Evaluating the OpenACC API for Parallelization of CFD Applications

Brent P. Pickering

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Aerospace Engineering

Christopher J. Roy, Chair
Wu-chun Feng
Kevin G. Wang

June 27, 2014
Blacksburg, VA

Keywords: graphics processing unit (GPU), computational fluid dynamics (CFD), directive-based programming, parallel programming, OpenACC, Fortran 2003, stencil code, finite-volume method
Evaluating the OpenACC API for Parallelization of CFD Applications

Brent P. Pickering

ABSTRACT

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 and to simplify the transition of legacy codes to new architectures. This work analyzes 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. Of particular interest is the handling of stencil algorithms, which are an important component of finite-difference and finite-volume numerical methods. To this end, the process of applying the OpenACC Fortran API to a preexisting finite-difference CFD code is examined in detail, and all modifications that must be made to the original source in order to run efficiently on the GPU are noted. Optimization techniques for OpenACC are also explored, and it is demonstrated that tuning the code for a particular accelerator architecture can result in performance increases of over 30%. There are also some limitations and programming restrictions imposed by the API: it is observed that certain useful features of modern Fortran (2003/8) are effectively disabled within OpenACC regions. Finally, a combination of OpenACC and OpenMP directives is used to create a truly cross-platform Fortran code that can be compiled for either CPU or GPU hardware. The performance of the OpenACC code is measured on several contemporary NVIDIA GPU architectures, and a comparison is made between double and single precision arithmetic showing that if reduced precision can be tolerated, it can lead to significant speedups. To assess the performance gains relative to a typical CPU implementation, the execution time for a standard benchmark case (lid-driven cavity) is used as a reference. The OpenACC version is compared against the identical Fortran code recompiled to use OpenMP on multicore CPUs, as well as a highly-optimized C++ version of the code that utilizes hardware aware programming techniques to attain higher performance on the Intel Xeon platforms being tested. Low-level optimizations specific to these architectures are analyzed and it is observed that the stencil access pattern required by the structured-grid CFD code sometimes leads to performance degrading conflict misses in the hardware managed CPU caches. The GPU code, which primarily uses software managed caching, is found to be free from these issues. Overall, it is observed that the OpenACC GPU code compares favorably against even the best optimized CPU version: using a single NVIDIA K20x GPU, the Fortran+OpenACC code is seen to outperform the optimized C++ version by 20% and the Fortran+OpenMP version by more than 100% with both CPU codes running on a 16-core Xeon workstation.
ACKNOWLEDGEMENTS

First, I must thank the Air Force Office of Scientific Research for providing funding for this project as part of a Basic Research Initiative (BRI) grant to investigate the co-design of CFD codes and hardware. The work described in this thesis is only a small part of a much larger undertaking supported through this grant, which involves researchers from multiple departments at both Virginia Tech and North Carolina State University. I would also like to thank my academic advisor, Dr. Christopher Roy, for his guidance, patience and direction over the course of this research project and my graduate studies in general.

Special thanks go to our BRI project collaborators in the Virginia Tech Department of Computer Science and Synergy Lab, in particular Dr. Wu-chun Feng who provided invaluable advice and insight into the computational platforms being tested, and also Thomas Scogland, who aided me in the development and debugging of the OpenACC-based CFD codes and carried out most of the research into multi-GPU scaling cited in this thesis. I also thank Dr. Kevin Wang for providing his knowledge of computational methods and GPU programming and helping to review this thesis. Without such input from these and other specialists this project could not have been a success.

I also want to thank my fellow grad students in the Virginia Tech Department of Aerospace and Ocean Engineering (AOE) CFD research group, especially Chip Jackson who assisted with the benchmarking process and wrote all of the scripts used for data collection. Of course this work would not have been possible without the help of the AOE computing staff, including Steve Edwards, who maintained the latest compiler versions and took the time to tweak BIOS settings on the workstations when necessary for my research. Finally I thank my parents, Ray and Jayne Pickering, for their support, and also my brother Todd who assisted me with the memory scaling tests on the Xeon platforms.

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.
Contents

1 Introduction ......................................................................................................................... 1

1.1 Related Work .................................................................................................................. 4

1.2 Parallelism in Computational Hardware ........................................................................ 10

1.2.1 Instruction-Level Parallelism ..................................................................................... 11

1.2.2 Task Parallelism .......................................................................................................... 13

1.2.3 Data Parallelism .......................................................................................................... 14

1.3 Memory and Cache ........................................................................................................ 16

1.3.1 Software-Managed Cache ............................................................................................ 17

1.3.2 Hardware-Managed Cache ........................................................................................... 17

1.4 GPU Hardware and Programming .................................................................................. 20

1.4.1 NVIDIA Fermi ............................................................................................................. 21

1.4.2 NVIDIA Kepler ............................................................................................................ 23

1.4.3 CUDA Programming Model ....................................................................................... 25

1.4.4 OpenACC .................................................................................................................... 27

1.5 CPU Hardware and Programming ................................................................................... 29

1.6 Programming for x86 SIMD ......................................................................................... 32

2 CFD Code .......................................................................................................................... 33

2.1 Discretization Scheme ................................................................................................... 34

2.2 Code Verification and Benchmark Case ......................................................................... 35

3 Preliminary Code Modifications ....................................................................................... 37

3.1 Reducing Memory Traffic .............................................................................................. 37

3.2 Data Structure Modification ........................................................................................... 39

4 Porting to the GPU with OpenACC ................................................................................ 41

4.1 OpenACC Performance Testing .................................................................................... 43

4.2 Tuning OpenACC ............................................................................................................ 44

