 122–131, 2010.

pipelines, and eventually to an approach where all its cores were of a unified, more flexible type, supporting much greater control from the programmer.

1.2 CUDA

The striking performance numbers of modern GPUs have resulted in a surge of interest in general-purpose computation on graphics processing units (GPGPU). GPGPU represents an inexpensive and power-efficient alternative to more traditional HPC platforms. In the past, there has been a substantial learning curve associated with GPGPU, and expert knowledge was required to attain impressive performance. This involved extensive modification of traditional approaches in order to effectively scale to the large number of cores per GPU. However, as the flexibility of the GPU has increased, there has been a welcomed decrease in the associated learning curve of the porting process. In this study, we utilize NVIDIA's Compute Unified Device Architecture (CUDA), a parallel programming model and software environment. CUDA exposes the power of the GPU to the programmer through a set of high level language extensions, allowing for existing scientific codes to be more easily transformed into GPU compatible applications.

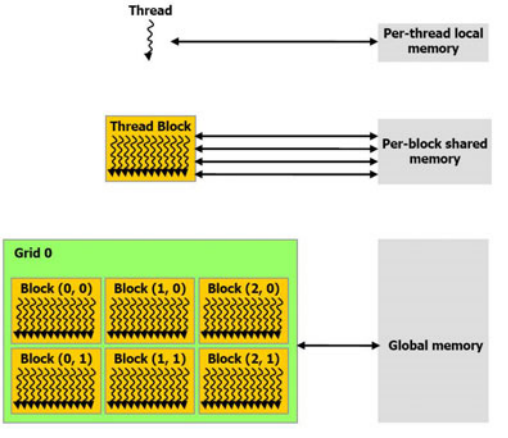

Fig. 1. CUDA Programming Model – Image from NVIDIA CUDA Programming Guide[1]

Programming Model. While a full introduction to CUDA is beyond the scope of this paper, this section mentions the basic concepts required to understand the scope of the parallelism involved. CUDA views the GPU as a highly parallel coprocessor. Functions called kernels, are composed of a large number of threads, which are organized into blocks. A group of blocks is known as a grid, see Figure 1. Blocks contain a fast shared memory that is only available to threads which belong to the block, while grids have access to the global GPU memory. Typical kernel launches involve one grid, which is composed of hundreds or thousands of individual threads, a much higher degree of parallelism than normally occurs

with traditional parallel approaches on the CPU. This high degree of parallelism and unique memory architecture have drastic consequences for performance, which will be explored in a later section.

1.3 Domain and Algorithm Description

S3D is a massively parallel direct numerical solver (DNS) for the full compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry[2, 3]. It is based on a high-order accurate, non-dissipative numerical scheme solved on a three-dimensional structured Cartesian mesh. S3D's performance has been studied and optimized including I/O[4] and control flow[5]. Still, further improvements allow for increased grid size, more simulation timesteps, and more species equations. These are critical to the scientific goals of turbulent combustion simulations in that they help achieve higher Reynolds numbers, better statistics through larger ensembles, more complete temporal development of a turbulent flame, and the simulation of fuels with greater chemical complexity.

Here we assess S3D code performance and parallel scaling through simulation of a small amplitude pressure wave propagating through the domain for a short period of time. The test is conducted with detailed ethylene-air (C_2H_4) chemistry consisting of twenty-two chemical species and mixture-averaged molecular transport model. Due to the detailed chemical model, the code solves for twenty-two species equations in addition to the five fluid dynamic variables.

2 Related Work

Recent work by a number of researchers has investigated GPGPU with impressive results in a variety of domains. Owens et. al. provide an excellent history of the GPU [6], chronicling its transformation in great detail. It is not uncommon to find researchers who achieve at least an order of magnitude improvement over reference implementations. GPUs have been used to accelerate a variety of application kernels, including more traditional operations like dense[7, 8, 9] and sparse[10] linear algebra as well as scatter-gather techniques[11]. The GPU has been successfully applied to a wide variety of fields including computational biophysics[12], molecular dynamics[13], and medical imaging[14, 15]. Our work takes a slightly higher level approach. While we do present performance measurements from an accelerated version of S3D, we examine the acceleration process as a whole, and endeavor to answer why certain applications perform so well on GPUs, while others fail to achieve significant performance improvements.

3 Identifying Candidates for Acceleration

3.1 Profiling

The first step in identifying a scientific application for acceleration is to identify the performance bottlenecks. The best case scenario involves a small number of computationally intense functions which comprise most of the runtime.

