Accepted for Publication in Computers and Fluids, March 2015

Directive-Based GPU Programming for Computational Fluid Dynamics

Brent P. Pickering[a], Charles W. Jackson[b], Thomas R.W. Scogland[c], Wu-Chun Feng[d], Christopher J. Roy[e]

[a] Virginia Tech Dept. of Aerospace and Ocean Engineering, 215 Randolph Hall Blacksburg, VA 24061, United States, bpickeri@vt.edu (Corresponding Author)
[b] Virginia Tech Dept. of Aerospace and Ocean Engineering, 215 Randolph Hall Blacksburg, VA 24061, United States, cwj5@vt.edu
[c] Virginia Tech Dept. of Computer Science, 2202 Kraft Drive Blacksburg, VA 24060, United States, tom.scogland@gmail.com
[d] Virginia Tech Dept. of Computer Science, 2202 Kraft Drive Blacksburg, VA 24060, United States, feng@cs.vt.edu
[e] Virginia Tech Dept. of Aerospace and Ocean Engineering, 215 Randolph Hall Blacksburg, VA 24061, United States, cjroy@vt.edu

Directive-based programming of graphics processing units (GPUs) has recently appeared as a viable alternative to using specialized low-level languages such as CUDA C and OpenCL for general-purpose GPU programming. This technique, which uses "directive" or "pragma" statements to annotate source codes written in traditional high-level languages, is designed to permit a unified code base to serve multiple computational platforms. In this work we analyze the popular OpenACC programming standard, as implemented by the PGI compiler suite, in order to evaluate its utility and performance potential in computational fluid dynamics (CFD) applications. We examine the process of applying the OpenACC Fortran API to a test CFD code that serves as a proxy for a full-scale research code developed at Virginia Tech; this test code is used to asses the performance improvements attainable for our CFD algorithm on common GPU platforms, as well as to determine the modifications that must be made to the original source code in order to run efficiently on the GPU. Performance is measured on several recent GPU architectures from NVIDIA and AMD (using both double and single precision arithmetic) and the accelerator code is benchmarked against a multithreaded CPU version constructed from the same Fortran source code using OpenMP directives. A single NVIDIA Kepler GPU card is found to perform approximately $20\times$ faster than a single CPU core and more than $2\times$ faster than a 16-core Xeon server. An analysis of optimization techniques for OpenACC reveals cases in which manual intervention by the programmer can improve accelerator performance by up to 30% over the default compiler heuristics, although these optimizations are relevant only for specific platforms. Additionally, the use of multiple accelerators with OpenACC is investigated, including an experimental high-level interface for multi-GPU programming that automates scheduling tasks across multiple devices. While the overall performance of the OpenACC code is found to be satifactory, we also observe some significant limitations and restrictions imposed by the OpenACC API regarding certain useful features of modern Fortran (2003/8); these are sufficient for us to conclude that it would *not* be practical to apply OpenACC to our full research code at this time due to the amount of refactoring required.

Keywords: Graphics Processing Unit (GPU), Directive-based programming, OpenACC, Fortran, finite-difference method

1. Introduction

Many novel computational architectures have become available to scientists and engineers in the field of high performance computing (HPC) offering improved performance and efficiency through enhanced parallelism. One of the better known is the graphics processing unit (GPU), which was once a highly specialized device designed exclusively for manipulating image data but has since evolved into a powerful general purpose stream processor—capable of high computational performance on tasks exhibiting sufficient data parallelism. The high memory bandwidth and floating point throughput available in modern GPUs makes them potentially very attractive for computational fluid dynamics (CFD), however the adoption of this technology is hampered by the requirement that existing CFD codes be re-written in specialized low-level languages such as CUDA or OpenCL that more closely map to the GPU hardware. Using platform specific languages such as these often entails maintaining multiple versions of a CFD application, and given the rapid pace of hardware evolution a more portable solution is desired.

Directive-based GPU programming is an emergent technique that has the potential to significantly reduce the time and effort required to port CFD applications to the GPU by allowing the re-use of existing Fortran or C code bases [1]. This approach involves inserting "directive" or "pragma" statements into source code that instruct the compiler to generate specialized code in the areas designated by the programmer; because such statements are ignored by compilers when unrecognized, directives should permit an application to be ported to a new platform without refactoring the original code base. The particular scheme of directive-based programming examined in this work is OpenACC, which is a standard designed for parallel computing that emphasizes heterogeneous platforms such as combined CPU/GPU systems. OpenACC defines an API that will appear familiar to any programmer who has used OpenMP, making it straightforward for domain scientists to adopt. Additionally, OpenACC is relatively well supported, with major compiler vendors such as Cray and PGI providing implementations.

One of the principle objectives of this project was to evaluate OpenACC for use in an in-house CFD application called SENSEI, which is a multi-block, structured-grid, finite-volume code written in Fortran 03/08 that currently uses a combination of OpenMP and MPI for parallelism [2, 3]. SENSEI is designed to solve the compressible Navier-Stokes equations in three dimensions using a variety of time integration schemes and subgrid-scale (turbulence) models. It has a substantial code base that incorporates several projects into a single CFD framework—due to this complexity, applying OpenACC to SENSEI was anticipated to be a labor-intensive undertaking that could have unforeseen complications. Furthermore, because SENSEI is fully verified and being actively extended for ongoing research projects, major alterations to the code structure were seen as undesirable unless the performance benefits were very significant. It was therefore decided to first test OpenACC on a simpler surrogate code with the aim of uncovering any major difficulties or incompatibilities before work began on the full-scale code and permitting an easier analysis of various refactoring schemes. This proxy code, which is discussed in more detail in Section 2, was derived from a preexisting Fortran finite-difference code written to solve the incompressible Navier-Stokes equations. Its simplicity, small size and limited scope made major revisions and even creating multiple experimental versions feasible within a limited timeframe, yet the data layout, code structure and numerical algorithm are still representative of SENSEI and many other structured-grid CFD codes.

1.1. GPU Programming Considerations

Modern graphics processors derive most of their computational performance from an architecture that is highly specialized for data parallelism, sacrificing low-latency serial performance in favor of higher throughput. They are sometimes referred to as *massively parallel* because the hardware is capable of executing (and maintaining context for) thousands of simultaneous threads, which is two orders of magnitude greater than contemporary CPUs [4, 5]. To efficiently express this level of concurrency, common *general-purpose GPU* (GPGPU) languages (such as NVIDIA CUDA) use programming models that are intrinsically parallel, where user-specified threads are applied across an abstract computational space of parallel elements corresponding to the hierarchy of hardware resources (e.g., CUDA defines “thread blocks” that map to the “streaming multiprocessors”, and the thread blocks contain parallel “threads” that map to the CUDA cores) [5]. This fundamental difference between GPGPU programming languages and the languages traditionally used in CFD, such as C and Fortran, makes porting existing codes to GPU platforms a non-trivial task.

Current generation GPUs are strictly coprocessors and require a host CPU to control their operation. On most HPC systems, the CPU and GPU memory spaces are physically separate and must communicate via data transfers across a PCIe (*Peripheral Component Interconnect Express*) interface, so GPGPU programming models include functions for explicitly managing data movement between the host (CPU) and device (GPU) [5, 6]. Contemporary AMD and NVIDIA devices usually implement either the PCIe 2.0 or 3.0 standards, which permit maximum bandwidths of approximately 8GB/s or 16GB/s respectively. This is an order of magnitude lower than the RAM bandwidths typically seen in HPC, so for data-intensive applications the PCIe connection can easily become a bottleneck—in general, it is best practice to minimize data transactions between the GPU and host [5].

1.2. The OpenACC Programming Model

OpenACC is a standard designed to enable portable, parallel programming of heterogeneous architectures such as CPU/GPU systems. The high-level API is based around *directive* or *pragma* statements (in Fortran or C/C++ respectively) that are used to annotate sections of code to be converted to run on an accelerator or coprocessor (e.g., a GPU). In Fortran, these directives take the form of comment-statements similar to those used in OpenMP [6]:

```
!$acc directive-name [clause [[,] clause]…] new-line
```

As with OpenMP, the directives are used to designate blocks of code as being suitable for parallelization. Ideally, no modification of the original source code is necessary—within an appropriate region the compiler can recognize data parallelism in sequential structures, such as loops, and automatically convert this into equivalent functionality in an accelerator specific language.

The programming model defines two main constructs that are used to indicate parallel regions in a code: *parallel* and *kernels*. Each type of region can be entered via the respective "`!$acc parallel`" or "`!$acc kernels`" statement, and all operations contained within will be mapped to the accelerator device. The difference between the two lies in how program statements such as loop-nests are translated into accelerator functions. A *parallel* region represents a single target parallel operation that compiles to a single function on the device, and uses the same parallel configuration (e.g., number of threads) throughout. As an example, a *parallel* statement will correspond to a single CUDA kernel on an NVIDIA device, mapping all concurrent operations to the same kernel launch configuration. A *parallel* region requires that the programmer manually identify data-parallel loops using relevant clauses, otherwise they will default to sequential operations repeated across the parallel elements of the accelerator. This is analogous to the OpenMP *parallel* directive, which implicitly begins a set of worker threads that redundantly execute sequential program statements until a clause indicating a work-sharing loop is reached. By contrast, a *kernels* region can represent multiple target parallel operations and will map each loop-nest to a separate accelerator function, meaning that a single *kernels* construct might compile into multiple CUDA kernels. Manual annotation of loops is optional within a *kernels* region, as the compiler will attempt to automatically detect data-parallelism and generate the most appropriate decomposition for each loop—serial sections will default to *serial* accelerator functions [6].