4.3 Single Precision Results ................................................................................................ 47

4.4 Multi-GPU Scaling ......................................................................................................... 49

5 Performance Tuning on x86 CPUs ................................................................................... 51

5.1 SIMD Vectorization ....................................................................................................... 51

5.2 Cache-Aware Programming ............................................................................................ 57

5.3 Simultaneous Multithreading ......................................................................................... 60

6 Comparisons of CPU and GPU Performance .................................................................... 63

7 Conclusions ....................................................................................................................... 66
Appendix........................................................................................................................................... 68
A.1 Arithmetic Throughput vs Memory Bandwidth............................................................... 68
A.2 Hardware Specifications ................................................................................................. 68
Bibliography ...................................................................................................................................... 70
List of Figures

Figure 1. A simple 2D finite-difference stencil. Derivatives at the red node are numerically approximated using information from the surrounding “stencil” of green nodes. ................................................................. 3

Figure 2. A 2D multi-block structured grid around a turbine-blade. This example was used for the TurboStream solver by Brandvik and Pullan, but is identical to the class of grids used with the target SENSEI code described in the introduction. Reproduced from T. Brandvik and G. Pullan, “SBlock: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms,” in 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), 2010. Used under fair use, 2014. ........... 6

Figure 3. Various implementation techniques and optimizations applied to a finite-difference stencil algorithm by Datta, et al. Results plotted as double-precision GFLOPS. The differing effectiveness of the implementations on each platform illustrates the difficulty in establishing cross-platform heuristics and generating optimal code without architecture specific tuning. Reproduced from 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. Used under fair use, 2014 .............................................................................................................. 9

Figure 4. Division of a 64-bit memory address into the tag, index and offset fields needed for a 256KB 8-way associative cache using a 64B cache line. This is similar to the L2-cache found in the Intel Nehalem and Sandy Bridge microarchitectures [14]. ................................................................. 19

Figure 5. CUDA programming model. Reproduced from NVIDIA Corporation, CUDA C Programming Guide, Version 5.5, 2013. Used under fair use, 2014 .............................................................................................................. 26

Figure 6. 128-bit SSE packed data types. Reproduced from Intel Corporation, Intel 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes 1,2A,2B,3A,3B and 3C, 2012. Used under fair use, 2014 .............................................................................................................. 30

Figure 7. Illustration of a 9-point finite-difference stencil ................................................................. 35

Figure 8. Horizontal velocity component and streamlines for a converged solution of the LDC benchmark on a 512x512 node grid. .............................................................................................................. 36

Figure 9. Original iteration algorithm for the INS code. Green boxes indicate 2D operations on the rectangular domain, green boxes are 1D loops along the boundary. .............................................................................................................. 38

Figure 10. Modified iteration algorithm for the INS code with artificial viscosity, time-step and residual operations fused into a single loop. Grey boxes indicate 2D operations on the rectangular domain, green boxes are 1D loops along the boundary. .............................................................................................................. 38

Figure 11. 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 Chapter 5). ...... 39

Figure 12. 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. ................................................................. 40

Figure 13. Example AOS and SOA derived types using Fortran 03 syntax. Allocated arrays of derived types are permitted (left), but types containing allocated arrays are not (right). ........................................ 43

Figure 14. Double-precision performance of LDC benchmark on NVIDIA C2075 (left) and K20c (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. ................................................................. 44

Figure 15. Mapping of CUDA thread-blocks to computational domain. 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. Adapted from NVIDIA Corporation, CUDA C Programming Guide, Version 5.5, 2013. Used under fair use, 2014 .............................................................................................................. 45

Figure 16. Double-precision performance vs. thread-block dimensions for a fixed-size LDC benchmark.
(4097x4097, 50 million DOF). NVIDIA C2075 (left) and K20c (right) ................................................................. 47

Figure 17. Single-precision performance vs. thread-block dimensions for a fixed-size LDC benchmark.
(4097x4097, 50 million DOF). NVIDIA C2075 (left) and K20c (right) ................................................................. 48

FIGURE 19. SIMD DATA LOADS FOR A 5-POINT STENCIL USING SSE2. EACH REGISTER LOADS TWO CONTIGUOUS DOUBLE PRECISION FLOATING POINT VALUES (16 BYTES) NEEDED FOR TWO INDEPENDENT STENCILS—THE INNER-LOOP IS UNROLLED BY STRIDE 2, AND THE DATA IS STORED IN A COLUMN MAJOR ARRAY. CLEARLY, THE (i, j) LOAD AND THE (i±1, j) LOADS CANNOT BOTH BE 16 BYTE ALIGNED BECAUSE THEIR STARTING ADDRESSES DIFFER BY 8 BYTES. ..................... 53

FIGURE 20. SIMD INSTRUCTIONS APPLIED TO C++ INS CODE ON SANDY BRIDGE. DUAL SOCKET XEON E5-2687W, 16 THREADS / 16 CORES (LEFT), 32 THREADS / 16 CORES (RIGHT). NOTE THAT SOFTWARE PREFETCH INSTRUCTIONS ARE BEING USED, BUT THERE IS NO DATA STRUCTURE PADDING IN THESE CASES. .......................................................... 54

FIGURE 21. SIMD INSTRUCTIONS APPLIED TO C++ INS CODE ON NEHALEM. DUAL SOCKET XEON X5560, 16 THREADS / 8 CORES. NOTE THAT SOFTWARE PREFETCH INSTRUCTIONS ARE BEING USED, BUT THERE IS NO DATA STRUCTURE PADDING IN THESE CASES. ........................................................................ 54