This is a fairly basic requirement and is a direct consequence of Amdahl's law. The CPU based profiling tool Tau identified S3D's getrates kernel as a major bottleneck[16]. This kernel involves calculating the rates of chemical reactions occurring in the simulation at each point in space. This computation comprises about half of the total runtime with the current chemistry model. As the chemical model becomes more complex, we anticipate that the getrates kernel will begin to comprise a stronger majority of total runtime. As the kernel's total percentage of runtime increases, the greater the potential for application speedup. Therefore, when choosing kernels to accelerate, the first to be examined should be the most time consuming.

3.2 Parallelism and Data Dependency

One of the main advantages of the GPU is the high number of processors, so it follows that kernels must exhibit a high degree of parallelism to be successful on a heterogenous GPU platform. While this can correspond to task-based parallelism, GPUs have primarily been used for data-parallel operations. This makes it difficult for GPUs to handle unstructured kernels, or those with intricate patterns of data dependency. Indeed, in situations with irregular control flow, individual threads can become serialized, which results in performance loss. Since the memory architecture of a GPU is dramatically different than most CPUs, memory access times can differ by several orders of magnitude based on access pattern and type of memory. For example, on the Tesla, an access to shared block memory is much faster than an access to global memory. Therefore, kernels must often be chosen based on memory access pattern, or restructured such that memory access is more uniform in nature. In S3D, the getrates kernel operates on a regular three dimensional mesh, so access patterns are fairly uniform, an easy case for the GPU.

The following psuedocode outlines the general structure of the sequential getrates kernel. The outer three loops can be computed in parallel, since points in the mesh are independent.

```
for x = 1 to length
   for y = 1 to length
      for z = 1 to length
         for n = 1 to nspecies
            grid[x][y][z][n] = F(grid[x][y][z][1:nspecies])
```

where length refers to the length of an edge of the cube, nspecies refers to the number of chemical species involved, and function F is an abstraction of the more complex chemical computations.

4 Kernel Acceleration

Once a suitable portion of the application has been identified, the acceleration process can begin. Parallel programming is inherently more difficult than

sequential programming, and developing high performance code for GPUs also incorporates complexity from architectural features. This "memory aware" programming environment grants the programmer control over low level memory movement, but demands meticulous data orchestration to maximize performance.

For S3D, the mapping between the getrates kernel and CUDA concepts is fairly simple. Since getrates operates on a regular, three-dimensional mesh, each point in the mesh is handled by a single thread. A block is composed of a local region of the mesh. Block size was chosen to be 256, based on the available number of registers per GPU core, in order to maximize occupancy.

During the development of the accelerated version of the getrates kernel, the memory access pattern was the most important factor for performance. When threads read or write memory in a highly parallel fashion, CUDA coalesces the memory access into a single operation, which has a dramatic and beneficial effect on performance. The optimized versions of the getrates kernel also use batched memory transfers and exploit block shared memory. This attention to detail pays off–accelerated versions of the getrates kernel exhibit promising speedups over the serial CPU version: up to 14.6x for the single precision version, and 9.3x for the double precision version for a single iteration of the kernel, see Figure 2. The serial CPU version was measured on an Intel Harpertown running at 2.5Ghz with 8GB of RAM.

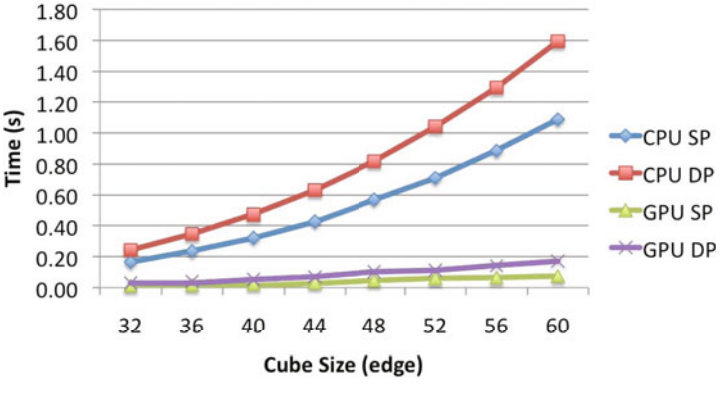

Fig. 2. Accelerated Kernel Results

5 Accuracy

While the evolution of the GPU has been remarkable, architectural remnants of its original, specialized function remain. Perhaps the most relevant of these to the scientific community is the bias towards single precision floating point computations. Single precision arithmetic was sufficient for the GPU's original tasks (rasterization, etc.). GPU benchmarking traditionally involved only these single precision computations, and performance demands have clearly shaped the GPU's allocation of hardware resources. Many GPUs are incapable of double

precision, and those that are typically pay a high performance cost. This cost generally arises from the differing number of floating point units, and it is almost always more than the performance difference between single and double precision on a traditional CPU. In S3D, the cost can clearly be seen in the performance difference in the single versus double precision versions of the getrates kernel.

