Accelerating reactive-flow simulations using graphics processing units

Kyle E. Niemeyer*
Case Western Reserve University, Cleveland, OH, 44106, USA
Chih-Jen Sung†
University of Connecticut, Storrs, CT, 06269, USA

The progress made in accelerating simulations of fluid flow using GPUs is surveyed. The review first provides an introduction to GPU computing and programming. A case study of simulating incompressible flow in a lid-driven cavity is performed, comparing the performance of CPU- and GPU-based solvers to demonstrate the potential improvement even with simple codes. Recent efforts to accelerate CFD simulations using GPUs are reviewed for reactive flow solvers. Finally, recommendations for implementing CFD codes on GPUs are given and remaining challenges are discussed, such as the need to develop new strategies and redesign algorithms to enable GPU acceleration.

I. Introduction

The computational demands of scientific computing are increasing at an ever-faster pace as scientists and engineers attempt to model increasingly complex systems and phenomena. This trend is equally true in the case of fluid flow simulations, where accurate models require large numbers of grid points and therefore large, expensive computer systems. In particular, high Reynolds numbers—the domain of many real-world flows—require even-denser computational grids to resolve the scales of importance or even turbulent flows. Modeling reactive flows, with realistic/detailed chemical models involving large numbers of species and reactions, can increase the computational demands further by another order of magnitude. Due to the stringent hardware requirements warranted by such computational demands, high-fidelity simulations are typically not accessible to most industrial or academic researchers and designers.

In the past, we could rely on computational capabilities improving with time, simply waiting for the next generation of central processing units (CPUs) to enable previously inaccessible calculations. However, in recent years, the pace of increasing processor speeds has slowed, largely due to limitations of power consumption and heat dissipation preventing further decreases in transistor size. Power consumption, and therefore heat output, scales with processor area, while processor speed scales with the square root of area. Adding cores allows processors to increase overall performance via parallelism while minimizing larger power consumption by avoiding significant increases in overall area. In order to keep up with Moore's Law—processor speeds doubling roughly every year and a half—processor manufacturers are embracing parallelism. Top-of-the-line CPUs used in personal computers and supercomputing clusters contain four to eight cores. Graphics processing units (GPUs), on the other hand, consist of many hundreds to thousands of—albeit fairly simple—processing cores, and fall in the category of "many-core" processors. This level of parallelism matches that of large clusters of CPUs.

Regarding the need for parallel computing, it would depend on whether the types of problems can be parallelized or whether they are fundamentally serial calculations. The key to parallel problems (sometimes termed "embarrassingly parallel") is data independence: multiple tasks must be able to operate on data independently. Examples of embarrassingly parallel problems include matrix multiplication, Monte Carlo simulations, calculating cell fluxes across space in the finite volume approach, and calculating finite differences

*Ph.D. Candidate, Department of Mechanical and Aerospace Engineering, AIAA Student Member.
†Professor, Department of Mechanical Engineering, AIAA Associate Fellow.

across a grid. In contrast, calculating the Fibonacci sequence is inherently serial, as each term relies on the previous two. In general, iterative solution methods are serial, although the internal calculations of each iteration might be parallelizable.

As researchers identify computing problems with appropriate data parallelism, GPUs are becoming popular in many scientific computing areas such as molecular dynamics,[1–5] protein folding,[6] quantum chemistry,[7–9] computational finance,[10–12] data mining,[13] and a variety of computational medical techniques such as computed tomography (CT) scan processing,[14–16] white blood cell detection and tracking,[17] cardiac simulation,[18] and radiation therapy dose calculation.[19] In 2007 and 2008, Owens et al.[20,21] surveyed the use of GPUs in general purpose computations, but as the popularity of GPU acceleration exploded in recent years many more areas are under investigation. Most efforts in moving existing applications to GPUs demonstrated around an order of magnitude (or more, in some cases) improvement in performance.

This survey is structured as follows. First, GPU computing is introduced and we discuss some topics related to programming applications running on graphics processors. Second, we present a case study relevant to CFD: a finite-difference incompressible Navier–Stokes solver for lid-driven cavity flow. We use this relatively simple example to demonstrate the potential performance improvement of GPU codes relative to CPU versions. Next, we review efforts to both modify existing and create new CFD solvers for GPU acceleration, focusing on chemically reactive flow solvers. Finally, we discuss the progress made thus far to exploit GPUs for CFD simulations as well as remaining challenges, and make some recommendations.

Note that while here we focus on GPUs, the approaches developed can apply to other newly developed many-core processing architectures in general. In fact, in the case of the OpenCL programming language,[22] the same programs written for GPUs now should be useable on whatever many-core processing standard is adopted—it is designed for executing programs on heterogeneous computing platforms. There is a trend towards massively parallel processors, and GPUs are an early entry in this category.

II. GPU computing

As their name suggests, GPUs were initially developed for graphics/video processing and display purposes, and programmed with specialized graphics languages. In these applications, where many thousands or millions of pixels need to be displayed on the screen simultaneously, throughput is more important than latency. In contrast, the CPU executes a single instruction (or few instructions, in the case of multiple cores) rapidly. This led to the current highly parallel architecture of modern graphics processors. The explosive growth of GPU processing capabilities as well as diminishing costs in recent years have been propelled mainly by the video game industry's demands for fast, high-quality processing (both for on-screen images and physics engines), and these trends will likely continue driven by commercial demand. Figure 1 demonstrates this recent growth in performance, comparing the maximum floating-point operations per second (FLOPS) of modern multicore CPUs and GPUs; the newest GPUs offer more than an order of magnitude higher performance, although no data was available for the most recent CPU models.

II.A. Programming GPUs

The current generation of application programming interfaces (APIs), such as CUDA[23] and OpenCL,[22] enables a C-like programming experience while exposing the massively parallel architecture of graphics processors. This avoids programming in the graphics pipeline directly. We will focus our discussion on CUDA, a programming platform created and supported by NVIDIA, but the programming model of OpenCL, an open-source framework supported by multiple vendors, is similar so the same concepts apply (albeit with slightly different names for equivalent features). This section is not intended to function as a complete reference for programming CUDA applications, but only to give a brief overview of the CUDA paradigm. Interested readers should see the textbooks, e.g., by Kirk and Hwu[24] and Sanders and Kandrot.[25]

In CUDA, a parallel function is known as a "kernel," which consists of many threads that perform tasks concurrently. Functions intended for operation on the GPU (the device) and the CPU (the host) are preceded with `__device__` and `__host__`, respectively. Kernel functions are indicated with `__global__`. Threads are organized into three-dimensional blocks, which in turn are organized into a two-dimensional grid.[a] All threads in a grid execute the same kernel function. The specific location (coordinate) of a thread inside the hierarchy of blocks and grids can be accessed using the variables `threadIdx` and `blockIdx`; the

[a]Recent GPU hardware allows a three-dimensional grid.