FIGURE 22. EFFECT OF MEMORY CLOCK SPEED ON INS CODE USING THREE TYPES OF SIMD INSTRUCTIONS. XEON E5-2650 @ 2.4 GHZ, 16 THREADS / 8 CORES. ................................................................. 55

FIGURE 23. EFFECTS OF CACHE-BLOCKING ON INS CODE PERFORMANCE. XEON X5560, 16 THREADS / 8 CORES. ............... 58

FIGURE 24. PERFORMANCE VARIATIONS WITH RESPECT TO GRID SIZE ON NEHALEM (LEFT). DETAIL FOR GRID SIZES BETWEEN 4096 AND 4224 (RIGHT). XEON X5560, 16T/8C. THE FORTRAN AND C++ CODES BOTH DISPLAYED VERY SIMILAR PATTERNS (USING BOTH THE PGI AND GCC COMPILERS). ................................................................. 59

FIGURE 25. (A) THE DISTANCE IN LINEAR MEMORY BETWEEN ADJACENT STENCIL NODES IS DETERMINED BY THE NUMBER OF NODES IN THE UNIT-STRIDE DIMENSION. (B) ILLUSTRATION OF THE ARRAY PADDING TECHNIQUE. ................................. 60

FIGURE 26. EFFECTS OF ARRAY-PADDING AND SOFTWARE PREFETCH OPTIMIZATIONS APPLIED TO C++ INS CODE. NEHALEM MICRORARCHITECTURE, XEON X5560, 16 THREADS, 8 CORES. .................................................. 62

FIGURE 27. PEAK DOUBLE-PRECISION PERFORMANCE ATTAINED BY TEST PLATFORMS ON THE LDC BENCHMARK. DETAILS FOR EACH TEST CASE ARE GIVEN IN TABLE 5. KEY: (GREEN) INTEL XEON CPUs, (RED) NVIDIA GPUs, (BLACK) AMD GPUs. ................................. 64

FIGURE 28. RELATIVE SPEEDUP OF GPU PLATFORMS VS A SINGLE CPU CORE. (LEFT) SPEEDUP OVER SINGLE SANDY BRIDGE E5-2687W CORE. (RIGHT) SPEEDUP OVER SINGLE NEHALEM X5560 CORE. ........................................................................ 65
List of Tables

TABLE 1. OPENACC THREAD-BLOCK OPTIMIZATION RESULTS FOR TESLA C2075 ............................................. 49
TABLE 2. OPENACC THREAD-BLOCK OPTIMIZATION RESULTS FOR TESLA K20c .................................................. 49
TABLE 3. OPTIMAL THREAD-BLOCK DIMENSIONS OBTAINED FOR DIFFERENT GPU ARCHITECTURES. *NOTE THAT FOR THE FERMI ARCHITECTURE, TWO BLOCK SIZES WERE OPTIMAL FOR DOUBLE PRECISION. .............................................. 49
TABLE 4. LATENCY AND THROUGHPUT FOR SEVERAL FLOATING POINT INSTRUCTIONS ON SANDY BRIDGE. (A) AVX-256 (B) AVX-128 (C) SCALAR 64-BIT. NOTE THAT ALL OF THESE INSTRUCTIONS ARE USED IN PERFORMANCE-CRITICAL SECTIONS OF THE INS CODE ................................................................. 57
TABLE 5. SPECIFICATIONS FOR THE TEST CASES DISPLAYED IN FIGURE 27 ...................................................... 57
TABLE A1. NVIDIA GPUs ........................................................................................................................................ 68
TABLE A2. AMD GPUs. THE HD 7970 IS A SINGLE DIE VERSION OF THE HD 7990 .................................................. 68
TABLE A3. INTEL XEON CPUs. CLOCK FREQUENCY REPRESENTS MAXIMUM "TURBO" LEVEL WITH ALL CORES ACTIVE. THEORETICAL GFLOPS WERE CALCULATED BASED ON THESE CLOCK FREQUENCIES AND USING THE GIVEN ISA EXTENSIONS. ......................................................................................... 69
### Nomenclature and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>arithmetic logic unit</td>
</tr>
<tr>
<td>AOS</td>
<td>array-of-struct</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>B</td>
<td>byte (8-bits)</td>
</tr>
<tr>
<td>CFD</td>
<td>computational fluid dynamics</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture (a proprietary NVIDIA computing platform and programming model)</td>
</tr>
<tr>
<td>DOF</td>
<td>degrees of freedom</td>
</tr>
<tr>
<td>FPU</td>
<td>floating point unit</td>
</tr>
<tr>
<td>GB</td>
<td>giga- $(10^3)$ byte</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>giga- $(10^9)$ floating point operations per second</td>
</tr>
<tr>
<td>GHz</td>
<td>giga- $(10^9)$ cycles per second</td>
</tr>
<tr>
<td>GPGPU</td>
<td>general-purpose computing on graphics processing units</td>
</tr>
<tr>
<td>GPU</td>
<td>graphics processing unit</td>
</tr>
<tr>
<td>HPC</td>
<td>high performance computing</td>
</tr>
<tr>
<td>ILP</td>
<td>instruction-level parallelism</td>
</tr>
<tr>
<td>INS</td>
<td>incompressible Navier Stokes</td>
</tr>
<tr>
<td>ISA</td>
<td>instruction set architecture</td>
</tr>
<tr>
<td>KB</td>
<td>kilo- $(10^3)$ byte</td>
</tr>
<tr>
<td>LDC</td>
<td>lid-driven cavity (a standard computational fluid dynamics benchmark case)</td>
</tr>
<tr>
<td>MB</td>
<td>mega- $(10^6)$ bytes</td>
</tr>
<tr>
<td>MHz</td>
<td>mega- $(10^6)$ cycles per second</td>
</tr>
<tr>
<td>MIMD</td>
<td>multiple-instruction multiple-data</td>
</tr>
<tr>
<td>MMS</td>
<td>method of manufactured solutions</td>
</tr>
<tr>
<td>PCIe</td>
<td>Peripheral Component Interconnect Express (an expansion bus standard)</td>
</tr>
<tr>
<td>(D)RAM</td>
<td>(dynamic) random access memory</td>
</tr>
<tr>
<td>SIMD</td>
<td>single-instruction multiple-data</td>
</tr>
<tr>
<td>SMT</td>
<td>simultaneous multi-threading</td>
</tr>
<tr>
<td>SOA</td>
<td>struct-of-array</td>
</tr>
<tr>
<td>SPMD</td>
<td>single-program multiple-data</td>
</tr>
<tr>
<td>TLP</td>
<td>thread-level parallelism</td>
</tr>
</tbody>
</table>
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 emerging 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. 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 approach to 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 application programming interface (API) that will appear familiar to any programmer who has used OpenMP, making it straightforward for domain scientists to adopt and suggesting it could become a long-term standard. 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 [1]. SENSEI is designed to solve the compressible Euler and Navier-Stokes equations in two and three dimensions using a variety of time integration schemes and 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 test code, which is discussed in more detail in Chapter 2, was derived from a preexisting Fortran finite-difference code written to solve the incompressible Navier-Stokes (INS) 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.