From a performance standpoint, single precision computations are favorable compared to double precision, but the computations in scientific applications can be extremely sensitive to accuracy. Moreover, some double precision operations are not always equivalent on the CPU and GPU. GPUs may sacrifice fully IEEE compliant floating point operations for greater performance. For example, scientific applications frequently make extensive use of transcendental functions (sin, cos, etc.), and the Tesla's hardware intrinsics for these functions are faster, but less accurate than their CPU counterparts.

5.1 Accuracy in S3D

In S3D, the reaction rates calculated by the getrates kernel are integrated over time as the simulation progresses, and error from inaccurate reaction rates compounds and propagates to other simulation variables. While this is the first comparison of double and single precision versions of S3D, the issue of accuracy has been previously studied, and some upper bounds for error are known. S3D has an internal monitor for the estimated error from integration, and can take smaller timesteps in an effort to improve accuracy. Figure 3 shows the estimated error from integration versus simulation time. In this graph, the CPU and GPU DP versions quickly begin to agree, while the single precision version is much more erratic. In both double precision versions, the internal mechanism for timestep control succeeds in settling on a timestep of appropriate size. The single precision version has a much weaker guarantee on accuracy, and the monitor has a

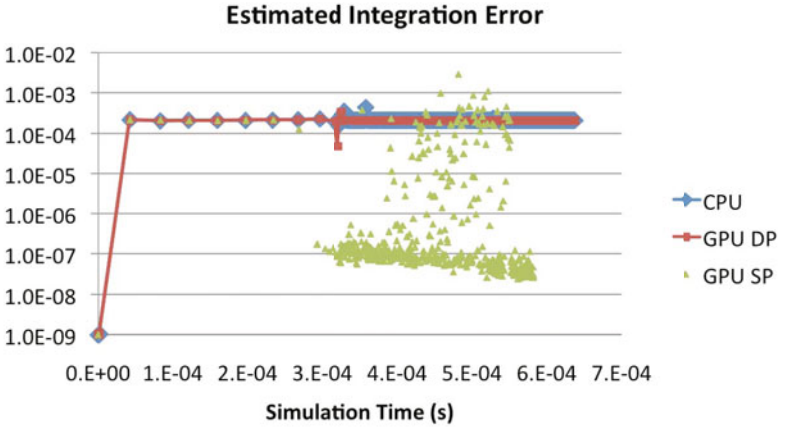

Fig. 3. Estimated Integrated Error. 1.00E-03 is the upper bound on acceptable error. The GPU DP and CPU versions completely overlap beginning roughly at time 4.00E-04.

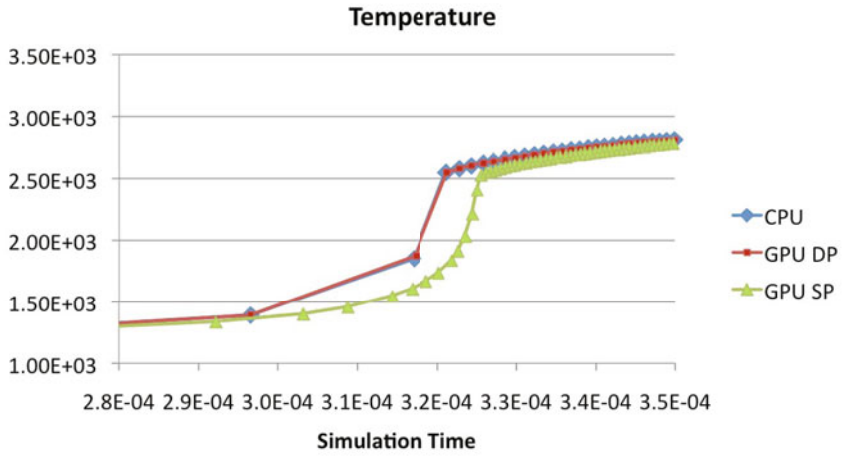

Fig. 4. Simulation temperature. Note the time gap of the increase in temperature at time roughly 3.00E-04. This corresponds to a delay in the prediction of ignition time.