OpenACC uses an abstract model of a target architecture that consists of three levels of parallelism: *gang, worker* and *vector*. Each level comprises one or more instances of the subsequent levels, meaning each *gang* will contain at least one *worker* which is itself divided into *vector* elements. The actual mapping from this representation into a lower-level accelerator programming model is specific to each target platform, so by default the decomposition of loop-nests is made transparent to the programmer. OpenACC does provide clauses permitting the user to override the compiler analysis and manually specify the *gang, worker* and *vector* arrangement—with appropriate knowledge of the target hardware, this can be used as a means of platform-specific optimization. On NVIDIA GPUs it is usual that the *gang* dimension will correspond to the number of CUDA thread-blocks while the *worker* and/or *vector* elements correspond to the threads within each block, so it is possible to use these clauses to specifically define a kernel launch configuration [7]. As will be discussed further in Section 4.2, this can have a significant effect on the performance of OpenACC code.

On heterogeneous platforms in which the host and device memory spaces are separate (which includes most contemporary GPUs) any data structures accessed within an accelerator region will be implicitly copied onto the device on entry and then back to the host when the region terminates. This automatic data management is convenient, but in many cases it is much less efficient than allowing the data to persist on the device across multiple kernel calls. For this reason, the OpenACC API also defines a *data* construct along with an assortment of data clauses that permit manual control over device memory. The *data* region behaves similarly to an accelerator region with regards to data movement, but it can be used to wrap large blocks of non-accelerator code to manage data across multiple accelerator regions. Using the various data clauses that can be appended to the accelerator and data directives, users can designate data structures to be copied on or off of the accelerator, or allocated only on the device as temporary storage. There is also an *update* clause that can be used within a *data* region to synchronize the host and device copies at any time.

2. CFD Code

The CFD code investigated in this paper solves the steady-state incompressible Navier-Stokes (INS) equations using the artificial compressibility method developed by Chorin [8]. The INS equations are a nonlinear system with an elliptic continuity equation that imposes a divergence free condition on the velocity field. In N dimensions, there are $N+1$ degrees of freedom (N velocity components and pressure). Letting ρ be the density constant and $\nu = \mu/\rho$ be the kinematic viscosity, the complete system takes the familiar form below.

$$\frac{\partial u_j}{\partial x_j} = 0 \tag{1}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} + \frac{1}{\rho}\frac{\partial p}{\partial x_i} - \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} = 0 \tag{2}$$

The artificial compressibility method transforms this INS system into a coupled set of hyperbolic equations by introducing a "pseudo-time" pressure derivative into the continuity equation. Since the objective is a steady state solution, the "physical" time in the momentum equations can be equated with the pseudo-time value and the following system of equations will result:

$$\frac{1}{\beta^2}\frac{\partial p}{\partial t} + \frac{\partial u_j}{\partial x_j} = 0 \tag{3}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} + \frac{1}{\rho}\frac{\partial p}{\partial x_i} - \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} = 0 \tag{4}$$

In Equation (3), β represents an artificial compressibility parameter which may either be defined as a constant over the entire domain or derived locally based on flow characteristics; the INS code does the latter, using the local velocity magnitude u_{local} along with user defined parameters u_{ref} (reference velocity) and r_κ to define $\beta^2 = \max\left(u_{local}^2\ ,\ r_\kappa * u_{ref}^2\right)$. The artificial viscosity based INS equations can be solved using any numerical methods suitable for hyperbolic systems—by iteratively converging the time derivatives to zero, the divergence free condition of the continuity equation is enforced and the momentum equations will reach steady state.

2.1. Discretization Scheme

The spatial discretization scheme employed in the INS code is a straightforward finite difference method with second order accuracy (using centered differences). To mitigate the odd-even decoupling phenomenon that can occur in the pressure solution, an artificial viscosity term (based on the fourth derivative of pressure) is introduced to the continuity equation, resulting in the following modification to Equation (3).

$$\frac{1}{\beta^2}\frac{\partial p}{\partial t} + \frac{\partial u_j}{\partial x_j} - \lambda_j \Delta x_j C_j \frac{\partial^4 p}{\partial x_j^4} = 0 \tag{5}$$

In this format, the C_j terms represent user adjustable parameters for tuning the amount of viscosity applied in each spatial dimension (typical values $\sim$0.01), while Δx_j represents the corresponding local grid spacing. The λ_j terms are determined from the local flow velocity as follows.

$$\lambda_j = \frac{1}{2}\left(|u_j| + \sqrt{u_j^2 + 4\beta^2}\right) \tag{6}$$

Note that in two dimensions, the fourth derivatives of pressure will require a 9-point stencil to maintain second order accuracy, while all other derivatives used in the solution need at most a 5-point stencil. These two dimensional stencils necessitate 19 solution-data loads per grid node or approximately 152B (76B) when using double (single) precision. For stencil algorithms such as this, which exhibit spatial locality in their data access patterns, much of the data can be reused between adjacent grid nodes via caching so that the actual amount of data loaded from main memory is significantly less. Effective use of cache or GPU shared memory (either by the programmer or compiler) is an important performance consideration that is discussed further in Section 4. The complete numerical scheme uses approximately 130 floating point operations per grid node, including three floating point

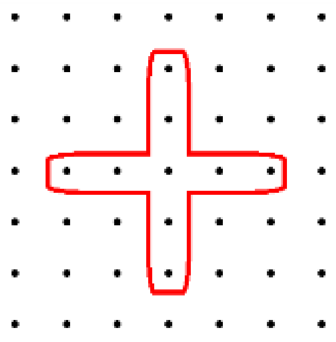

Figure 1. Illustration of a 9-point finite-difference stencil. *Required to approximate the fourth derivatives of pressure with second order accuracy.*

division operations and two full-precision square roots. Boundary conditions are implemented as functions independent from the interior scheme, and in cases where approximation is required (such as pressure extrapolation for a viscous wall boundary) second-order accurate numerical methods are employed.

The time-integration method used for all of the benchmark cases was forward-Euler. While the original INS code was capable of more efficient time-discretization schemes, a simple explicit method was selected (over implicit methods) because it shifts the performance focus away from linear equation solvers and sparse-matrix libraries and onto the stencil operations that were being accelerated.

2.2. Code Verification and Benchmark Case

The INS code was verified using the *method of manufactured solutions* (MMS) with an analytic solution based on trigonometric functions [9]. The code was re-verified after each major alteration, and it was confirmed that the final OpenACC implementation displayed the same level of discretization error and observed order of accuracy as the original Fortran version. As an additional check, the solutions to the benchmark case (described below) were compared between the versions and showed no discrepancy beyond round-off error.

Throughout this paper the INS code is used to run the familiar lid-driven cavity (LDC) problem in 2-dimensions, which is a common CFD verification case that also makes a good performance benchmark. All of the benchmark cases were run on a fixed-size square domain, with a lid velocity of $1\ m/s$, Reynolds number of 100 and density constant $1\ kg/m^3$. The computational grids used for the simulations were uniform and Cartesian, ranging in size from 128×128 to 8192×8192 nodes. A fully converged solution is displayed in Figure 2.

To evaluate relative performance of different versions of the code, the wall-clock time required for the benchmark to complete a fixed number of time-steps (1000) was recorded. Then, to make the results easier to compare between dissimilar grid sizes, this wall-clock time was converted into a GFLOPS (billion floating point operations per second) value based on the known number of floating point operations used for the calculations at each grid point. The GFLOPS metric is used in all figures and speedup calculations in subsequent sections, and is equivalent to a constant (130) multiplied by the rate of *nodes per second* computed. Note that division and square-root are counted as single floating-point operations in this metric, even though these are disproportionately slow on all the CPU and GPU architectures tested [5, 10]. Also note that, due to the smaller stable time-step, 1000 iterations results in a less converged solution for larger grid sizes than the smaller ones; this was considered acceptable for our purposes because computational performance was observed to be independent of the state of the solution (i.e., a nearly converged solution runs at the same rate as an initialized solution).