This thesis focuses on the study of directive-based programming techniques (OpenACC) and parallel computational platforms applied to stencil algorithms, which are an essential component of finite-difference and finite-volume based CFD codes. A stencil algorithm is a type of computation on a grid of numerical data in which calculations are performed locally on each grid element using information from neighboring elements. A basic example is illustrated by Figure 1 showing a finite-difference stencil used to calculate the x and y derivatives at a single grid node—the resulting derivative values correspond to the red node at the center of the stencil, but data from the green nodes is used in the calculation. It is usually possible to parallelize stencil algorithms because the calculations at each grid element can be performed independently.
While there are many other algorithms relevant to CFD applications, particularly sparse linear-algebra operations, these can often be encapsulated by library routines and thus their implementations are of less immediate concern to domain scientists. Much of this work is therefore dedicated to exploring the process of applying OpenACC to stencil algorithms in a CFD code, describing any alterations that are necessary in order to run efficiently on the GPU and generally attempting to determine if OpenACC, in its present form, is practical for complex, real-world CFD. Because of the particular language of interest (Fortran 03/08), any constraints in the API that restrict object-oriented features of modern Fortran are also noted (see Chapter 4). In Chapter 4.2, some optimization techniques for OpenACC code are examined which are found to be effective for NVIDIA GPUs, but these methods involve tuning parameters for specific platforms (and thus are not necessarily optimal on different GPU architectures). The end result of the entire procedure is the successful creation of a single Fortran source code that, through a combination of OpenACC and OpenMP directives, can be compiled for multiple CPU and GPU platforms. This has the major advantage over applications written in GPU-specialized languages in that only a single-language source code needs to be maintained, and the code can be easily extended to any new architecture supported by an OpenACC compiler.

As a counterpoint to these directive-based techniques, Chapter 5 examines low-level, hardware-aware programming methods for x86 CPUs. This chapter uses a C++ version

$$\frac{\partial u_{i,j}}{\partial x} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}$$

$$\frac{\partial u_{i,j}}{\partial y} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}$$

Figure 1. A simple 2D finite-difference stencil. Derivatives at the red node are numerically approximated using information from the surrounding “stencil” of green nodes.
of the test CFD code that exactly duplicates the algorithm used in the Fortran version in order to analyze the process (and benefits) of optimizing the code exclusively for particular platforms (in this case, Intel Xeon CPUs based on the Nehalem and Sandy Bridge microarchitectures). Techniques such as manual vectorization with SIMD intrinsic functions, loop-blocking and software prefetch are used to create a version that performs very well on the Xeon platforms but is not generally portable. This version is compared with the cross-platform Fortran code to explore the performance improvements possible with programming methods tailored to a specific problem and architecture. An interesting phenomenon is also revealed through performance analyses of both the C++ and Fortran versions, in which the execution speed of the solver degrades at specific, regular dimensions of the solution data structure (i.e., grid sizes), sometimes by as much as 40%. Discussed in detail in Chapter 5.2, the problem is attributed to cache interactions (particularly conflict misses) which effectively reduce the bandwidth of the processor and can potentially have ramifications for similar structured-grid CFD codes on platforms using hardware-managed caches. This issue can be mitigated by modifying the data structure with padding, but at the cost of increased code complexity. By contrast, the GPU platforms, which use software managed caches, are free from these complications.

In Chapter 6 the performance of the test code is evaluated on several contemporary CPU and GPU platforms representing hardware readily available in the HPC domain, with the well-known lid-driven cavity (LDC) flow employed as a reference benchmark. Results are presented for three main cases: the platform-specific C++ (which runs only on the CPU), as well as the Fortran code compiled with OpenMP (CPU) and OpenACC (GPU).

1.1 Related Work

Due to the prevalence of stencil-based numerical algorithms in various fields of scientific computing, there has been significant academic interest in stencil-code performance and optimization for many years. With the introduction of practical GPU-based computing hardware along with general-purpose GPU programming languages, this interest has naturally extended to the new platforms because they offer the promise of
significant performance improvement for data-parallel algorithms. Partly inspiring the research in this thesis, there are already numerous works exploring the use of GPU hardware for finite-difference and finite-volume algorithms, including successful transitions of full-scale, working CFD codes to GPU platforms. Although many of these examples utilize low-level, GPU-specific languages (e.g. CUDA C) and focus on optimizing a specific application code for a given GPU architecture, they do show what can be achieved with practical CFD applications.