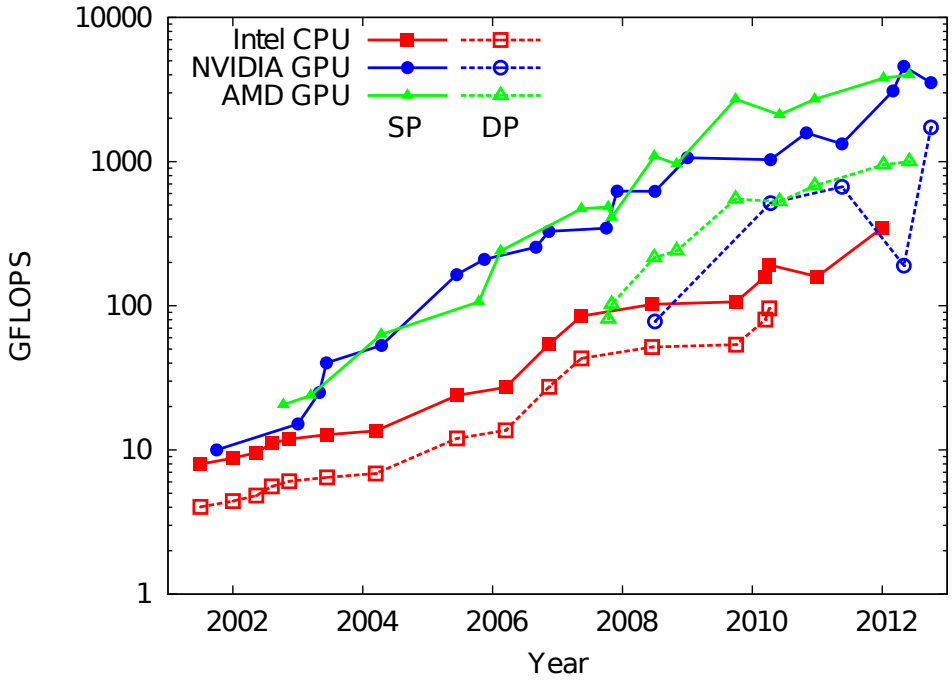

Figure 1: Performance comparison of Intel CPUs and NVIDIA and AMD GPUs, measured in GFLOPS (gigaFLOPS, i.e., billion floating-point operations per second), over the past decade. "SP" and "DP" refer to single and double precision, respectively. Note that the performance is presented in logarithmic scale. Source: John Owens personal communication and vendor specifications.

dimensions of the block (i.e., the number of threads) and grid (i.e., the number of blocks) can be retrieved using `blockDim` and `gridDim`, respectively.

Another avenue for accelerating applications using GPUs is OpenACC,[26,27] which uses compiler directives (e.g., `#pragma`) placed in Fortran, C, and C++ codes to identify sections of code to be run in parallel on GPUs. This approach is similar to OpenMP[28–30] for parallelizing work across multiple CPUs or CPU cores that share memory. OpenACC is an open standard being jointly developed by NVIDIA, Cray, the Portland Group, and CAPS. Since OpenACC is relatively new and immature, only a few groups have used OpenACC thus far to accelerate their applications. Wienke et al.[31] found that OpenACC achieved 80% of the performance of OpenCL in simulations of bevel gear cutting, but only 40% in solving the neuromagnetic inverse problem in the neuroimaging technique magnetoencephalography (reconstructing focal activity in the brain). Reyes et al.[32] showed a similar range of performance, comparing OpenACC with CUDA implementations of LU decomposition, a thermal simulation tool, and a nonlinear global optimization algorithm for DNA sequence alignments. Recently, Levesque et al.[33] reported on their experience hybridizing Sandia National Laboratory's massively parallel direct numerical simulation code S3D from MPI-only to MPI/OpenMP/OpenACC for three levels of parallelism; we will discuss their results in greater detail in Section V.

The main benefit of the OpenACC approach (as well as OpenMP) is that programs may be accelerated without modifying the underlying source code—a non-OpenACC-enabled compiler would treat the directives as comments. Fortran code is handled similarly, albeit with a different directive indicator syntax (`C$OMP` or `!$omp` rather than `#pragma omp`). This contrasts greatly with porting applications written for the CPU to either CUDA or OpenCL, which must be completely rewritten. This convenience comes at the cost of slightly degraded performance, but OpenACC allows researchers to accelerate existing code in a matter of hours, rather than days or weeks. We will compare the performance of OpenACC implementations of our case study in Section III.

III. Case study: lid-driven cavity flow

A case study, relevant to CFD applications, was performed to demonstrate the potential acceleration of CFD applications using graphics processors: solution of the two-dimensional, laminar incompressible Navier–Stokes equations based on the finite difference method, using the solution procedure given in the textbook by Griebel et al.[34] Four versions of the solver were compared: single-core CPU, six-core CPU using OpenMP, native GPU using CUDA, and GPU-accelerated using OpenACC. The GPU performance experiments were performed using an NVIDIA Tesla c2075 GPU with 6 GB of global memory. An Intel Xeon X5650 CPU, running at 2.67 GHz with 256 KB of L2 cache memory per core and 12 MB of L3 cache memory, served as the host processor for the GPU calculations and ran the single-core CPU and OpenMP calculations.

We used the GNU Compiler Collection (gcc) version 4.6.2 (with the compiler options "`-O3 -ffast-math -std=c99 -m64`") to compile the CPU programs, the PGI Compiler toolkit version 12.9 to compile the OpenMP ("`-fast -mp`") and OpenACC ("`-acc -ta=nvidia,cuda4.2,cc20 -lpgacc`") versions, and the CUDA 5.0 compiler nvcc version 0.2.1221 ("`-O3 -arch=sm_20 -m64`") to compile the GPU version. The functions `cudaSetDevice()` and `acc_init()` were used to hide any device initialization delay in the CUDA and OpenACC implementations, respectively.

III..1. Methodology

The domain was discretized with a uniform, staggered grid (i.e., the pressure values are located at the centers of grid cells while velocity values are located along the edges). Briefly, the discretized momentum (assuming no gravity/body-force terms) and pressure-Poisson equations are:

$$F_{i,j}^{(n)} = u_{i,j}^{(n)} + \delta t \left(\frac{1}{\mathrm{Re}} \left(\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j}^{(n)} + \left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j}^{(n)} \right) - \left[\frac{\partial (u^2)}{\partial x}\right]_{i,j}^{(n)} - \left[\frac{\partial (uv)}{\partial y}\right]_{i,j}^{(n)} \right), \quad i = 1, \ldots, i_{\max} - 1, \quad j = 1, \ldots, j_{\max}, \tag{1}$$

$$G_{i,j}^{(n)} = v_{i,j}^{(n)} + \delta t \left(\frac{1}{\mathrm{Re}} \left(\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j}^{(n)} + \left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j}^{(n)} \right) - \left[\frac{\partial (uv)}{\partial x}\right]_{i,j}^{(n)} - \left[\frac{\partial (v^2)}{\partial y}\right]_{i,j}^{(n)} \right), \quad i = 1, \ldots, i_{\max}, \quad j = 1, \ldots, j_{\max} - 1, \tag{2}$$

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = \frac{1}{\delta t} \left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y} \right), \quad i = 1, \ldots, i_{\max}, \quad j = 1, \ldots, j_{\max}, \tag{3}$$

$$u_{i,j}^{(n+1)} = F_{i,j}^{(n+1)} - \frac{\delta t}{\delta x} \left(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)} \right), \quad i = 1, \ldots, i_{\max} - 1, \quad j = 1, \ldots, j_{\max}, \tag{4}$$

$$v_{i,j}^{(n+1)} = G_{i,j}^{(n+1)} - \frac{\delta t}{\delta y} \left(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)} \right), \quad i = 1, \ldots, i_{\max}, \quad j = 1, \ldots, j_{\max} - 1, \tag{5}$$

where i and j indicate the x and y cell coordinates, respectively, (n) indicates the time step (i.e., corresponding to time t_n), u and u are the velocity components in the x and y directions, respectively, p is the pressure, Re is the Reynolds number, and δt is the time-step size. The second derivatives in Eqs. (1) and (2) were treated with central differences, while the first derivatives were treated with a mixture of central differences and the donor-cell discretization (see Griebel et al.[34] for details).