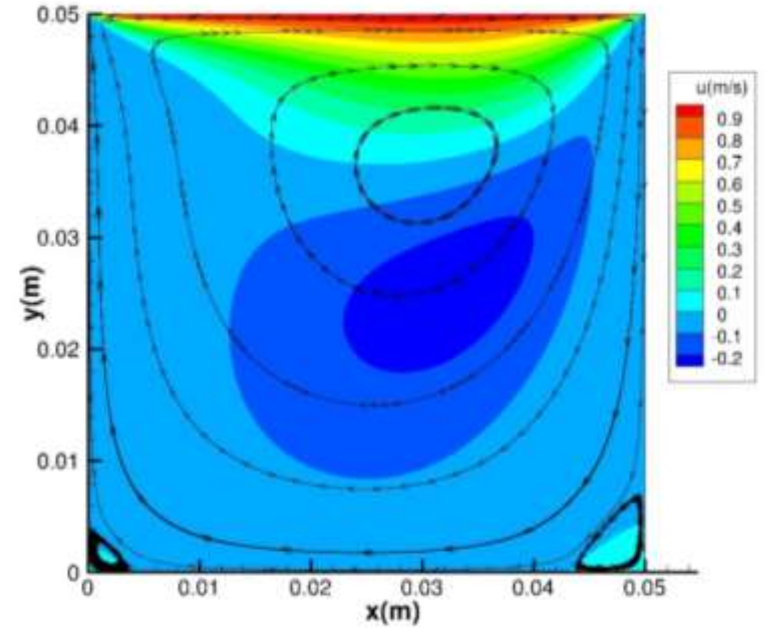

Figure 2. Horizontal velocity component and streamlines for a converged solution of the LDC benchmark.

3. Preliminary Code Modifications

Before attempting to migrate the INS code to the GPU, some general high-level optimizations were investigated. These were simple modifications to the Fortran code that required no language extensions or specialized hardware aware programming to implement, but yielded performance improvements across all platforms. Some of these alterations also reduced the memory footprint of the code, which permitted larger problems to fit into the limited GPU RAM. Additionally, the layouts of the main data structures were revised to permit contiguous access on the GPU (and other SIMD platforms). Note that although adjustments were made to the implementation, no alterations were made to the mathematical algorithm in any version of the INS code.

3.1. Reducing Memory Traffic

The most successful technique for reducing data movement was the removal of temporary arrays and intermediate data sets wherever possible by combining loops that had only *local* dependencies. As an example, the original version of the INS code computed an artificial viscosity term at each node in one loop, stored it in an array, and then accessed that array at each node in a separate residual calculation loop. By “fusing” the artificial viscosity loop into the residual loop, and calculating the artificial viscosity when needed at each node, it was possible to completely remove the artificial viscosity array and all associated data access.

In the INS finite-difference scheme a total of three performance-critical loops could be fused into one (artificial viscosity, local maximum time-step and residual) which resulted in an overall performance increase of approximately 2x compared to the un-optimized version (see Figure 3). It should be noted that fusing loops in this manner is not always straightforward for stencil codes since the stencil footprints may differ, meaning the domains or bounds of the loops are not the same. In this particular case, the artificial viscosity, local time-step and residual stencils were 9-point, 1-point and 5-point respectively, which necessitated “cleanup” loops along the boundaries and slightly increased the complexity of the code. It was also not possible to fuse *all* of the loops in the code because some operations, such as the pressure-rescaling step, were dependent on the output of preceding operations over the whole domain.

An additional modification made to decrease memory traffic was to replace an unnecessary memory-copy with a pointer-swap. The INS code includes multiple time integration methods, the simplest being an explicit Euler scheme. This algorithm reads solution data stored in one data structure (solution “A”) and writes the updated data to a second structure (solution “B”). In the original version, after solution B was updated, the data was copied from B back to A in preparation for the next time step. Obviously the same functionality can be obtained by swapping the data structures through a pointer exchange, thus avoiding the overhead of a memory-copy. The pointer swap was trivial to implement and resulted in an additional speedup of approximately 25% (Fig. 3). This could also be effective in more complex time integration algorithms that require storage of multiple solution levels, such as multistage Runge-Kutta methods, although the speedup may not be as significant.

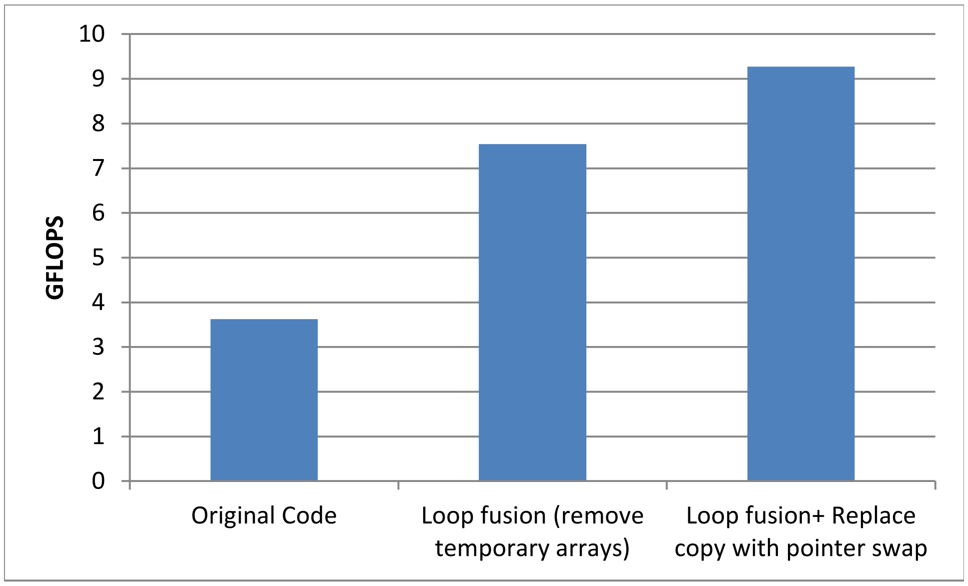

Figure 3. High-level optimizations applied to INS Fortran code, cumulative from left to right. *CPU performance on LDC benchmark, 512x512 grid. Dual-socket Xeon x5355 workstation, 8 threads / 8 cores (note that this is an older model than the Nehalem 8-core CPU benchmarked in Section 4.5).*

3.2. Data Structure Modification: AOS to SOA

One of the more extensive alterations made to the INS code when preparing for OpenACC involved revising the layout of all data structures to ensure contiguous access patterns for SIMD hardware. The main solution data in the code comprises a grid of nodes in two spatial dimensions with three degrees of freedom (DOF) per node corresponding to the pressure and two velocity components at each grid location. In the original version this data was arranged in an “array-of-struct” (AOS) format in which all three DOF were consecutive in memory for each node—meaning accessing a given DOF (e.g., pressure) for a set of multiple consecutive nodes produced a non-unit-stride (non-contiguous) access pattern. To permit efficient SIMD loads and stores across sequential grid nodes the data structure was altered to a “struct-of-array” (SOA) format, in which it was essentially broken into three separate two 2D arrays, each containing a single DOF over all the nodes (Figure 4). Contiguous memory transactions are usually more efficient on SIMD hardware because they avoid resorting to scatter-gather addressing or shuffling of vector operands; both Intel and NVIDIA recommend the SOA format for the majority of array data access [5, 10].

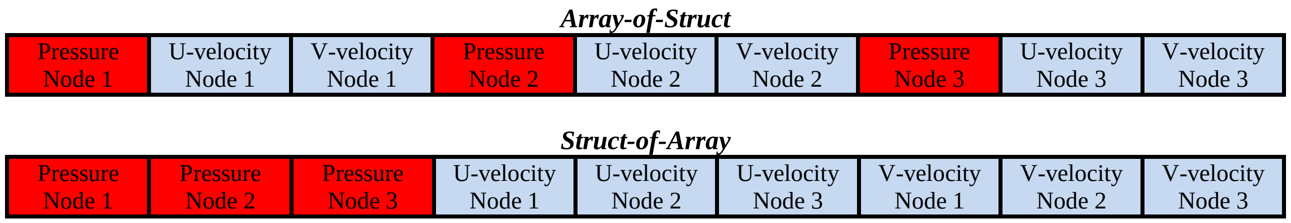

Figure 4. Illustration of the “array-of-struct” and “struct-of-array” layouts for a sequence of 3 grid nodes in linear memory (3-DOF per node). *Red cells represent access of the pressure field for three consecutive nodes.*

4. Porting to the GPU with OpenACC

Because the INS code was being used to test potential enhancements to a full-scale CFD research code, it was important to examine not just performance metrics but the entire procedure involved when using the OpenACC API. The objective here was to evaluate its practicality for use with complex CFD applications, including assurance that OpenACC would not require major alterations to the original source code that might complicate maintenance or degrade the performance on the CPU. It was also desirable that OpenACC work well with modern Fortran features and programming practices, including object-oriented extensions such as derived types [2, 11].

The OpenACC code was constructed from the branch incorporating the high-level optimizations and data-structure refactoring described in Section 3, so the memory layout was already configured for contiguous access. The next task involved experimenting with OpenACC data clauses to determine the optimal movement of solution data between host and device. As expected, best efficiency was observed when the entire data structure was copied onto the GPU at the beginning of a run and remained there until all iterations were complete, thus avoiding frequent large data transfers across the PCIe bus. Maintaining the solution data on the GPU was accomplished by wrapping the entire time-iteration loop in a “`!$acc data`” region, and then using the “`present`” clause within the enclosed subroutines to indicate that the data was already available on the device. If the data were needed on the host between iterations (to output intermediate solutions, for example) the “`!$acc update`” directive could be used in the data-region to synchronize the host and device data structures, however this practice was avoided whenever possible as it significantly reduced performance (updating the host data on every iteration increased runtime by an order of magnitude). The only device to host transfers strictly required between iterations were the norms of the iterative residuals, which consisted of three scalar values (one for each primitive variable) that were needed to monitor solution convergence. Small transactions were generated implicitly by the compiler whenever the result of an accelerator reduction was used to update another variable; these were equivalent to synchronous *cudaMemcpy* calls of a single scalar value.