Of particular interest is work by Brandvik and Pullan [2] describing a software framework called SBLOCK, which uses automatic source code generation to expedite writing and maintaining stencil based algorithms on dissimilar platforms. Like OpenACC, SBLOCK aims to express an algorithm with a single code base and transparently generate platform-specific code for various architectures; however the framework is domain-specific and specialized for finite-difference method (FDM) and finite-volume method (FVM) stencils. SBLOCK uses the Python-based templating system Cheetah to generate C/CUDA C code based on templates that are specific to each architecture (e.g., a unique template is created for generating GPU code and another for generating CPU code). Unlike a directive-based API, SBLOCK cannot be applied to existing source codes in languages such as Fortran but rather the stencil portions of the application must be ported into a Python meta-language that can be interpreted by the template system. Brandvik and Pullan demonstrate the utility of SBLOCK by porting an older Fortran 77 finite-volume code (TBLOCK) into the new software framework and then generating computational kernels for both the CPU and GPU from the same high-level source. The newly generated code becomes part of the Turbostream solver, a compressible-Navier-Stokes FVM code for simulating flow in turbomachinery. It is significant to note that both TBLOCK and Turbostream are multi-block, structured grid, FVM codes, which are very similar in function to SENSEI.
Figure 2. A 2D multi-block structured grid around a turbine-blade. This example was used for the Turbostream solver by Brandvik and Pullan, but is identical to the class of grids used with the target SENSEI code described in the introduction. Reproduced from T. Brandvik and G. Pullan, "SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms,'' in 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), 2010. Used under fair use, 2014.

The authors observe that SBLOCK achieves approximately the same performance on the CPU as the original Fortran code while achieving a nearly 10-fold speedup on the GPU. A significant limitation of SBLOCK relative to general-purpose frameworks like OpenACC is that "non-stencil" kernels (such as complex boundary conditions) and other operations which have not been integrated into the template system must be manually written in the native language of the target platform (e.g., C or CUDA C).

Other notable cases illustrating the performance advantages of GPUs for practical CFD applications include a paper by Tolke and Krafczyk [3], which presents an optimized CUDA C implementation of a 3D Lattice-Boltzman code developed for computing flows through arbitrary porous media. The authors report speedups of nearly two orders of magnitude relative to their original CPU implementation, running non-trivial cases with this complex application. Also of note is work by Cohen and Molemaker [4] that describes the process of manually porting an existing FVM code from Fortran to CUDA C. This code solves the Boussinesq approximation of the
incompressible Navier-Stokes equations using pressure-projection and multigrid techniques. In this case, the authors choose to compare performance between contemporary CPU and GPU systems of equivalent cost—thus, they observe their CUDA C implementation to run approximately 8-times faster on a Quadro FX5800 than the Fortran version running on an 8-core Xeon E5420. Similar to the results obtained in this paper, Cohen and Molemaker also notice that the GPU performs relatively poorly on the smallest grid sizes, only attaining significant speedups as the data sets became larger.

In certain instances involving very complex and evolving CFD applications, it might be seen as more practical to incrementally transition to a GPU platform by offloading only a few computationally expensive operations while leaving the remainder unmodified. This could potentially simplify maintenance and reduce the likelihood of introducing errors into verified software, but it also has the significant disadvantage of requiring frequent data transfers between the CPU and GPU memory spaces. The issue is discussed in detail by Jespersen [5] describing work on the OVERFLOW code, which is a well-known overset grid finite-difference solver for compressible flows developed at NASA. The scope of the OVERFLOW code base precluded a direct port of the entire application, so instead it was decided to focus on the general linear-solver algorithm which profiling had revealed to consume approximately 80% of code runtime. The author states that the linear solver was first modified to be more efficient on the GPU by switching from a combined SSOR-Jacobi algorithm to a new Jacobi implementation more suitable to the massively parallel hardware, and then re-written into CUDA C. Significantly, Jespersen notes that trials with 32-bit arithmetic produced no loss of solution accuracy relative to 64-bit arithmetic (the solver converged in the same number of iterations) but led to noticeably improved runtimes on both the GPU and CPU, leading the developers to add a general option to the code permitting users to use 32-bit floating-point arithmetic for all computations. This mirrors some observations made in this thesis regarding single-precision arithmetic, which was shown to yield significant performance increases on the GPU platforms tested (see Chapter 4.3). Even when using 64-bit arithmetic, the GPU version of the OVERFLOW linear solver outperformed the best CPU implementation by up to 2.5-fold; however, the improvement in wall-clock time for the complete OVERFLOW application was only about 25-30% due at least in part to the
cost of data transactions between the GPU and CPU. Observations such as this provide further impetus to run stencil portions of a CFD code on the GPU, even when using implicit methods, because all the solution data could then remain on the GPU and the overhead of CPU-GPU data transfer would be eliminated.