In this example, we used the above procedure to solve the case of lid-driven flow in a square cavity, corresponding to no-slip boundary conditions on the vertical sides and bottom and a unit horizontal velocity along the top. Boundary conditions were treated numerically as described by Griebel et al.[34] We used

boundary cells to avoid conditional statements and the associated potential thread divergence. Kernel functions evaluated the boundary conditions for both velocity and pressure values.

We solved Eq. (3) iteratively using the red-black Gauss–Seidel method with successive over-relaxation (SOR). Traditional Gauss–Seidel or SOR approaches are not suitable for use on parallel CPU or GPU systems since the order of operations is neither known nor controllable, and conflicts in accessing and writing to memory may occur (although Jacobi iteration is suitable for parallel/GPU implementation, since calculation of new values depends only on old values). Red-black SOR solves this problem by coloring the grid like a checkerboard, alternating red and black cells. First, the algorithm updates values at red cells—which depend only on black cells—then black cells—which depend only on red cells. Both of these operations can be performed in parallel. Red-black SOR was first used to solve a system of linear equations on vector and parallel computer systems by Adams and Ortega,[35] although introduced earlier (e.g., by Young[36]). Liu et al.[37] provide a more detailed analysis of red-black SOR implemented on GPUs.

The GPU code contains 11 kernel functions: one to set the velocity boundary conditions, two corresponding to Eqs. (1) and (2), one to calculate the L^2-norm of the pressure (for a relative SOR tolerance), two to set the horizontal and vertical pressure boundary conditions, two for the red and black portions of the SOR algorithm solving Eq. (3), one to calculate the pressure residual for each SOR iteration, and two corresponding to Eqs. (4) and (5) (which also return the maximum u- and v-velocities). All memory remains on the GPU during the simulation (e.g., arrays holding F_{ij}, p_{black}, u_{ij}), except for that needed to evaluate the stopping criterion for the SOR iteration and the maximum velocities to calculate the time-step size based on stability criteria (see Griebel et al.[34]). We used shared memory for these global reduction operations such that only one value per block needed to be transferred back to the CPU. Performance timing included all these memory transfers, including those needed to initialize all variables on the GPU at the beginning and return the pressure and velocity values at the end of the simulation.

Our OpenACC solver was based on the CPU solver, with directives instructing the compiler to use the GPU on loops matching the kernel functions of the GPU solver. The OpenMP solver was created the same way, using the appropriate compiler directives. No specific optimization instructions were given to either the OpenMP or OpenACC solvers; rather, we allowed the compiler to manage this automatically.

In order to study the performance of the GPU and OpenACC solvers against the CPU versions, we varied the mesh size from 64^2 to 2048^2. The source code was written in standard and CUDA C for the CPU and GPU versions, respectively, with compiler directives added to the CPU version to create OpenMP and OpenACC versions.[b] In the native, CUDA-based GPU version, we kept a constant block size of 128×1 except for the cases of the mesh size being 64^2 or 128^2, where we used half the mesh size in one direction (e.g., 32×1 for a mesh size of 64^2). We used grid configurations of $N/2B \times N$ for the pressure solver and $N/B \times N$ for the velocity portions, where N is the number of mesh cells in one direction (e.g., 512) and B is the block size (e.g., 128). For each mesh size, we performed a single time step integration—the initial time step. Due to the stability criteria for selecting the time-step size, the finer mesh sizes required increasingly smaller time-step sizes. All calculations were performed in double precision.

III..2. Results

Figure 2 shows the performance comparison of the solvers over a wide range of mesh sizes, for a single time step. At mesh sizes smaller than 128^2, the single- and six-core CPU solvers run faster than either GPU solver, but with increasing mesh size the native, optimized GPU solver becomes 8.1 and 2.8 times faster than the single- and six-core CPU solvers, respectively, at a mesh size of 2048^2.

With increasing problem size, the OpenACC solver approaches the performance of the native GPU version, running 1.3 times slower than the native GPU code at the largest problem size. At smaller problem sizes, it runs nearly four times slower than the native GPU version. It is clear, however, that OpenACC offers nearly the same performance as native GPU code at large problem sizes. On the other hand, for smaller problem sizes, neither GPU approach offers better performance than CPU-based codes.

Note that we did not optimize the block size for the GPU solver, but left it constant for this simple demonstration. Similarly, we allowed the OpenACC compiler to determine the optimal configuration, rather than manually adjust its equivalents ("gang" and "vector" sizes). More in-depth optimization could further improve performance for both implementations.

[b]The full source code is available online: http://github.com/kyleniemeyer/lid-driven-cavity_gpu

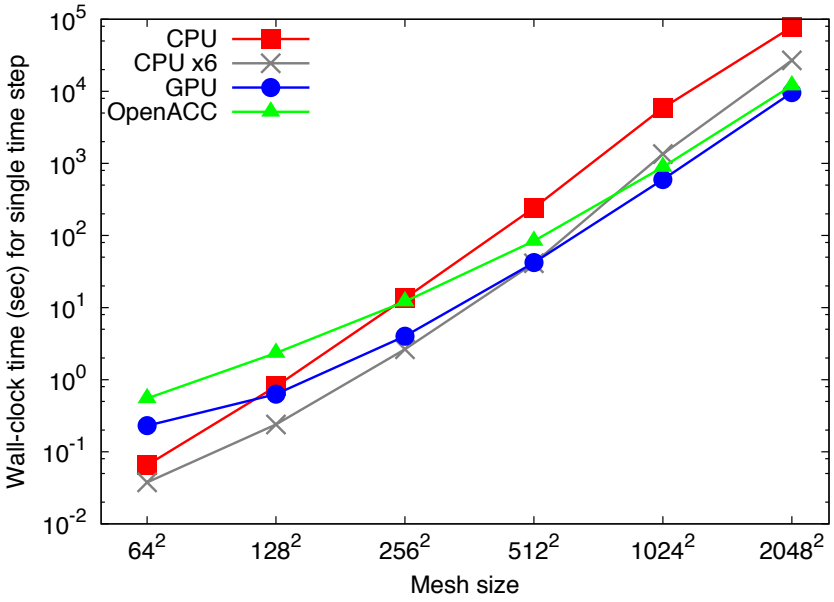

Figure 2: Performance comparison of a single time step of the lid-driven cavity problem using the CPU solver on one core and six cores (via OpenMP), the fully optimized CUDA-based GPU solver, and OpenACC-based GPU solver, for a wide range of mesh sizes.

Finally, we observe that while the native GPU solver shows the best performance for larger grid sizes, the OpenACC version performs nearly as well, especially for double-precision calculations. These results suggest that OpenACC is a good alternative to writing native GPU code, especially considering the fact that porting CPU applications to CUDA can require days to weeks of work while adding the OpenACC compiler directives takes only hours or even minutes. However, OpenACC support is currently limited, and can only be applied to applications that already favor parallelization—such as those based on loops with independent iterations—and where functions may be inlined. With this in mind, OpenACC is a good choice to quickly accelerate existing code and determine potential speedup, while writing native GPU applications offers the highest potential performance if fully optimized. In either case, algorithms may need to be redesigned in order to support massive parallelization, although this was not necessary in the current example.