For the 2D INS code, which has a single-block grid structure and only required two copies of the solution for explicit time integration, problems with over 200 million double-precision DOF fit easily into the 5-6GB of RAM available on a single compute card. This was aided by the reduced memory footprint that the optimizations in Section 3.1 provided; if the temporary arrays had not been removed, the total memory used would be approximately 30% greater. Codes with more complicated 3D data structures, more advanced time integration schemes and/or additional source terms would likely see fewer DOF fit into GPU memory, so either data would have to be transferred to and from host memory on each iteration or multiple GPUs would be needed to run larger problems.

Ideally, applying OpenACC directives to existing software should be possible without any modification to the original source code; however, we encountered two instances where restrictions imposed by the API forced some refactoring. One case stemmed from a requirement (of OpenACC 1.0) that any subroutine called within an accelerator region be inlined, which in Fortran means that the call must satisfy all the criteria for automatic inlining by the compiler [6]. For the INS code this was merely inconvenient, necessitating manual intervention where one subroutine was causing difficulty for the PGI 13.6 compiler, however this inlining requirement also has the significant consequence of prohibiting function pointers within accelerator regions. Function pointers form the basis of runtime polymorphism in object-oriented languages such as C++ and Fortran 2003 (e.g., virtual methods, abstract interfaces) which can be very useful in practical CFD applications—for example, SENSEI uses such capabilities present in Fortran 2003 to simplify calling boundary condition routines [2]. Applications that use these language features would need to be rewritten to work with OpenACC, possibly by replacing polymorphic function calls with complicated conditionals. Even in the newer OpenACC 2.0 standard (which relaxes the requirements on function inlining) there is no support for function pointers in accelerator code [12], and although CUDA devices do support the use of function pointers [5] this is not necessarily true for every supported accelerator platform, so it seems likely that OpenACC users will have to work with this restriction for now.

Another inconvenience for modern Fortran programs is the prohibition of allocatables as members of derived types. Within an accelerator region, the versions of the PGI compiler that we tested did not permit accessing allocatable arrays that were part of a user-defined type, although arrays consisting of user-defined types were allowed (Figure 5). This is unfortunate because using a derived type to hold a set of arrays is a convenient method of expressing the SOA data layout, which is better for contiguous memory access on the GPU. In the relatively simple INS code this was easy to work around, but in the full scale research code derived types are used extensively to manage allocated data structures, so applying OpenACC would entail more refactoring [2].

```
!AOS: permitted in accelerator regions
  Velocity(i,j)%V1 = 0.0_dp
  Velocity(i,j)%V2 = 0.0_dp
```

```
!SOA: not permitted in accelerator regions
  Velocity%V1(i,j) = 0.0_dp
  Velocity%V2(i,j) = 0.0_dp
```

Figure 5. Example AOS and SOA derived types. ***Allocated arrays of derived types are permitted (left), but types containing allocated arrays are not (right).***

4.1. OpenACC Performance

In this section the computational performance of the OpenACC code is evaluated on three models of NVIDIA GPU representing two distinct microarchitectures—the specifications of the test devices are presented in Table A1 of the Appendix. The compiler used for these test cases was PGI 13.6, which implements the OpenACC 1.0 standard. This version was only capable of generating accelerator code for CUDA enabled devices, limiting the selection of hardware to that made by NVIDIA; in Section 4.4 a newer version (PGI 14.1) is also evaluated which is capable of compiling for AMD devices, and the code is run on AMD 7990 and 7970 cards. Compilation was carried out using the flags "`-O4 -acc -Mpreprocess -Minfo=accel -mp -Minline`" set for all test runs. No additional architecture specific flags or code modifications were used—the PGI compiler was capable of automatically generating binaries for CUDA *compute capability* 1.x, 2.x and 3.x devices.

A check of the accelerator-specific compiler output (generated with the "`-Minfo=accel`" flag) indicated that shared memory was being used to explicitly cache the solution data around each thread-block. The statement "`Cached references to size [(x+4)x(y+4)x3] block`" corresponds correctly to the dimensions needed for the 9-point finite difference stencil in the interior kernel, however the exact shared memory layout and access pattern cannot be determined without direct examination of the generated CUDA code. As shown by the red lines in Figure 6, preliminary benchmark results displayed steady high performance on medium and large grid sizes with declining performance as problem size decreased—this could be the result of kernel call overhead or less efficient device utilization on the smallest grids. These results were obtained using the default OpenACC configuration; as discussed in the next section, this performance can in some cases be improved upon through tunable parameters that are part of the OpenACC API (Figure 6, green lines).

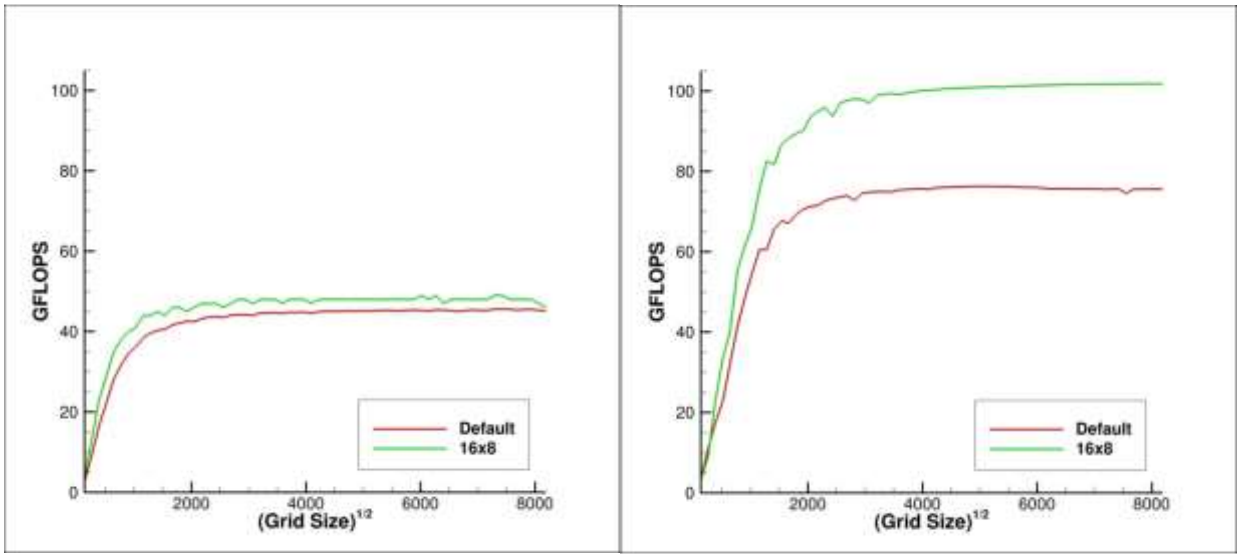

Figure 6. Double-precision performance of LDC benchmark on NVIDIA C2075 (left) and K20x (right). *The red line indicates the default OpenACC kernel performance, while the green line indicates the maximum performance attained through tuning with the vector clause. The K20c results are not pictured, but are approximately 12% lower than the K20x for all grid sizes.*

4.2. Tuning OpenACC

The OpenACC standard is designed to be transparent and portable, and provides only a few methods for explicitly controlling the generated device code. Notable examples are the *gang* and *vector* clauses, which as described in Section 1.2 can be used to manually specify the CUDA thread-block and grid dimensions: by default, the PGI compiler automatically defines grid and block dimensions based on its analysis of the user code, but this behavior can be overridden by inserting *gang* and *vector* parameters near the directives annotating loops. On CUDA devices, the launch configuration can have a significant influence on kernel performance because it affects the utilization of multiprocessor resources such as shared memory and registers, and can lead to variations in occupancy. More subtly, changes to the *shape* of the thread blocks in kernels using shared memory can also affect the layout of the shared data structures themselves, which in turn might lead to bank conflicts that reduce effective bandwidth [5].