Because many stencil algorithms employed in CFD utilize very large data sets, data movement (rather than floating-point arithmetic) frequently limits the performance on modern HPC hardware. Effectively using a given processor’s cache hierarchy to take advantage of data re-use (locality) in a code is usually essential for peak performance, but this non-trivial task becomes even more challenging when attempting to design multi-platform software. The problem has been approached from various angles, more recently through code-generating and auto-tuning methodologies which have been shown in some instances to produce dramatic increases in performance (Figure 3). It was observed by Datta et al. [6] in their work on stencil auto-tuning methods that the difference between “naïve” and “tuned” implementations of their test finite-difference algorithm could be as large as 5.6-fold on certain architectures, with no changes to the underlying numerical algorithm but simply through adjusting implementation parameters (e.g., cache-block dimensions, software pre-fetch instructions, etc.). The size of the parameter space used for this optimization was very large, and the authors were unable to develop a heuristic to circumvent testing most cases. They also noted the wide range of approaches to software-managed caching conceivable on GPUs, eventually settling on a novel circular queue method which they believed was optimal for their finite-difference algorithms. In another work investigating optimizations for finite-difference methods, Livesey [7] observed high sensitivity of CUDA applications to certain low-level programming details such as thread-block dimensions, which are noted to have a significant effect on utilization of hardware resources such as warp occupancy, register pressure and shared memory allocation and bandwidth. Furthermore, the optimal kernel launch configuration was found to be inconsistent across the two generations of CUDA architectures tested, meaning that code would have to be re-tuned for each version. Such observations raised obvious questions about how well the PGI OpenACC compiler would be able to automatically optimize the test application in this paper, motivating the investigations in Chapters 4.2 and 4.3 into the benefits of manual optimization of OpenACC code.
Figure 3. Various implementation techniques and optimizations applied to a finite-difference stencil algorithm by Datta, et al. Results plotted as double-precision GFLOPS. The differing effectiveness of the implementations on each platform illustrates the difficulty in establishing cross-platform heuristics and generating optimal code without architecture specific tuning. Reproduced from 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. Used under fair use, 2014.

Compared to general-purpose GPU (GPGPU) programming using languages such as CUDA C, the topic of directive-based GPU programming is a more recent development that has amassed a relatively smaller body of pertinent literature. Interest is growing, however, as directive based APIs mature and compiler vendors continue to develop improved implementations of the standards. One work that deals directly with the application of directive-based APIs to numerical algorithms is a paper by Reyes et al. [8] that surveys three programming models for GPUs: hiCUDA, PGI Accelerator and the authors’ own implementation of OpenACC called accULL. The PGI Accelerator tested by Reyes et al. actually represents an earlier version (12.2) of the PGI compiler used in this thesis for compiling OpenACC code, however the authors’ make use of the PGI Accelerator syntax instead of the OpenACC standard, which differ slightly in the naming of certain clauses and directives. The three programming schemes were evaluated by implementing a variety of algorithms designed to represent common paradigms...
encountered in scientific computing, including common linear algebra operations such as LU factorization as well as numerical benchmarks from the Rodinia benchmark suite. It was observed that while none of the directive schemes were able to match the performance of a native CUDA C implementation, they did achieve very good performance in straightforward nested-loop algorithms; for example, the directive-based schemes attained 60-75\% of the performance of the native CUDA on the HotSpot benchmark, which is a 2D thermal simulation for microprocessor heat-dissipation. The paper also discusses some other emerging directive-based standards that offer their own variants of accelerator support but are not yet implemented by any major compiler (e.g., OpenMP 4.0). While informative, the work by Reyes et al. suffers slightly from the rapid pace of development of directive-based programming standards; for instance, hiCUDA was already considered somewhat obsolescent by the authors and predicted to be superseded by OpenACC.

Finally, it should be noted that this thesis is closely related to a prior work by Pickering et al. titled Directive-Based GPU Programming for Computational Fluid Dynamics [9], which was presented at the 2014 AIAA SciTech conference. There is a great deal of overlap between the two as they both primarily stem from the same research project, but the conference paper includes some additional results from studies performed by contributing authors on multi-GPU scaling and alternate GPU platforms, while this thesis goes into much greater depth with respect to low-level programming for x86 CPUs. Where relevant, the work of Ref. [9] is cited in this thesis, and the interested reader is referred to the original source for more detail.

1.2 Parallelism in Computational Hardware

Parallel computation is a method of reducing the time required to complete a computational task by breaking it into smaller operations that can be performed simultaneously. Parallelism in various guises is employed extensively in all parts of modern HPC hardware, ranging from bit and instruction-level parallelism within arithmetic logic units (ALUs) to task parallelism on massively-threaded multi-node compute clusters. In recent years, as clock frequency scaling has encountered physical
limits, parallelism has become one of the main drivers of computational performance growth in HPC, and understanding the types of parallelism available in an algorithm and how best to take advantage of them on a given platform is essential to writing efficient CFD applications. This chapter briefly describes some of the common paradigms used in modern computers—for a more thorough and detailed discussion of parallel architectures, two excellent resources are provided by Hennessey and Patterson [10, 11].