IV. GPUs in computational fluid dynamics

The heavy computational demands of high-fidelity fluids simulations typically prevent industrial or academic researchers from performing and using such studies. CFD applications in particular stand to benefit from GPU acceleration due to the inherent data parallelism of calculations for both finite difference and finite volume methods, and as GPU hardware and software matured more researchers exploited the application of GPU computing in their respective areas of interest. While Vanka et al.[38] reviewed some of the literature on using GPUs for CFD applications, we will attempt to provide a survey of recent advances for reactive-flow simulations.

Bolz et al.,[39] Harris and coworkers,[40,41] and Krüger and Westermann[42] first implemented real-time, physics-based fluids simulation on GPUs using the stable fluids approach of Stam[43] for solving the incompressible Navier–Stokes equations. Liu et al.[44] used the same approach to solve flow around complex boundaries. In these approaches, the finite difference equations were parallelized such that the discretized derivatives for each grid point were solved concurrently. In other words, instead of looping over each point in a serial fashion, the derivatives were solved simultaneously. However, the stable fluids approach includes excessive numerical dissipation, such that the approach is limited to physics-based animations rather than accurate, numerical simulations.

Following these initial efforts, researchers began to develop more accurate Navier–Stokes solvers for GPUs. Earlier efforts in particular focused on solvers capable of simulating laminar flows.[45–60] Most efforts relied on

structured grids, limiting the complexity of potential objects in the domain; Corrigan et al.[59] developed the first GPU-accelerated solver for unstructured grids, capable of simulating three-dimensional compressible, inviscid flows.

Most real-world fluid flow problems, and in particular those relevant to engineering applications such as flow through turbomachinery and engines, and over cars and aircraft, are turbulent. However, due to the difficulty in transferring CFD codes to operation on graphics processors, only recently have researchers begun to accelerate turbulent flows. Ironically, this is one of the areas that need acceleration the most due to the added expense of accurate turbulence models. Phillips et al.[61] developed one of the first GPU solvers capable of simulating turbulence, extending on the group's previous work porting portions of the existing MBFLO solver to the GPU.[52] Following this initial effort and continuing to the present, researchers developed GPU-accelerated solvers capable of predicting turbulent flow. Approaches include those based on Reynolds-averaged Navier–Stokes (RANS),[62–66] direct numerical simulation (DNS),[67–69] and large-eddy simulation (LES).[70–72] As with laminar flow solvers, most efforts relied on structured grids; Kampolis et al.[63] and Asouti et al.[64] developed GPU-accelerated unstructured grid-based turbulent flow solvers, both based on the vertex-centered finite difference approach.

V. GPU acceleration of reactive-flow solvers

In order to design the next-generation of engines and combustors, accurate and efficient simulations of reactive flow are vital. Traditionally, chemistry was represented in a simple manner, using one- or multiple-step global reaction mechanisms to capture the overall fuel oxidation and heat release. Unfortunately, the inability to capture pressure dependence, vital in high compression-ratio internal combustion engines, is one of the fundamental issues with such global mechanisms. In addition, as emissions regulations become more and more stringent, simulations must be able to predict concentrations of pollutant species and soot precursors for the development of advanced high-efficiency, low-emissions combustors.

Including detailed chemistry in simulations of reactive flow induces greater computational expense for two reasons: (1) chemical stiffness, caused by rapidly depleting species and/or fast reversible reactions, requires specialized integration algorithms (traditionally, high-order implicit solvers based on backward differentiation formulae); and (2) the large and ever-increasing size of detailed reaction mechanisms for transportation fuels of interest. While mechanisms for fuels relevant to hypersonic engines, such as hydrogen or ethylene, may contain 10–70 species,[73,74] a recent surrogate mechanism for gasoline consists of about 1550 species and 6000 reactions;[75] a surrogate mechanism for biodiesel contains almost 3300 species and over 10000 reactions.[76] Incorporating such large, realistic reaction mechanisms in reactive flow simulations is beyond the scope of this survey; Lu and Law[77] recently reviewed the subject. In brief, mechanism reduction is typically performed on large mechanisms to decrease the size from hundreds or thousands of species (and many more reactions) to a manageable size (e.g., < 100 species); unfortunately, high-fidelity simulations using reaction mechanisms of this size are still too demanding to be performed on average computer systems. By utilizing the massively parallel architecture of relatively low-cost GPUs, previously inaccessible reactive-flow simulations may be performed on such systems. Though GPUs offer the potential of significant performance enhancement, researchers are only beginning to explore GPU acceleration of reactive-flow solvers.

Spafford et al.[78] made the first foray into this area by porting the species rate evaluation of Sandia National Laboratory's massively parallel DNS code S3D to the GPU. S3D is capable of solving the fully compressible Navier–Stokes equations combined with detailed chemistry.[79,80] On the CPU, the time integration of the chemical source terms was performed with an explicit fourth-order Runge–Kutta method. Each integration step therefore requires four species rate evaluations and consequently four GPU kernel invocations and the associated memory transfer. In addition to calculating the rates for all grid points in parallel, the evaluation of reaction rates was accelerated by the GPU's hardware-accelerated transcendental function calls. Using an ethylene reaction mechanism with 22 species, they achieved a performance speedup of around 15 and 9 for single and double precision calculations, respectively.

In a different approach, Shi et al.[81] used the GPU to simultaneously calculate all the reaction rates for a single kinetic system (i.e., a single computational volume/grid point) and accelerate the matrix inversion step of the implicit time integration using a commercial GPU linear algebra library, CULA.[82] Initially, this was demonstrated using homogeneous autoignition simulations, which are essentially the integration of a system of ordinary differential equations (ODEs) for temperature and the species rates. Overall, Shi et al.[81] demonstrated a speedup of up to 22 for large chemical reaction mechanisms (> 1000 species), but for

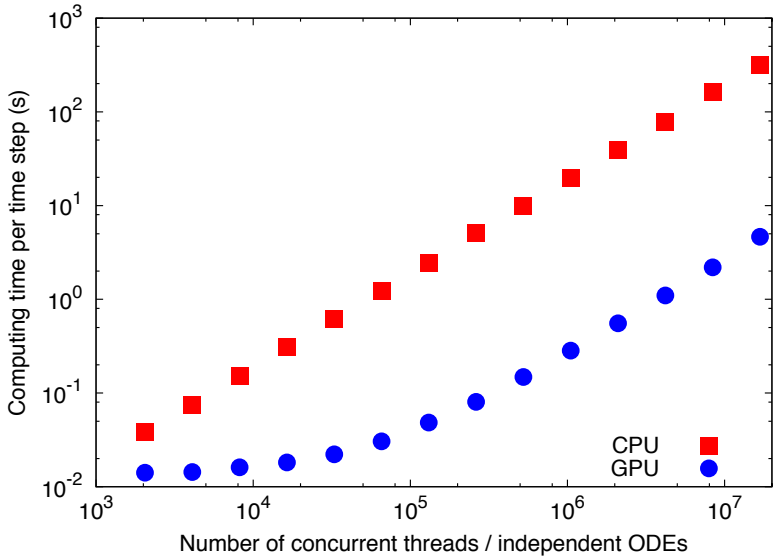

Figure 3: Performance comparison of CPU and GPU explicit integration of a nonstiff hydrogen mechanism. Note that both axes are displayed in logarithmic scale.

moderate-size mechanisms (< 100 species) the GPU version performed similarly or even slower than the CPU-only version.