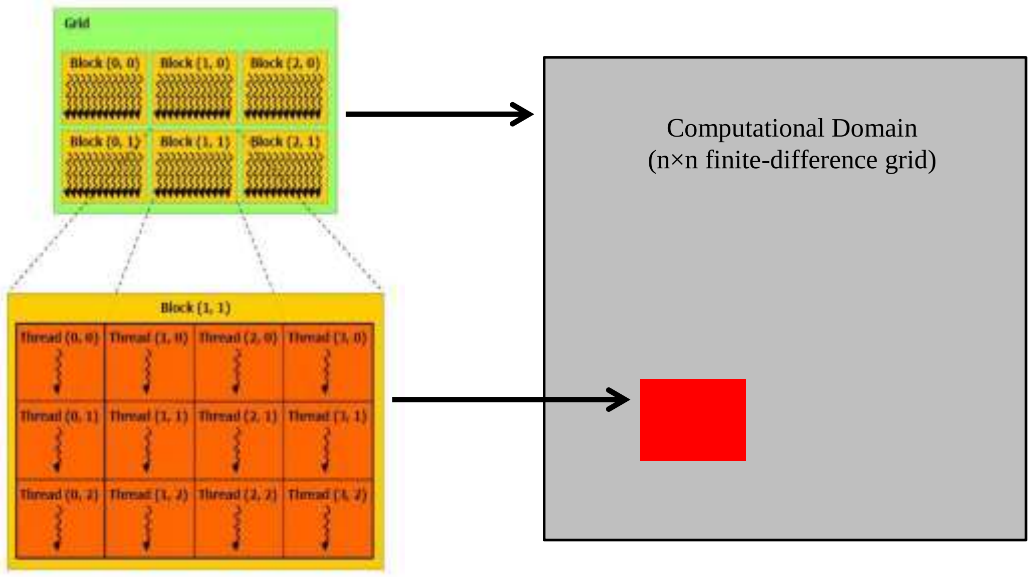

Figure 7. Mapping of CUDA thread-blocks to computational domain. (Adapted from [5]) *Each CUDA thread-block maps to a rectangular subsection of the solution grid, with each thread performing independent operations on a single grid-node. Altering the x and y dimensions of the thread-blocks can significantly affect performance.*

For the INS code, compiler output indicated that the interior kernel (which was derived from two tightly nested loops) defaulted to a grid of 2-dimensional 64x4 thread-blocks, while the boundary scheme used 1-dimensional blocks with 128 threads each; this same structure was used for both compute capability 2.x and 3.x binaries. The larger dimension (64) in the interior kernel corresponds to the `blockDim.x` parameter in CUDA C and was mapped by OpenACC to the *inner loop*, which seems like a reasonable heuristic since this will result in each of the eight 32-thread warps accessing data in a contiguous (unit-stride) manner. To test if this configuration was actually optimal for all architectures, the code was modified using the *vector* clause so that the 2D block dimensions of the interior scheme could be specified at compile time. The compiler was permitted to choose the number of blocks to launch (*gang* was not specified) and the entire parameter space was explored for a fixed size problem of 50 million DOF. A surprising observation made during this test was that the total number of threads per block was limited to a maximum of only 256; larger numbers would still compile, but would generate an error at runtime. This was unexpected because the number of threads per block permitted by compute capability 2.0 and greater should be up to 1024 [5]. Exactly why this limitation exists was never determined—there was no evidence that more than 256 threads would consume excessive multiprocessor resources (e.g., shared memory), while the runtime error messages were inconsistent between platforms and seemed unrelated to thread-block dimensions. It could be speculated that there is some “behind the scenes” allocation at work that is not expressed in the compiler output (such as shared memory needed for the reduction operations) but this could not be verified.

On the Fermi device it was found that the compiler default of 64x4 was not the optimal thread-block size, although the difference between default and optimal performance was small. As seen in Figure 8, best double precision performance occurs at block sizes of 16x8 and 16x4, both of which yield nearly 48 GFLOPS compared to 44.6 GFLOPS in the default configuration. This is a difference of less than 8%, so there appears to be little benefit in manually tuning the OpenACC code in this instance. For Kepler, a similar optimal block size of 16x8 was obtained, however the difference in performance was much more significant: on the K20c, the 64x4 default yielded 68.5 GFLOPS while 16x8 blocks gave 90.6 GFLOPS, an increase of over 30% (Figure 8). This result illustrates a potential tradeoff between performance and coding effort in OpenACC—relying on compiler heuristics does not necessarily yield peak performance, however it avoids the complexity of profiling and tuning the code for different problem/platform combinations.

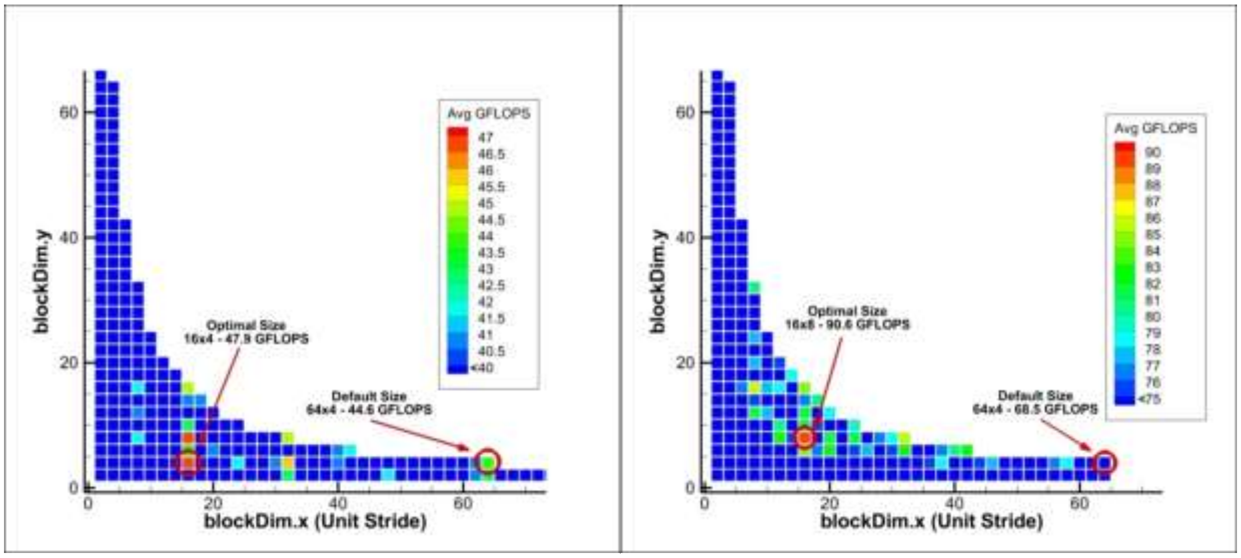

Figure 8. Double-precision performance vs. thread-block dimensions for a fixed-size LDC benchmark. *(4097x4097, 50 million DOF). NVIDIA C2075 (left) and K20c (right)*

4.3. Single Precision Results

The use of single-precision (32-bit) floating point arithmetic has the potential to be much more efficient than double-precision on NVIDIA GPUs. Not only is the maximum throughput 2-3x greater for single-precision operations, but 32-bit data types also require only half the memory bandwidth, half the register file and half the shared memory space of 64-bit types. In the Fortran code it was trivial to switch between double and single-precision simply by redefining the default precision parameter used for the *real* data type; because the INS code used the *iso_c_binding* module for interoperability with C code, the precision was always specified as either *c_double* or *c_float* [11].

It is beyond the scope of this paper to analyze the differences between double and single precision arithmetic in CFD, however for the particular case of the INS code running the LDC benchmark it was observed that single precision was perfectly adequate to obtain a converged solution; in fact, there was no discernible difference between the double and single precision results beyond the expected round-off error (about 5 significant digits). The LDC benchmark uses a uniform grid, which is probably more amenable to lower precision calculations than a grid with very large ratios in node spacing (e.g., for a boundary layer), but the result does constitute an example in which single precision is effective for incompressible flow calculations. The combination of OpenACC and Fortran made alternating between double and single precision versions of the INS code very straightforward.

On both the Fermi and Kepler devices the PGI compiler defaulted to the same 64x4 thread-block size that it used for the double precision cases, and as before this was found to be suboptimal. As seen in Figure 9, the C2075 was observed to perform best with a block size of 16x6, attaining almost 17% better performance than default, while the K20c saw only a 3% speedup over default at its best size of 32x4. Interestingly, the default configuration on the K20c was nearly optimal for single precision but performed poorly for double precision, while the reverse was true for Fermi—there was no single block size that provided peak (or near-peak) performance on both architectures for both the single and double precision cases. This reiterates the notion that heuristics alone are insufficient for generating optimum code on the GPU and illustrates the difficulty of tuning for multiple platforms.