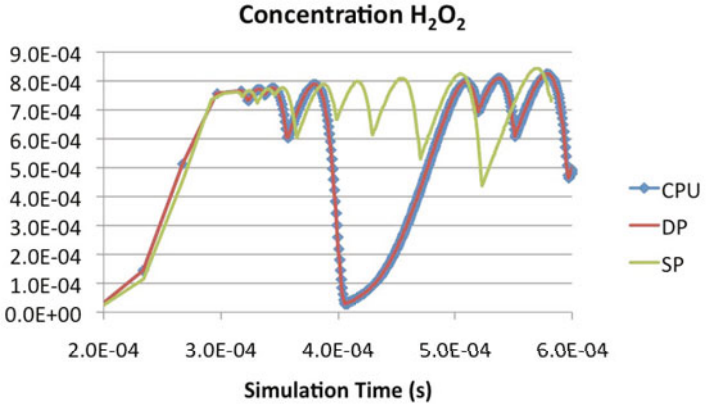

Fig. 5. Chemical Species H_2O_2. The CPU and GPU DP versions completely agree, while the GPU SP version significantly deviates, and fails to identify the dip at time 4.00E-O4.

difficult time controlling the timestep, oscillating between large timesteps with high error (sometimes beyond the acceptable bounds), and short timesteps with very low error. The increased number of timesteps required by the GPU single precision version will have consequences for performance, which will be explored in a later section.

The error from low precision can also be observed in simulation variables such as temperature (see Figure 4) or in chemical species, such as H_2O_2(see Figure 5). The current test essentially simulates a rapid ignition, and a relatively significant time gap can be seen between the rapid rise in temperature in the GPU single precision kernel versus the other versions. In the sensitive time scale

Table 1. Performance Results - Normalized cost is the average time it takes to simulate a single point in space for one nanosecond. S - Single Precision D - Double Precision.

Size	Kernel Speedup		% of	Amdahl's	Actual Speedup		Normalized Cost (ms)		
	S	D	Total	Limit	S	D	CPU	GPU DP	GPU SP
32	13.05x	8.17x	46.01%	1.82x	1.78x	1.61x	16.7	10.44	9.49
60	14.56x	9.32x	58.02%	2.38x	2.32x	2.27x	13.53	5.95	5.97

of ignition, this gap represents a serious error. In Figure 5, the error is much more pronounced, as the single precision version fails to predict the sudden decrease in H_2O_2 which occurs roughly at time 4.00E-04.

A similar trend can be observed throughout many different simulation variables in S3D. The CPU version tends to agree almost perfectly with the GPU double precision version, while the single precision version deviates substantially. Consequently, while the single precision version is much faster, it may be insufficient for sensitive simulations.

6 S3D Performance Results

In an ideal setting, the chosen kernel would strongly dominate the runtime of the application. However, in S3D, the getrates kernel comprises roughly half of the total runtime, with some variation based on problem size. Table 1 shows how speedup in the getrates kernel scales to whole-code performance improvements. Amdahl's limit is the theoretical upper bound on speedup, $s_\infty \approx \frac{1}{1-f_a}$, where f_a is the fraction of runtime that is accelerated.

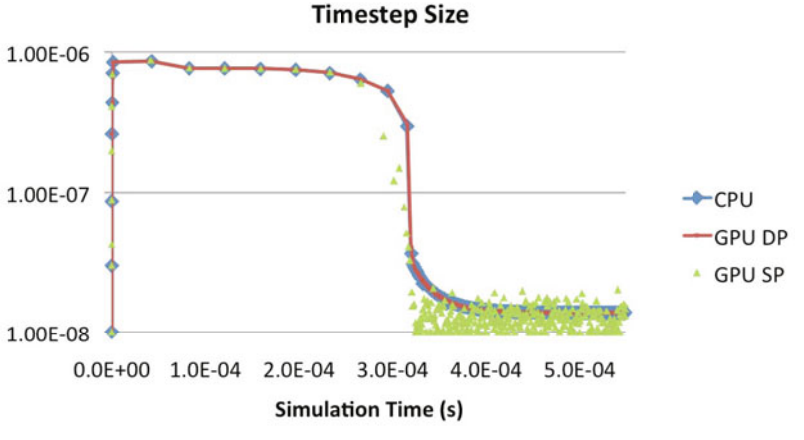

Fig. 6. Timestep Size – This graph shows the size of the timesteps taken as the rapid ignition simulation progressed. S3D reduces the timestep size when it detects integration inaccuracy. While the double precision versions take timesteps of roughly equivalent size, the single precision version quickly reduces timestep size in an attempt to preserve accuracy.