Shi et al.[83] also implemented their GPU-based chemistry solver, paired with a dynamic reaction mechanism reduction algorithm, into the existing engine simulation code KIVA-Chemkin. Unlike the approach of Spafford et al.,[78] the species rates for each grid cells were evaluated in serial across the grid. In this case, acknowledging the poor performance of the GPU-based ODE solver on smaller mechanisms, the CPU was used to perform the species integration when the mechanism size dropped below 300 species due to the dynamic reduction. The GPU-based solver was only used when the mechanism was larger than 300 species. Using a large detailed mechanism for methyl decanoate (2877 species and 8555 reactions) to represent biodiesel,[76,84] Shi et al.[83] simulated a two-dimensional homogeneous-charge compression-ignition engine. Without the dynamic mechanism reduction, the GPU version performed nearly 20 times faster than the CPU-only version. With the dynamic reduction, the mixed CPU/GPU version only performed about twice as fast as the CPU-only version. Therefore, if dynamic reduction is used, combining the GPU acceleration with this approach may not be needed since the mechanism size is sufficiently small during most of the simulation.

Building upon the work of Spafford et al.,[78] Niemeyer et al.[85] developed a GPU-based explicit integration algorithm for nonstiff chemistry, where the species rate equations for all grid cells were integrated concurrently. In order to minimize the memory transfer between the CPU and GPU, the entire time integration step of a fourth-order Runge–Kutta method was performed on the GPU. Contrast this with the approach of Spafford et al.,[78] where communication must occur before and after each of the four species rate calculations. Using a compact hydrogen mechanism with 9 species and 38 irreversible reactions,[86] Niemeyer et al.[85] demonstrated a computational speedup of up to 75 compared to a single-core CPU over a wide range of independent spatial locations (i.e., cell volumes/grid points), as shown in Fig. 3. The performance of the GPU over the CPU implementation improved with increasing number of computational cells, so this approach would be beneficial for large-scale simulations. Unlike the GPU-accelerated CFD approaches discussed above, the chemistry integration of the strategy of Niemeyer et al.[85] would be performed on the GPU simultaneously with CPU transport calculations, so no portion of the calculation waits for the other to finish. In fact, the computational time required for chemistry is reduced enough to potentially allow additional calculations (e.g., transport terms) to be performed on the GPU, further improving performance.

Shi et al.[87] presented a hybrid CPU/GPU chemistry integration strategy where, similar to the formulation of Niemeyer et al.,[85] the GPU was used to explicitly integrate nonstiff chemistry in many grid cells (concurrently). In their work, though, spatial locations with stiff chemistry were handled with a standard implicit integrator on the CPU. This combined approach achieved a performance speedup of 11–46. How-

ever, unlike the strategy of Niemeyer et al.,[85] CPU transport calculations by Shi et al.[87] were performed in serial with the CPU stiff chemistry integrations, limiting the potential performance. Further, the lack of any stiffness removal would hinder the implicit integration.

Le et al.[88] developed the first reactive flow solver where both the fluid transport and chemical kinetics terms are evaluated on the GPU. As with most other approaches, they used operator splitting to decouple and independently solve the fluid transport and chemistry terms. The transport terms were solved using two approaches: (1) a fifth-order monotonicity-preserving (MP5) discretization paired with third-order Runge–Kutta time integration, and (2) arbitrary derivative Riemann solver using weighted essentially non-oscillatory (ADERWENO) reconstruction, which doesn't require time-stepping and therefore involves less overhead. The stiff chemical kinetics terms were solved using a first-order implicit method (the implicit/backward Euler method), employing a direct Gaussian elimination to solve the resulting linear system of equations. To implement their algorithms on the GPU, Le et al.[88] employed the same thread-per-cell approach as Spafford et al.,[78] Niemeyer et al.,[85] and Shi et al.[87] In the chemical kinetics kernel, Le et al.[88] attempted to use shared memory to store the system of variables (e.g. temperature, pressure, species mass fractions) as well as the Jacobian matrix—necessary for the Euler time integration. Due to the limited size of shared memory, only two rows of the Jacobian matrix were stored at a time, exploiting the row-by-row sequence of Gaussian elimination. Unfortunately, this resulted in many memory transfers between the global and shared memories, resulting in low performance for the reaction mechanisms employed in this study; using coalesced global memory alone offered better performance. Compared against an equivalent CPU version executed on a single processor core, their combined GPU solver performed up to 40 times faster using a reaction mechanism for methane with 36 species and 308 reactions, on a grid with greater than 10^4 cells. However, the low accuracy of the chemistry solver—first order—may limit the usefulness of this approach to high-fidelity simulations.

Recently, Levesque et al.[33] presented their experiences hybridizing S3D from one level of parallelism using MPI to three levels, combining distributed memory CPUs across multiple cluster nodes, multi-core CPUs and CPUs sharing memory on the same node, and GPUs. Rather than developing native GPU code using CUDA or OpenCL, they used OpenACC paired with MPI and OpenMP, so that the resulting code was agnostic to architecture. To our knowledge, this was the first production CFD code ported to the GPU using OpenACC. Rather than the early work accelerating S3D by Spafford et al.[78] where the GPUs only evaluated species rates, Levesque et al.[33] moved all calculations to the GPUs, using the host CPUs to perform communication and boundary treatments.

In order to better parallelize the code on both OpenMP and OpenACC, Levesque et al.[33] made a number of changes to the structure of the code—although the actual algorithms and computations remained the same. For example, the source term calculations—which are mostly point-wise, requiring only local information—were separated from derivative computations and moved into the same loop, maximizing spatial granularity and allowing overlapping of computation and communication. In another example, they developed a novel derivative queuing system where the derivative computation was split into a number of tasks that could be overlapped with other calculations and communication. A number of other restructuring changes were made to reduce memory operations and vectorize loops. In fact, optimizations targeted at GPU acceleration also resulted in improved CPU-only performance with OpenMP. The MPI+OpenMP version, without any GPU acceleration, ran up to around 1.4 times faster than the MPI-only version—using the same number of CPU cores. Levesque et al.[33] then compared the performance of the OpenMP and OpenMP+OpenACC codes on a single node, and found that the GPU-accelerated version paired with a single 16-core CPU performed around 1.5 times faster than the OpenMP-only version on dual 16-core CPUs (32 total cores).

VI. Concluding remarks

In this article, we reviewed the progress made in developing GPU-based computational fluid dynamics solvers for reactive flows. In addition, we presented a case study of a lid-driven cavity flow solver to provide an example of potential performance increase. In general, we observed trends of around an order of magnitude improvement in performance compared to versions running on multicore CPUs.

In a number of early efforts, comparisons were made between the performance of a GPU code and a serial CPU code running on a single processing core. Since nearly all modern CPUs consist of at least two—and commonly four or more—cores, and parallelization via OpenMP or MPI is common in scientific computing, future comparisons made should include a CPU version running in parallel on multiple cores. In addition,

comparing the performance normalized by both hardware cost and power consumption, as suggested by Zaspel and Griebel,[56] provides even more insight to the benefits of moving computationally expensive CFD codes to graphics processors.