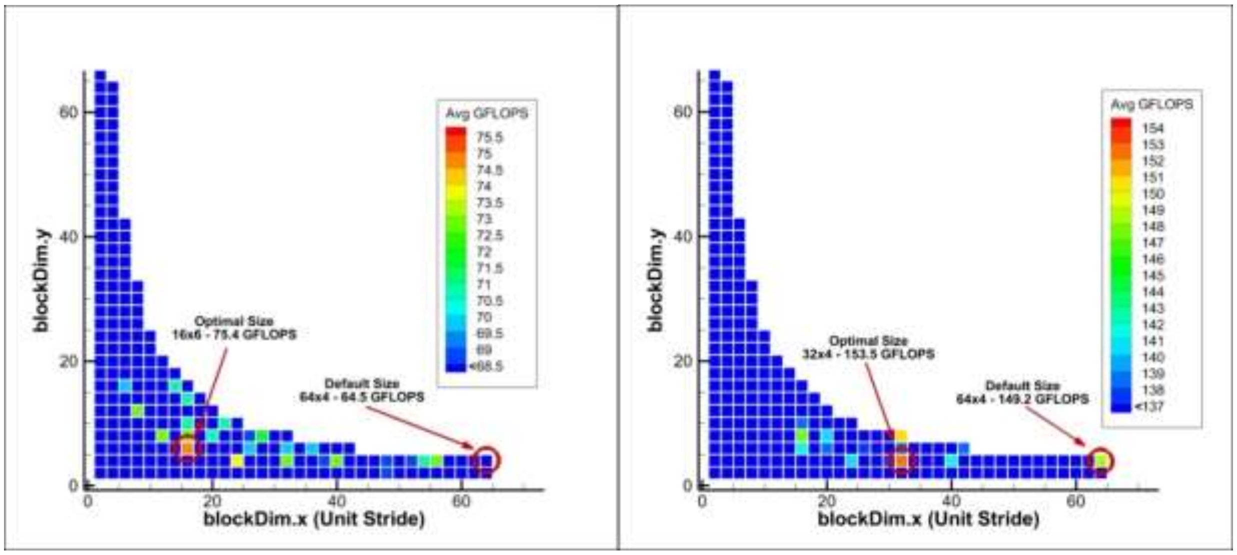

Figure 9. Single-precision performance vs. thread-block dimensions for a fixed-size LDC benchmark. ***(4097x4097, 50 million DOF). NVIDIA C2075 (left) and K20c (right)***

The speedups observed with single precision arithmetic were less impressive than expected based on the theoretical throughputs given in Table A1. On the C2075, the difference was about 50%, while the K20c saw a speedup of more than 100% at the default block size and about 70% for the tuned configuration. These numbers are still significant, however, and given the ease with which the code can alternate between double and single precision it is probably worth testing to see if the full-scale CFD code achieves acceptable accuracy at lower precision. A comparison of the double and single precision results at default and optimal block sizes are displayed in Figure 10.

Tesla C2075 (Fermi)	**Default Block Size** (GFLOPS)	**Optimal Block Size** (GFLOPS)	Speedup (%)
Double Precision (GFLOPS)	44.6	47.9	7.4%
Single Precision (GFLOPS)	64.5	75.4	16.9%
Speedup (%)	44.6%	57.4%	

Tesla K20c (Kepler)	**Default Block Size** (GFLOPS)	**Optimal Block Size** (GFLOPS)	Speedup (%)
Double Precision (GFLOPS)	68.5	90.6	32.3%
Single Precision (GFLOPS)	149.2	153.5	2.9%
Speedup (%)	117.8%	69.4%	

Optimal Thread-block Dimension	Double Precision	Single Precision
C2075 (Fermi)	16x4 (16x8)	16x6
K20c (Kepler)	16x8	32x4

Figure 10. Effects of thread-block optimizations and single vs double precision artithmetic on OpenACC performance. ***Fixed-size LDC benchmark: 4097x4097, 50 million DOF.***

4.4. Targeting Multiple GPUs and Architectures with OpenACC

While OpenACC is designed to provide a portable programming model for accelerators, there are certain configurations that still require manual intervention by a programmer. Perhaps the most important of these is the use of multiple accelerators. When a region is offloaded with OpenACC, it is offloaded to exactly one device. A number of modifications are necessary to allow support for multiple devices, and further to support multiple devices of multiple types. In order to explore this issue, we evaluated our test code with two approaches to spreading work across multiple GPUs: we created a version that manually partitions the work and hand-tuned the data transfers and also evaluated an extension to OpenACC/OpenMP that automatically provides multi-device support.

In order to exploit multiple devices when available, the application needs to be structured such that the work can be divided into a number of independent workloads, or blocks, much like the more traditional issue with distributed computing models such as MPI. Our 2D INS code lends itself to a straightforward decomposition into blocks along the y-axis, resulting in contiguous chunks of memory for each block and only boundary rows to be exchanged between devices across iterations. The multi-GPU version determines how many blocks are required by using the "`acc_get_num_devices(<device_type>)`" API function provided by OpenACC, but this in and of itself presents an issue. While the API is quite simple, the device_type is implementation defined, and the PGI accelerator compiler provides no generic device type for "accelerator" or "GPU." Instead, the options are ACC_DEVICE_NVIDIA, ACC_DEVICE_RADEON and ACC_DEVICE_HOST. Since the number of accelerators must be known, and there is no generic way to compute it, we check the availability of both NVIDIA and AMD Radeon devices, falling back on the host CPU whenever the others are unavailable.

Given the number of devices, an OpenMP parallel region creates one thread per device, and sets the device of each thread based on its ID. As with the single GPU case, data transfer costs are a concern, so each thread uses a data-region to copy in the section of the problem it requires. Unlike the single GPU case, the data cannot all be left on each GPU across iterations, since the boundary values all must be copied back to main memory at the end of each iteration and exchanged to other GPUs before the beginning of the next. This change incurs both extra synchronization between threads, and extra data movement, meaning that the multi-GPU version is less efficient on a single device than the single-GPU version described above. To mitigate this, we implemented the transfer to only copy the boundary elements back and forth, and to do that asynchronously. While this does not completely offset the cost, it does lower it considerably. Mechanisms do exist to transfer these values directly from one GPU to another, but they are not exposed through the OpenACC programming model at this time.

We evaluated this version of the code with a recently released version of the PGI OpenACC compiler, version 14.1 with support for both NVIDIA and AMD Radeon GPUs. The GPU hardware specs are described in Tables A1 and A2 in the Appendix, and results are presented in Figure 11. Even with a relatively simple decomposition, the INS code clearly benefits from the use of multiple GPUs, scaling 3.8 times from one NVIDIA c2070 to four, or nearly linear scaling onto four devices. The NVIDIA k20x system performs better than the c2070s, and in fact on a single k20x outperform the k20c described earlier by a small margin. The Kepler architecture seems to be materially more sensitive to the extra synchronization overhead and the high register usage of the application than the Fermi architecture for this code. Finally, the AMD architecture proves to perform extremely well for this kind of application, with a single consumer-grade GPU board containing two GPU dies outperforming both of our NVIDIA configurations by a relatively wide margin. This is especially unexpected due to the lower theoretical peak floating point performance per die on the AMD 7990. Based on our tests, and discussions with PGI engineers, the result appears to be due to the simpler architecture of the AMD GPU making it easier to attain close to theoretical peak on the hardware, where the NVIDIA GPUs might still perform better with a greater degree of hand-optimization.

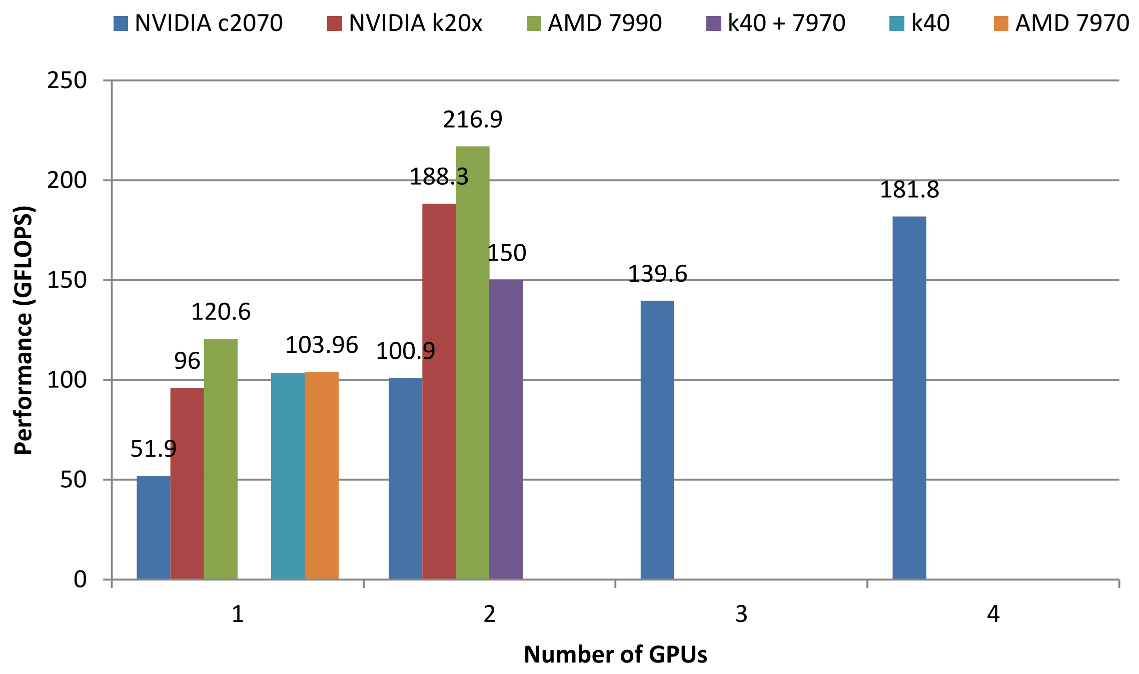

Figure 11. Double-precision performance and scaling of multi-GPU LDC benchmark. *NVIDIA c2070, NVIDIA k20x and AMD Radeon 7990, across number of GPUs for a fixed grid size of 7000^2.*

In addition to our manual version, we also evaluate a version using an experimental multi-GPU interface, called CoreTSAR [13]. In essence, CoreTSAR allows users to specify the associations between their computations and data, and uses that information to automatically divide the work across devices, managing the splitting, threading and data transfer complexities internally. The resulting implementation is simpler than the manual version evaluated above, but not necessarily as tuned. Figure 12 shows our results with both versions scaling across the four NVIDIA Fermi c2070 GPU system. As you can see from these results, CoreTSAR does in fact scale with the additional GPUs, but not as well as the manual version, scaling to only 2.6x rather than nearly 4x for the manual version on four GPUs. The reason appears to be an increase in the cost of the boundary exchange after each step, since CoreTSAR requires the complete data set to be synchronized with the host to merge data before sending the updated data back out. As solutions like CoreTSAR mature, we suggest that they focus on offering simple and efficient boundary transfer mechanisms to better support CFD applications.