In S3D, there is a complex relationship between performance and accuracy. When inaccuracy is detected, timestep size is reduced in an attempt to decrease error, see Figure 6. Since single precision is less accurate, one can see erratic timestep sizes. This means that given the same number of timesteps, a highly accurate computation can simulate more time. In order to truly measure performance, it is important to normalize the wallclock time to account for this effect. In Table 1, normalized cost is the wallclock time it takes to simulate one nanosecond at one point in space. While the getrates kernel can be executed faster in single precision, the lack of accuracy causes the simulation to take very small timesteps. In some cases, the loss of accuracy in single precision calculations causes the total amount of simulated time to decrease, potentially eliminating any performance benefits.

7 Conclusions

Graphics processors are rapidly emerging as a viable platform for high performance scientific computing. Improvements in the programming environments and libraries for these devices are making them an appealing, cost-effective way to increase application performance. While the popularity of these devices has surged, GPUs may not be appropriate for all applications. They offer the greatest benefit to applications with well structured, data-parallel kernels. Our study has described the strengths of GPUs, and provided insights from our experience in accelerating S3D. We have also examined one of the most important aspect of GPUs for the scientific community, accuracy. The differences in accuracy between GPU and IEEE arithmetic resulted in drastic consequences for correctness in S3D. Despite this relative weakness, the heterogeneous GPU version of the kernel still manages to outperform the more traditional CPU version and produce high quality results in a real scientific application.

References

[1] NVIDIA: CUDA programming guide 2.0 downloaded (December 1, 2008), `www.nvidia.com/object/cudadevelop.html`

[2] Hawkes, E.R., Sankaran, R., Sutherland, J.C., Chen, J.H.: Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. Journal of Physics: Conference Series 16, 65–79 (2005)

[3] Sutherland, J.C.: Evaluation of mixing and reaction models for large-eddy simulation of nonpremixed combustion using direct numerical simulation. Dept. of Chemical and Fuels Engineering, PhD, University of Utah (2004)

[4] Yu, W., Vetter, J., Oral, H.: Performance characterization and optimization of parallel I/O on the Cray XT. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–11 (April 2008)

[5] Mellor-Crummey, J.: Harnessing the power of emerging petascale platforms. Journal of Physics: Conference Series 78(1), 12–48 (2007)

[6] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008)

[7] Barrachina, S., Castillo, M., Igual, F., Mayo, R.: Evaluation and tuning of the level 3 CUBLAS for graphics processors. In: Proceedings of the IEEE Symposium on Parallel and Distributed Processing (IPDPS), April 2008, pp. 1–8 (2008)
[8] Fujimoto, N.: Faster matrix-vector multiplication on GeForce 8800GTX. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, April 2008, pp. 1–8 (2008)
[9] Cummins, G., Adams, R., Newell, T.: Scientific computation through a GPU. In: Southeastcon, pp. 244–246. IEEE, Los Alamitos (April 2008)
[10] Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: SIGGRAPH'03: ACM SIGGRAPH 2003 Papers, pp. 917–924. ACM, New York (2003)
[11] He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient gather and scatter operations on graphics processors. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pp. 1–12. ACM, New York (2007)
[12] Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K.: Accelerating molecular modeling applications with graphics processors. Journal of Computational Chemistry 28, 2618–2640 (2005)
[13] Rodrigues, C.I., Hardy, D.J., Stone, J.E., Schulten, K., Hwu, W.M.W.: GPU acceleration of cutoff pair potentials for molecular modeling applications. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, pp. 273–282. ACM, New York (2008)
[14] Kruger, J., Westermann, R.: Acceleration techniques for GPU-based volume rendering. In: Visualization, VIS 2003, October 2003, pp. 287–292. IEEE, Los Alamitos (2003)
[15] Mueller, K., Xu, F.: Practical considerations for GPU-accelerated CT. In: 3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, April 2006, pp. 1184–1187 (2006)
[16] Shende, S., Malony, A.D., Cuny, J., Beckman, P., Karmesin, S., Lindlan, K.: Portable profiling and tracing for parallel, scientific applications using C++. In: SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, pp. 134–145. ACM, New York (1998)