A number of challenges remain in developing GPU solvers for simulations of flows with complex physics. For reactive-flow calculations, most strategies presented so far (e.g., by Spafford et al.,[78] Niemeyer et al.,[85] and Shi et al.[81,83,87]) performed the integration of the chemistry terms on the GPU while the controlling CPU handled transport calculations. Solvers based on this approach necessarily perform repeated CPU-GPU memory transfer of the thermochemical and species mass fraction variables, and suffer from the corresponding added latency. This could be balanced by the performance increase of calculating these expensive terms on the GPU, but it should be investigated whether migrating all calculations to the GPU would result in even higher performance, at the risk of leaving the CPU unnecessarily idle. Furthermore, no practical demonstration of a GPU chemistry solver capable of handling stiff chemistry has yet been made. This is one area where efforts need to be focused.

In addition, no GPU-accelerated simulations with other complex physical models such as multiphase flows have yet been demonstrated. Only two efforts, from the same group, have shown a GPU-based solver capable of simulating of two-phase, incompressible flows,[55,56] related to a rising air bubble in water.

A common question regarding GPU computing may be: How much can an application benefit from GPU acceleration? This may be answered by Amdahl's Law,[89] which states that the overall speedup achievable N, if a proportion P of the code can be accelerated by a factor of S, is

$$N = \frac{1}{(1-P) + \frac{P}{S}}. \tag{6}$$

Using this equation, if 50% of the application can be GPU-accelerated then the overall speedup is $2S/(1+S)$. Even if the portion performed on the GPU can be accelerated by a factor of 100, the overall application will only perform about twice as fast. On the other hand, if 99% of the application is parallelizable, then the overall potential speedup is around $100S/(99+S)$. Taking the case of Niemeyer et al.[85] as an example, for larger grid sizes the evaluation of the reactive source terms could be accelerated by a factor of 75. In this case, the overall speedup could reach about 43×. Therefore, taking full advantage of GPU acceleration can only be achieved if most of the application is parallelizable.

Early GPU work in many areas tackled "low-hanging fruit" problems that required little to no significant change in program structure. For example, an application that involves frequent large matrix inversions could simply replace the subroutine call from the CPU version to the GPU version. However, most software was designed to run on traditional, sequential processors (e.g., CPUs), so this "incremental" approach can only help so far. In order to truly harness the massive parallelism of many-core processors (and GPUs in particular), new programming strategies and new algorithms must be designed specifically targeted at massively parallel processors. In particular, reactive-flow solvers that require complex, implicit integration methods for stiff chemistry at every spatial location cannot be simply ported to the GPU, so alternative approaches must be developed in order to achieve a more useful improvement in performance. By developing new solvers for fluid flow simulations using GPUs to significantly improve performance, we can facilitate the transition of high-fidelity CFD from a tool of science to that of technology and design, enabling previously inaccessible levels of detail and parametric variation in modeling studies.

Acknowledgments

This work was supported by the National Science Foundation under Grant No. 0932559, the U.S. Department of Defense through the National Defense Science and Engineering Graduate Fellowship program, and the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0951783.

References

[1]Elsen, E., Houston, M., Vishal, V., Darve, E., Hanrahan, P., and Pande, V., "N-Body simulation on GPUs," *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, No. 188 in SC '06, ACM, New York, NY, USA, 2006.

[2]Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., and Schulten, K., "Accelerating molecular modeling applications with graphics processors," *J. Comput. Chem.*, Vol. 28, No. 16, 2007, pp. 2618–2640.

[3]Anderson, J. A., Lorenz, C. D., and Travesset, A., "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Comput. Phys.*, Vol. 227, No. 10, May 2008, pp. 5342–5359.

[4]Friedrichs, M. S., Eastman, P., Vaidyanathan, V., Houston, M., Legrand, S., Beberg, A. L., Ensign, D. L., Bruns, C. M., and Pande, V. S., "Accelerating molecular dynamic simulation on graphics processing units," *J. Comput. Chem.*, Vol. 30, No. 6, April 2009, pp. 864–872.

[5]Hardy, D. J., Stone, J. E., and Schulten, K., "Multilevel summation of electrostatic potentials using graphics processing units," *Parallel Comput.*, Vol. 35, No. 3, March 2009, pp. 164–177.

[6]Beberg, A. L., Ensign, D. L., Jayachandran, G., Khaliq, S., and Pande, V. S., "Folding@home: Lessons from eight years of volunteer distributed computing," *IEEE International Symposium on Parallel Distributed Processing*, May 2009.

[7]Vogt, L., Olivares-Amaya, R., Kermes, S., Shao, Y., Amador-Bedolla, C., and Aspuru-Guzik, A., "Accelerating Resolution-of-the-Identity Second-Order Møller–Plesset Quantum Chemistry Calculations with Graphical Processing Units," *J. Phys. Chem. A*, Vol. 112, No. 10, March 2008, pp. 2049–2057.

[8]Yasuda, K., "Accelerating Density Functional Calculations with Graphics Processing Unit," *J. Chem. Theory Comput.*, Vol. 4, No. 8, August 2008, pp. 1230–1236.

[9]Olivares-Amaya, R., Watson, M. A., Edgar, R. G., Vogt, L., Shao, Y., and Aspuru-Guzik, A., "Accelerating Correlated Quantum Chemistry Calculations Using Graphical Processing Units and a Mixed Precision Matrix Multiplication Library," *J. Chem. Theory Comput.*, Vol. 6, No. 1, 2010, pp. 135–144.

[10]Surkov, V., "Parallel option pricing with Fourier space time-stepping method on graphics processing units," *Parallel Comput.*, Vol. 36, No. 7, July 2010, pp. 372–380.

[11]Pagès, G. and Wilbertz, B., "GPGPUs in computational finance: massive parallel comput. for American style options," *Concurrency Computat. Pract. Exper.*, Vol. 24, No. 8, 2012, pp. 837–848.

[12]Fatone, L., Giacinti, M., Mariani, F., Recchioni, M. C., and Zirilli, F., "Parallel option pricing on GPU: barrier options and realized variance options," J. Supercomput., doi:10.1007/s11227-012-0813-7.

[13]Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., and Shi, Y., "Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)," J. Supercomput., doi:10.1007/s11227-011-0672-7.

[14]Sherbondy, A., Houston, M., and Napel, S., "Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware," *14th IEEE Visualization Conference*, 2003, pp. 171–176.

[15]Mueller, K. and Xu, F., "Practical considerations for GPU-accelerated CT," *3rd IEEE ISBI*, 2006, pp. 1184–1187.

[16]Birk, M., Guth, A., Zapf, M., Balzer, M., Ruiter, N., Hübner, M., and Becker, J., "Acceleration of image reconstruction in 3D ultrasound computer tomography: an evaluation of CPU, GPU and FPGA computing," *2011 Conference on Design and Architectures for Signal and Image Processing*, November 2011.

[17]Boyer, M., Tarjan, D., Acton, S. T., and Skadron, K., "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors," *2009 IEEE International Symposium on Parallel & Distributed Processing*, IEEE, 2009.

[18]Nimmagadda, V. K., Akoglu, A., Hariri, S., and Moukabary, T., "Cardiac simulation on multi-GPU platform," *J. Supercomput.*, Vol. 59, No. 3, 2011, pp. 1360–1378.