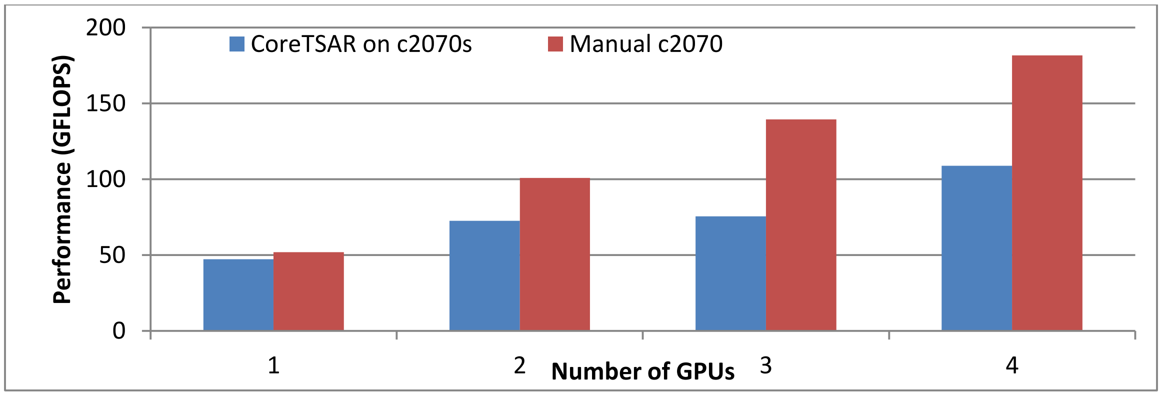

Figure 12. **Scaling of manual and CoreTSAR scheduled versions of LDC across GPUs.**

4.5. Comparisons with CPU performance

To evaluate the performance benefits of OpenACC acceleration, the results from the GPU benchmarks were compared against the INS code re-compiled for multicore x86 CPUs. This represents a unified INS code base capable of running on either the CPU or GPU. It is identical to the Fortran+OpenACC code discussed throughout Section 4 except that the OpenACC directives are switched off in favor of OpenMP statements (using the preprocessor) and the compiler flags are modified appropriately to generate efficient CPU code: "`–O4 -Minfo –Mpreprocess -mp=all -Mvect -Minline`". The compiler used for the CPU case was PGI 13.6, and according to the output from the "`–Minfo`" flag this compiler was able to automatically generate vector instructions

(SSE, AVX) for the target CPU platforms. The same version was also used to generate code for the NVIDIA platforms, while version 13.10 was used for the AMD GPU.

The CPU test hardware consisted of two dual-socket Intel Xeon workstations that were selected to match the GPU hardware of comparable generations. Thus there is an 8-core Nehalem workstation to match the Fermi GPUs, and a 16-core Sandy Bridge for comparison against the Keplers. More detailed specifications for these test platforms are given in the appendix (Table A3). In Figure 13 we compare the maximum double-precision performance attained by each platform on large grids (with over 4000×4000 nodes). In Figure 14, the relative speedups of each GPU vs the two CPU platforms are shown.

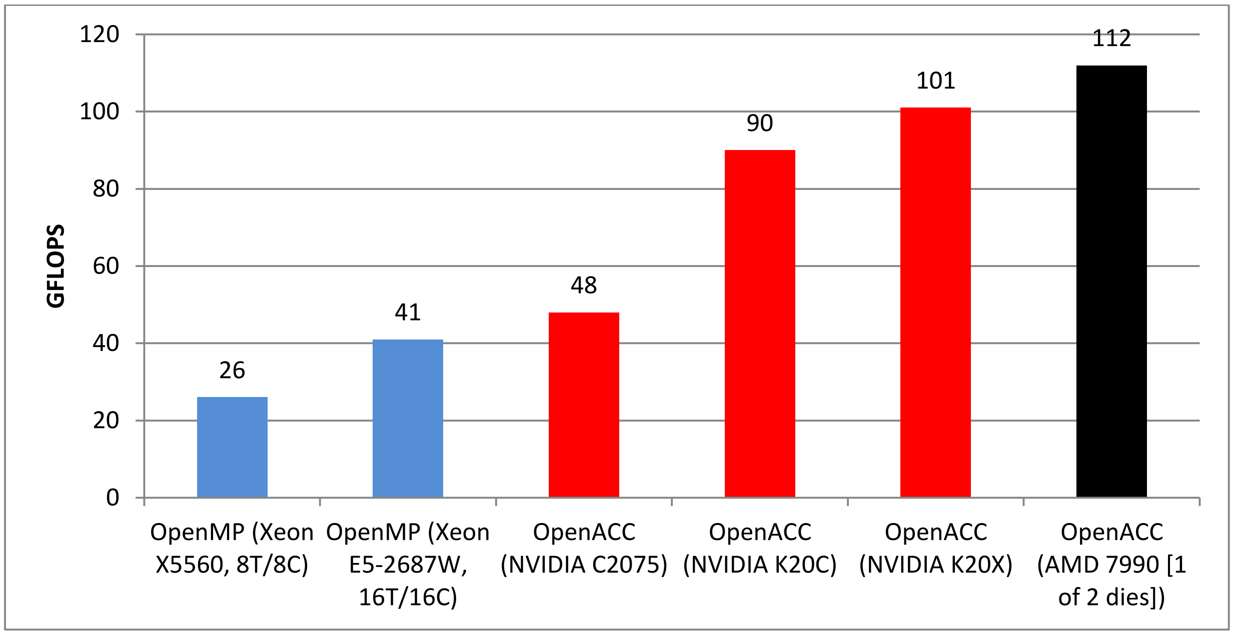

Figure 13. Maximum double-precision performance attained on large LDC benchmark cases (over 4000×4000). *Color indicates the compiler and directives used: Blue = PGI 13.6 Fortran/OpenMP; Red = PGI 13.6 Fortran/OpenACC; Black = PGI 13.10 Fortran /OpenACC. (Note that the PGI 13.10 compiler shows slightly reduced performance compared to the 14.1 version tested in Section 4.4.)*

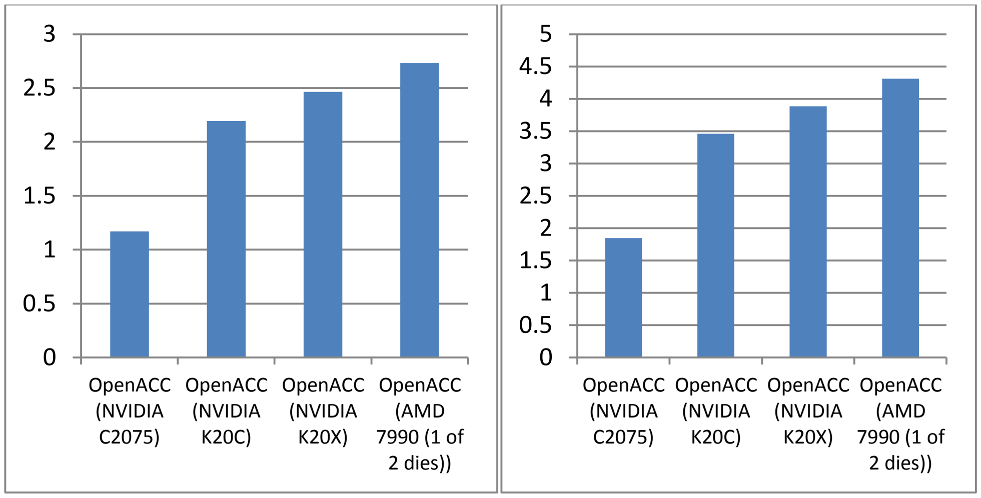

Figure 14. Relative speedup of each GPU platforms vs CPU platforms. (Left) *Speedup over Sandy Bridge E5-2687W 16-cores*. (Right) *Speedup over Nehalem X5560 8-cores.*

The results presented in Figure 13 represent the *best* performance attained for each platform on large LDC benchmark cases (over 4000×4000 nodes); this means that the *most optimized* versions of the OpenACC code from Section 4 were used, not the default. In the case of the CPU (OpenMP) version of the code, the grid size was selected carefully to return the best results, because the actual performance varied significantly with the *size* of the grid on the CPU platforms, as illustrated in Figure 15. The precise reasons for this erratic performance are not completely understood, but the small oscillations visible in the plot could be the result of variations in the address alignment of the solution array columns, which can have an effect on the bandwidth of SIMD load/store operations on Intel CPUs [10, 14]. The larger performance drops (such as the 4096×4096 grid size) are most likely caused by cache *conflict misses*, which have been shown to have significant performance ramifications for stencil codes on x86-64 hardware [15, 16]. A complete analysis of the CPU performance is well beyond the scope of this work, however these results are presented here to highlight an important advantage of the software-managed shared memory on the GPU, which is free from the effects of cache associativity and shows relatively uniform performance.