1.2.1 Instruction-Level Parallelism

One of the most common forms of concurrent execution in computer hardware is known as instruction-level parallelism (ILP), which refers to executing multiple instructions at once from a sequential instruction stream. The use of ILP was pioneered in the earliest transistor machines of the 1960s and became endemic by the following decade; today, nearly all processors make use of ILP [12]. A standard method of implementing ILP involves recognizing that many of the instructions employed in a processor can be broken into simpler operations that can be performed in independent hardware elements, thus permitting execution to begin on an instruction before the previous ones have completed. Overlapping the execution of multiple instructions in this manner is known as pipelining, and the number of operations or stages each instruction is decomposed into is referred to as the pipeline depth [10]. Most modern hardware has pipelines that permit several instructions to be processing simultaneously, enabling a new instruction to be issued and to complete every clock cycle even if the time each takes to complete (the latency) is several cycles [13]. Frequently, redundant execution units are incorporated into a single processor core to further increase the throughput—such processors are usually capable of completing multiple instructions per clock cycle and are known as superscalar architectures [10]. Pipelined and/or superscalar hardware utilizing ILP has an important advantage over most other forms of parallelism in that it can benefit sequential algorithms which, from a mathematical perspective, may not appear to display any concurrency at all.
In general, the greatest performance benefit of superscalar hardware is observed when executing a sequence of instructions that have few or no data dependencies—that is, the input data required by an instruction does not depend on the output from any previous instructions that are less than a pipeline-depth away. In such a case where few dependencies exist the code is said to exhibit high levels of ILP and the instructions can be overlapped almost entirely, greatly increasing the throughput. When data dependencies do occur, it forces the pipelines to stall and complete all stages of the data-producing instructions before the data-dependent instruction may begin [14]. Depending on the depth of the pipeline and the latency of the instructions, frequent pipeline stalls can significantly degrade performance—deeply pipelined and superscalar architectures depend more heavily on ILP to achieve maximum performance.

Since a sequence of instructions in program order may not always expose sufficient ILP to keep the instruction pipelines full, many processors incorporate additional means of ensuring that hardware resources are optimally utilized. Some modern processors feature extensive out-of-order execution capability in which a scheduling engine analyzes the dependencies in a pre-fetched window of instructions before they are dispatched to the execution hardware [14]. When possible, the instructions within this window are re-arranged to make optimum use of processor resources as they become available. Out-of-order execution can significantly speed up many sequential algorithms, but its efficacy varies based on how much ILP is actually available to work with in the code—if an algorithm consists of a string of data-dependent operations then re-ordering may not be possible and no performance gains can be achieved. Furthermore, out-of-order execution usually requires significant hardware resources to implement, increasing the design complexity and power consumption of the processor core [10].

Another common scheme for improving the efficiency of pipelined and superscalar processors involves executing multiple instruction streams on the same processor core in the form of independent threads of execution: a technique known as simultaneous multithreading (SMT). By maintaining the context of multiple threads on the same hardware, a consistent supply of independent instructions is made available since the threads represent unrelated tasks; even if the individual threads have limited ILP, the instruction scheduler need only select the next available instruction from any one of the
threads and dispatch it to an open execution unit [11]. SMT obviates the need for complex instruction reordering in superscalar architectures (the individual threads can remain in program order) and may require less hardware complexity than out-of-order execution [10]. Processor resources are shared competitively by the threads (e.g., twice as many registers are required to maintain the context of two threads) but any resulting inefficiencies are frequently offset by better utilization of those resources. Obviously, SMT is not really a form of instruction-level parallelism but rather a form of task parallelism (described below) since it does not function with a sequential algorithm and relies on the existence of multiple instruction streams for efficiency.

1.2.2 Task Parallelism

As the name suggests, task parallelism refers to the execution of independent computational tasks concurrently and asynchronously. A processor capable of task parallelism is said to employ a multiple-instruction, multiple-data (MIMD) paradigm and may be physically implemented in a number of ways, the most common being duplication of the entire processor leading to the familiar multicore CPU [10]. In a multicore MIMD architecture each “core” is a complete processor capable of executing an autonomous instruction stream or thread, and although there is provision for communication and synchronization with the other cores it is not required for the processor to complete a task. This hardware duplication model is taken even further in modern supercomputers, which are massively task parallel machines consisting of hundreds of nodes, each a separate, independently functioning computer, all connected by a high speed network. On a smaller scale, MIMD architectures may also utilize simultaneous multithreading as described in the previous chapter to share a superscalar processor core between multiple threads. This is sometimes referred to as utilizing thread-level parallelism (TLP) in addition to instruction level parallelism on superscalar hardware [15].

The instruction sequences being executed do not necessarily need to differ between thread contexts; in HPC applications, MIMD hardware is frequently used to execute the same task on different datasets, such as when a problem domain is partitioned and the
same operations are performed on each subset. This pattern, sometimes referred to as single-program multiple-data (SPMD), is an implementation of data parallelism (performing the same task concurrently on separate data) which will be discussed in more detail in the following chapter [7]. What is important to note is that a computer capable of task parallelism is also capable of data parallelism, and this is often the manner in which MIMD hardware is employed in CFD.

1.2.3 Data Parallelism

Data parallelism refers to the concurrent execution of a single instruction on different elements of a data set. This type of parallelism is very common in CFD algorithms such as finite-difference or finite-volume methods, in which the same sequence of operations is performed at each node or cell for every index in a grid. Data parallelism can be viewed as a subset of task parallelism in which each task or thread is comprised of essentially the same instruction but operates on a separate data set, and data partitioning (or domain decomposition) is one of the primary methods employed in CFD applications to make use of multicore processors.¹

Many processors also feature hardware specialized for data-parallel operations. Such architectures are said to implement a single-instruction multiple-data (SIMD) paradigm, which differs from MIMD in that no provision is made for executing simultaneous independent instructions on a SIMD processor [11]. Instead, SIMD architectures usually employ specialized instructions that generate a single operation on multiple data elements, a procedure that resembles vector arithmetic in mathematics and gives rise to the term vector processor. When performing SIMD operations, processors will generally store the operands in special “wide” SIMD registers that can hold several data elements at once, and then manipulate the operands with identically wide execution hardware (i.e., hardware that can perform an operation on several data elements at once) [11]. In order to handle data transfers between the SIMD registers and memory, processors may use either a single pointer to access a contiguous region of memory equivalent in size to the

¹ Domain decomposition is also required for multi-node clusters and supercomputers due to their distributed memory layout.
register, or they may implement a *gather-scatter* technique in which each data element of the vector register is independently addressed (possibly by a *vector* of pointers) [11]. Clearly the latter offers greater programming flexibility, but it can be difficult to implement (not all SIMD hardware does [16, 17]) and is usually less efficient than contiguous loads and stores. For this reason, SIMD hardware tends to perform best on algorithms that exhibit large amounts of data parallelism and display predominantly contiguous memory access patterns. Many stencil algorithms do fall into this category, particularly on structured grids.