[19]Pratx, G. and Xing, L., "GPU computing in medical physics: A review," *Med. Phys.*, Vol. 38, No. 5, May 2011, pp. 2685–2697.

[20]Owens, J. D., Luebke, D., Govindaraju, N., Harris, M. J., Krueger, J., Lefohn, A. E., and Purcell, T. J., "A survey of general-purpose computation on graphics hardware," *Comput. Graphics Forum*, Vol. 26, No. 1, 2007, pp. 80–113.

[21]Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., "GPU Computing," *Proc. IEEE*, Vol. 96, No. 5, 2008, pp. 879–899.

[22]Munshi, A., "The OpenCL Specification," Khronos Group, November 2011.

[23]NVIDIA, *CUDA C Programming Guide*, 4th ed., 2011.

[24]Kirk, D. B. and Hwu, W.-M. W., *Programming Massively Parallel Processors*, Morgan Kaufmann, 2010.

[25]Sanders, J. and Kandrot, E., *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 1st ed., 2010.

[26]OpenACC, "OpenACC Application Programming Interface," http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.

[27]Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F., "accULL: An OpenACC Implementation with CUDA and OpenCL Support," *Euro-Par 2012 Parallel Processing*, edited by C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Vol. 7484 of *LNCS*, Springer-Verlag Berlin Heidelberg, 2012, pp. 871–882.

[28]Dagum, L. and Menon, R., "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, Vol. 5, No. 1, January 1998, pp. 46–55.

[29]Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R., *Parallel Programming in OpenMP*, Academic Press, San Diego, CA, 2001.

[30]OpenMP Architecture Review Board, "OpenMP Application Program Interface," http://www.openmp.org/mp-documents/OpenMP3.1.pdf, July 2011.

[31]Wienke, S., Springer, P., Terboven, C., and Mey, D. a., "OpenACC — First Experiences with Real-World Applications," *Euro-Par 2012 Parallel Processing*, edited by C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Vol. 7484 of *LNCS*, Springer-Verlag Berlin Heidelberg, 2012, pp. 859–870.

[32]Reyes, R., López, I., Fumero, J. J., and de Sande, F., "Directive-based Programming for GPUs: A Comparative Study," *IEEE 14th International Conference on High Performance Computing and Communications*, June 2012, pp. 410–417.

[33]Levesque, J. M., Sankaran, R., and Grout, R., "Hybridizing S3D into an Exascale Application using OpenACC," *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 15:1–15:11.

[34]Griebel, M., Dornseifer, T., and Neunhoeffer, T., *Numerical Simulation in Fluid Dynamics*, SIAM, Philadelphia, 1998.

[35]Adams, L. M. and Ortega, J. M., "A multi-color SOR method for parallel computation," ICASE report 82-9, ICASE-NASA Langley Research Center, Hampton, VA, 1982.

[36]Young, D. M., *Iterative Solution of Large Linear Systems*, Academic Press, New York, 1971.

[37]Liu, J., Ma, Z., Li, S., and Zhao, Y., "A GPU Accelerated Red-Black SOR Algorithm for Computational Fluid Dynamics Problems," *Adv. Mat. Res.*, Vol. 320, 2011, pp. 335–340.

[38]Vanka, S. P., Shinn, A. F., and Sahu, K. C., "Computational fluid dynamics using graphics processing units: challenges and opportunities," *ASME Conf. Proc.*, Vol. 2011, No. 54921, November 2011, pp. 429–437.

[39]Bolz, J., Farmer, I., Grinspun, E., and Schröder, P., "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Trans. Graphics*, Vol. 22, No. 3, 2003, pp. 917–924.

[40]Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A., "Simulation of cloud dynamics on graphics hardware," *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, 2003, pp. 92–101.

[41]Harris, M. J., "Fast Fluid Dynamics Simulation on the GPU," *GPU Gems*, edited by R. Fernando, Addison-Wesley, 2004, pp. 637–665.

[42]Krüger, J. and Westermann, R., "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graphics*, Vol. 22, No. 3, 2003, pp. 908–916.

[43]Stam, J., "Stable fluids," *SIGGRAPH '99*, ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128.

[44]Liu, Y., Liu, X., and Wu, E., "Real-Time 3D Fluid Simulation on GPU with Complex Obstacles," *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, 2004, pp. 247–256.

[45]Scheidegger, C. E., Comba, J. L. D., and da Cunha, R. D., "Practical CFD simulations on programmable graphics hardware using SMAC," *Comput. Graphics Forum*, Vol. 24, No. 4, 2005, pp. 715–728.

[46]Hagen, T. R., Lie, K.-A., and Natvig, J. R., "Solving the Euler Equations on Graphics Processing Units," *Computational Science - ICCS 2006, Part IV, Lecture Notes in Computer Science*, edited by V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, Springer-Verlag, Berlin Heidelberg, 2006, pp. 220–227.

[47]Brandvik, T. and Pullan, G., "Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware," *J. Mech. Eng. Sci.*, Vol. 221, No. 12, 2007, pp. 1745–1748.

[48]Brandvik, T. and Pullan, G., "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," *46th AIAA Aerospace Sciences Meeting*, 2008.

[49]Elsen, E., LeGresley, P., and Darve, E., "Large calculation of the flow over a hypersonic vehicle using a GPU," *J. Comput. Phys.*, Vol. 227, No. 24, December 2008, pp. 10148–10161.

[50]Molemaker, J., Cohen, J. M., Patel, S., and Noh, J., "Low viscosity flow simulations for animation," *2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, July 2008, pp. 9–18.

[51]Cohen, J. M. and Molemaker, M. J., "A Fast Double Precision CFD Code using CUDA," *Parallel CFD 2009*, 2009.

[52]Phillips, E. H., Zhang, Y., Davis, R. L., and Owens, J. D., "Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units," *47th AIAA Aerospace Sciences Meeting*, 2009.

[53]Shinn, A. F. and Vanka, S. P., "Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit," *ASME Conf. Proc.*, Vol. 2009, No. 43864, November 2009, pp. 125–133.

[54]Thibault, J. C. and Senocak, I., "CUDA Implementation of a Navier–Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," *47th AIAA Aerospace Sciences Meeting*, 2009.

[55]Griebel, M. and Zaspel, P., "A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier–Stokes equations," *Comput. Sci. Res. Dev.*, Vol. 25, No. 1–2, April 2010, pp. 65–73.

[56]Zaspel, P. and Griebel, M., "Solving incompressible two-phase flows on multi-GPU clusters," Comput. Fluids, doi:10.1016/j.compfluid.2012.01.021.

[57]Ran, W., Cheng, W., Qin, F., and Luo, X., "GPU accelerated CESE method for 1D shock tube problems," *J. Comput. Phys.*, Vol. 230, No. 24, 2011, pp. 8797–8812.

[58]Kuo, F.-A., Smith, M. R., Hsieh, C.-W., Chou, C.-Y., and Wu, J.-S., "GPU acceleration for general conservation equations and its application to several engineering problems," *Comput. Fluids*, Vol. 45, No. 1, 2011, pp. 147–154.

[59]Corrigan, A., Camelli, F. F., Löhner, R., and Wallin, J., "Running unstructured grid-based CFD solvers on modern graphics hardware," *Int. J. Numer. Methods Fluids*, Vol. 66, No. 2, 2011, pp. 221–229.