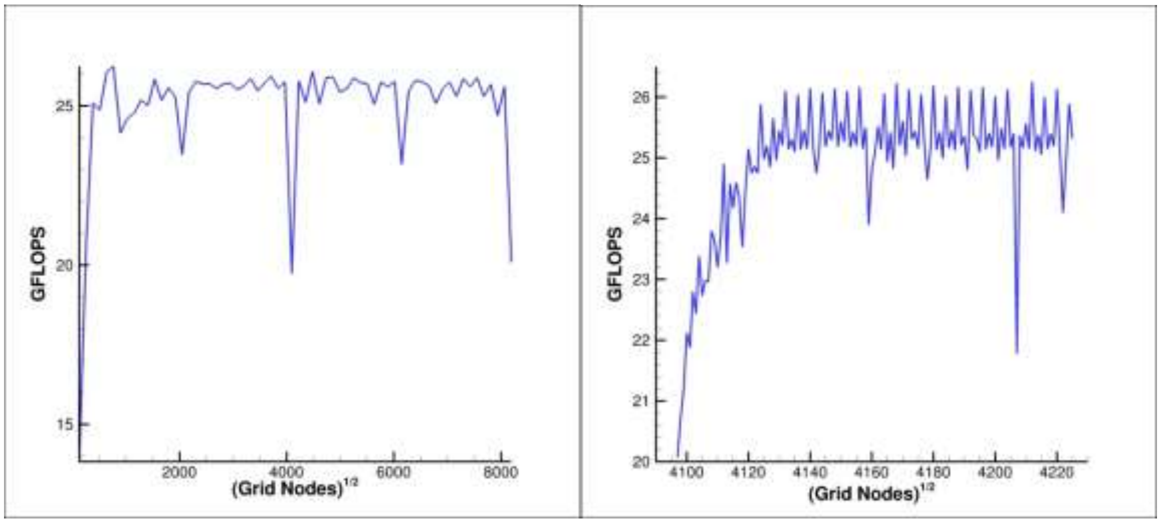

Figure 15. Performance of CPU (OpenMP) code vs grid size. ***Xeon X5560 workstation, 16T/8C.*** (Left) ***All grid sizes from 128×128 to 8192×8192, in increments of 128.*** (Right) ***Detail of grid sizes above 4096×4096, in increments of 1.***

5. Conclusions

The directive-based OpenACC programming model proved very capable in our test CFD application, permitting the creation of an efficient cross-platform source code. Using the implementation of OpenACC present in the PGI compiler suite, we were able to construct a portable version of the finite-difference INS code with a unified Fortran code base that could be compiled for either x86 CPU or NVIDIA GPU hardware. The performance of this application on a single high-end NVIDIA card showed an average speedup of more than 20× vs. a single CPU core, and approximately 2.5× when compared to the equivalent OpenMP implementation run on a dual-socket Xeon server with 16 cores. Results with multiple GPUs displayed excellent scalability as well, and tests with a version of the PGI compiler capable of generating code for AMD GPUs illustrated how an OpenACC application can be *transparently* extended to new accelerator architectures, which is one of the main benefits of directive-based programming models.

It was noted that some non-trivial modifications to the original version of the Fortran INS code were required before it would run efficiently on the GPU, particularly alterations to the data structures needed for contiguous memory access. These modifications required additional programming effort, but did not harm the CPU performance of the INS code—in fact, they appear to have improved CPU performance in some cases (possibly due to better compiler vectorization). More significant setbacks were the restrictions imposed by OpenACC 1.0 and the PGI compiler on device function calls and derived types containing allocatables, both of which disable very useful features of modern Fortran (e.g. function pointers). These limitations were sufficient for us to decide that it would *not* be productive to apply OpenACC to the Fortran 2003 code SENSEI at this time, as it would require too many alterations to the present code structure. We are instead investigating newer directive-based APIs that may ameliorate some of these issues, particularly the OpenMP 4.0 standard which promises to support similar functionality as OpenACC (on both CPU and GPU platforms) [17].

Appendix

The specifications of several CPU and GPU platforms are listed in the tables below for reference.

Model	Architecture	Compute Capability	ECC	Memory (Size)	Memory (Bandwidth)	Peak SP GFLOPS	Peak DP GFLOPS
Tesla C2070	Fermi GF110	2.0	Yes	6 GB	144 GB/s	1030	515
Tesla C2075	Fermi GF110	2.0	Yes	6 GB	144 GB/s	1030	515
Tesla K20c	Kepler GK110	3.5	Yes	5 GB	208 GB/s	3520	1170
Tesla K20x	Kepler GK110	3.5	Yes	6 GB	250 GB/s	3950	1310
Tesla K40	Kepler GK110	3.5	Yes	12 GB	288 GB/s	4290	1430

Table A1. NVIDIA GPUs.

Model	Architecture	ECC	Memory (Size)	Memory (Bandwidth)	Peak SP GFLOPS	Peak DP GFLOPS
HD 7970	Southern Islands GCN	No	3 GB	264 GB/s	3788	947
HD 7990	Southern Islands GCN	No	3GB (x2)	288 GB/s (x2)	4100 (x2)	947 (x2)

Table A2. AMD GPUs. *The HD 7970 is a single die version of the HD 7990.*

Model	Architecture	ISA Extension	Threads / Cores	Clock (MHz)	ECC	Memory (Bandwidth)	Peak SP GFLOPS	Peak DP GFLOPS
X5355	Core	SSSE3	4/4	2667	Yes	21 GB/s	84	42
X5560	Nehalem	SSE4.2	8/4	3067	Yes	32 GB/s	98	49
E5-2687W	Sandy Bridge	AVX	16/8	3400	Yes	51 GB/s	434	217

Table A3. Intel Xeon CPUs. *Clock frequency represents maximum sustained "turbo" level with all cores active. Theoretical GFLOPS were calculated based on these clock frequencies and using the given ISA extensions.*

Acknowledgments

This work was supported by an Air Force Office of Scientific Research (AFOSR) Basic Research Initiative in the Computational Mathematics program with Dr. Fariba Fahroo serving as the program manager.

Bibliography

[1] R. Reyes, I. Lopez, J. Fumero and F. de Sande, "Directive-based Programming for GPUs: A Comparative Study," in *IEEE 14th International Conference on High Performance Computing and Communications*, Liverpool, 2012.

[2] J. M. Derlaga, T. S. Phillips and C. J. Roy, "SENSEI Computational Fluid Dynamics Code: A Case Study in Modern Fortran Software Development," in *21st AIAA Computational Fluid Dynamics Conference*, San Diego, 2013.

[3] B. P. Pickering, C. W. Jackson, T. R. Scogland, W.-C. Feng and C. J. Roy, "Directive-Based GPU

Programming for Computational Fluid Dynamics," in *AIAA Scitech, 52 Aerospace Sciences Meeting*, National Harbor, Maryland, 2014.

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

[5] NVIDIA Corporation, *CUDA C Programming Guide, Version 5.5,* 2013.

[6] *The OpenACC Application Programming Interface, Version 1,* OpenACC, 2011.

[7] C. Woolley, "Profiling and Tuning OpenACC Code," in *GPU Technology Conference*, 2012.

[8] A. Chorin, "A Numerical Method for Solving Incompressible Viscous Flow Problems," *Journal of Computational Physics,* vol. 2.1, pp. 12-26, 1967.

[9] P. Knupp and K. Salari, *Verification of Computer Codes in Computational Science and Engineering,* Chapman & Hall/CRC, 2003.

[10] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual,* 2013.

[11] W. Brainerd, Guide to Fortran 2003 Programming, Springer-Verlag, 2009.

[12] *The OpenACC Application Programming Interface, Version 2,* 2013.

[13] T. R. W. Scogland, W.-C. Feng, B. Roundtree and B. R. de Supinski, "CoreTSAR: Core Task-Size Adapting Runtime," *IEEE Transactions on Parallel and Distributed Systems,* 2014.

[14] Intel Corporation, *Intel Architecture Instruction Set Extensions Programming Reference,* 2012.

[15] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf and K. Yelick, "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, TX , 2008.

[16] M. J. Livesey, *Accelerating the FDTD Method Using SSE and Graphics Processing Units,* Manchester: University of Manchester, 2011.

[17] *OpenMP Application Programming Interface, Version 4.0,* OpenMP Architecture Review Board, 2013.

[18] P. N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture,* NVIDIA Corporation, 2009.

[19] *AMD Graphics Cores Next (GCN) Architecture,* Advanced Micro Devices, Inc., 2012.