[60]Lefebvre, M., Guillen, P., Gouez, J.-M. L., and Basdevant, C., "Optimizing 2D and 3D structured Euler CFD solvers on Graphical Processing Units," Comput. Fluids, doi:10.1016/ j.compfluid.2012.09.013, 2012.

[61]Phillips, E. H., Davis, R. L., and Owens, J. D., "Unsteady Turbulent Simulations on a Cluster of Graphics Processors," *40th AIAA Fluid Dynamics Conference*, 2010.

[62]Jesperson, D. C., "Acceleration of a CFD code with a GPU," *Sci. Program.*, Vol. 18, No. 3–4, 2010, pp. 193–201.

[63]Kampolis, I. C., Trompoukis, X. S., Asouti, V. G., and Giannakoglou, K. C., "CFD-based analysis and two-level aerodynamic optimization on graphics processing units," *Comput. Meth. Appl. Mech. Eng.*, Vol. 199, No. 9–12, 2010, pp. 712–722.

[64]Asouti, V. G., Trompoukis, X. S., Kampolis, I. C., and Giannakoglou, K. C., "Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units," *Int. J. Numer. Methods Fluids*, Vol. 67, No. 2, May 2010, pp. 232–246.

[65]Brandvik, T. and Pullan, G., "An Accelerated 3D Navier–Stokes Solver for Flows in Turbomachines," *J. Turbomach.*, Vol. 133, No. 2, 2011, pp. 021025–1–021025–9.

[66]Gohari, S. M. I., Esfahanian, V., and Moqtaderi, H., "Coalesced computations of the incompressible Navier–Stokes equations over an airfoil using graphics processing units," Comput. Fluids, doi:10.1016/j.compfluid.2012.04.022.

[67]Shinn, A. F., Vanka, S. P., and Hwu, W. W., "Direct Numerical Simulation of Turbulent Flow in a Square Duct using a Graphics Processing Unit (GPU)," *40th Fluid Dynamics Conference & Exhibit*, June 2010.

[68]Alfonsi, G., Ciliberti, S. A., Mancini, M., and Primavera, L., "Performances of Navier–Stokes Solver on a Hybrid CPU/GPU Computing System," *Parallel Comput. Technologies, Lecture Notes in Computer Science*, edited by V. Malyshkin, Springer-Verlag, Berlin Heidelberg, 2011, pp. 404–416.

[69]Salvadore, F., Bernardini, M., and Botti, M., "GPU accelerated flow solver for direct numerical simulation of turbulent flows," J. Comp. Phys., doi:10.1016/j.jcp.2012.10.012.

[70]Shinn, A. F. and Vanka, S. P., "Large Eddy Simulations of Film-Cooling Flows With a Micro-Ramp Vortex Generator," *ASME Conf. Proc.*, Vol. 2011, No. 54921, November 2011, pp. 439–451.

[71]DeLeon, R. and Senocak, I., "GPU-Accelerated Large-Eddy Simulation of Turbulent Channel Flows," *50th AIAA Aerospace Sciences Meeting*, 2012.

[72]DeLeon, R., Jacobsen, D., and Senocak, I., "Large Eddy Simulations of Turbulent Incompressible Flows on GPU Clusters," *Comput. Sci. Eng.*, 2012.

[73]Burke, M. P., Chaos, M., Ju, Y., Dryer, F. L., and Klippenstein, S. J., "Comprehensive H2/O2 kinetic model for high-pressure combustion," *Int. J. Chem. Kinet.*, Vol. 44, No. 7, 2011, pp. 444–474.

[74]Qin, Z., Lissianski, V. V., Yang, H., Gardiner, W. C., Davis, S. G., and Wang, H., "Combustion chemistry of propane: a case study of detailed reaction mechanism optimization," *Proc. Combust. Inst.*, Vol. 28, 2000, pp. 1663–1669.

[75]Mehl, M., Pitz, W. J., Westbrook, C. K., and Curran, H. J., "Kinetic modeling of gasoline surrogate components and mixtures under engine conditions," *Proc. Combust. Inst.*, Vol. 33, 2011, pp. 193–200.

[76]Herbinet, O., Pitz, W. J., and Westbrook, C. K., "Detailed chemical kinetic mechanism for the oxidation of biodiesel fuels blend surrogate," *Combust. Flame.*, Vol. 157, No. 5, 2010, pp. 893–908.

[77]Lu, T. and Law, C. K., "Toward accommodating realistic fuel chemistry in large-scale computations," *Prog. Energy Combust. Sci.*, Vol. 35, No. 2, 2009, pp. 192–215.

[78]Spafford, K., Meredith, J., Vetter, J., Chen, J. H., Grout, R., and Sankaran, R., "Accelerating S3D: A GPGPU Case Study," *Euro-Par 2009 Parallel Processing Workshops, LNCS 6043*, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 122–131.

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

[80]Chen, J. H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E. R., Klasky, S., Liao, W. K., Ma, K. L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., and Yoo, C. S., "Terascale direct numerical simulations of turbulent combustion using S3D," *Comput. Sci. Discovery*, Vol. 2, No. 1, 2009, pp. 015001.

[81]Shi, Y., Green, W. H., Wong, H.-W., and Oluwole, O. O., "Redesigning combustion modeling algorithms for the Graphics Processing Unit (GPU): Chemical kinetic rate evaluation and ordinary differential equation integration," *Combust. Flame*, Vol. 158, No. 5, May 2011, pp. 836–847.

[82]Humphrey, J. R., Price, D. K., Spagnoli, K. E., Paolini, A. L., and Kelmelis, E. J., "CULA: Hybrid GPU Accelerated Linear Algebra Routines," *Proc. SPIE*, edited by E. J. Kelmelis, Vol. 7705, April 2010, p. 770502.

[83]Shi, Y., Oluwole, O. O., Wong, H.-W., and Green, W. H., "A multi-approach algorithm for enabling efficient application of very large, highly detailed reaction mechanisms in multi-dimensional HCCI engine simulations," *7th National Combustion Meeting*, 2011.

[84]Herbinet, O., Pitz, W. J., and Westbrook, C. K., "Detailed chemical kinetic oxidation mechanism for a biodiesel surrogate," *Combust. Flame.*, Vol. 154, No. 3, 2008, pp. 507–528.

[85]Niemeyer, K. E., Sung, C. J., Fotache, C. G., and Lee, J. C., "Turbulence-chemistry closure method using graphics processing units: a preliminary test," *7th Fall Technical Meeting of the Eastern States Section of the Combustion Institute*, October 2011.

[86]Yetter, R. A., Dryer, F. L., and Rabitz, H., "A comprehensive reaction mechanism for carbon monoxide/hydrogen/oxygen kinetics," *Combust. Sci. Tech.*, Vol. 79, 1991, pp. 97–128.

[87]Shi, Y., Green, W. H., Wong, H.-W., and Oluwole, O. O., "Accelerating multi-dimensional combustion simulations using hybrid CPU-based implicit/GPU-based explicit ODE integration," *Combust. Flame*, Vol. 159, No. 7, July 2012, pp. 2388–2397.

[88]Le, H. P., Cambier, J.-L., and Cole, L. K., "GPU-based flow simulation with detailed chemical kinetics," Comput. Phys. Commun., doi:10.1016/j.cpc.2012.10.013.

[89]Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Spring Joint Computer Conference*, 1967.