Since a SIMD architecture encodes data parallel operations in single instructions, there is usually limited provision for branching and no real capability to follow divergent code paths across the vector lanes. Obviously a data-parallel algorithm with many data-dependent branches stretches the definition, but it is not uncommon for algorithms to be “mostly” data-parallel with a few simple branches, perhaps conditional data moves or single “if” statements, and for this reason practical SIMD hardware almost always implements some form of basic vector control flow. The actual method tends to be platform-specific, but a common theme of SIMD architectures is that conditional execution is implemented through data masks and/or instruction predication instead of by manipulating the program counter with conditional jumps [11]. This means that for a two-path conditional such as an “if-else” statement a SIMD processor will execute all instructions in both branches, but it will first generate a mask from a vector comparison operation and then use the mask to prevent execution (or nullify the data) on those vector elements that do not satisfy the appropriate condition. For more complex conditional statements, or cases where the branches are very computationally expensive, this has the clear disadvantage of significantly increasing the number of instructions to be executed, and can negate the performance advantages of SIMD. Because of this, SIMD architectures often perform poorly on algorithms with many conditional statements and branches—for such algorithms, MIMD platforms are more effective.
1.3 Memory and Cache

The systems of parallelism described in Chapter 1.2 all serve to increase the computational throughput of a processor, but a critical element not addressed by these schemes is the movement of data between the processor (registers) and main memory (RAM). Many CFD applications involve operations on very large data sets that must be read and written repeatedly, and depending on the algorithm the rate at which memory transactions are performed might become a constraint on performance—in such a case a code is said to be memory bound, meaning that performance is restricted by memory bandwidth and not the arithmetic throughput of the processor [6]. The situation is aggravated in modern hardware where the arithmetic throughput is often significantly greater than the memory bandwidth, so memory access time has a disproportionate effect on overall performance. For instance, it is shown in Appendix A.1 that one of the Intel CPUs tested in this thesis is capable of performing more than 20 floating-point arithmetic operations for each floating-point value loaded from RAM (and the hardware specifications listed in Appendix A.2 show even greater discrepancy for the GPU platforms).

To attempt to alleviate some of the contention for main memory bandwidth, all of the processors (CPU and GPU) examined in this work feature a cache system. A cache is a small memory area that is physically near (or on) the processor die and is designed to provide significantly faster access time than the DRAM comprising main memory. Cache is designed to take advantage of algorithms that display locality in their data access patterns: either temporal locality (meaning that data is re-used shortly after it is accessed) or spatial locality (meaning data elements at nearby memory addresses are used together) [18]. In either case, a cache can reduce the number of times data must be transferred between the processor and main memory by storing it locally for quick re-use.

For the purposes of this thesis, caches can be broadly classified as either software-managed (directly controlled by software) or hardware-managed (controlled automatically by the hardware and transparent to software); both types are discussed further in the following chapters. All of the architectures described in Chapters 1.4 and 1.5 have hardware-managed caches, but the NVIDIA GPUs also feature an additional software-managed cache referred to as shared memory in CUDA terminology (see
Chapter 1.4). Unless otherwise noted, the term *cache* will be used to denote to hardware-managed caches, while the GPU software-managed caches will be referred to explicitly as *shared memory*.

### 1.3.1 Software-Managed Cache

A software-managed cache functions as a memory region that is independent from main memory and separately addressed. This means that the cache does not automatically store data when it is accessed, but rather data must be explicitly moved to and from the cache by software, functioning as a sort of “scratchpad” memory for data that will be repeatedly accessed. Software-managed caching can be very efficient for algorithms where the pattern of data access is known a priori because the software (i.e., programmer or compiler) can make optimal use of the cache, loading precisely the data that will be reused and avoiding evictions due to exceeding the capacity. Because they function as an independent memory system, software-managed caches are also immune to *conflict misses* and other artifacts of the mapping between cache and RAM addresses that can sometimes appear in hardware-managed schemes (see Chapter 1.3.2). This advantage is demonstrated for the *structured grid* INS code in Chapter 6, where it is noted that the NVIDIA GPUs (using software managed shared memory) display more consistent performance than the CPUs as the size of the structured grid is varied.

The primary disadvantage of software-managed caching is that some a priori knowledge of the data access pattern is *required* to use them effectively—they cannot automatically determine what data to store at runtime. This makes them practically useless for algorithms with random access patterns, such as *unstructured grid* finite-volume codes where the cells comprising a stencil cannot be deduced from the grid geometry.

### 1.3.2 Hardware-Managed Cache

Hardware-managed caches are usually designed to be transparent to software, automatically moving data in and out of the cache according to a replacement algorithm
that functions based on the pattern in which main memory is accessed.\(^1\) Only the main memory address space is exposed to software, which simplifies the task of a programmer or compiler when compared to software-managed caching \([18]\). Hardware-managed caches are more suitable than software-managed caches for algorithms in which the data access pattern is not known a priori (or random) because the data replacement scheme is determined entirely at runtime.

Hardware-managed caches typically load data in fixed size blocks of contiguous memory addresses known as *cache lines*; for example, the Intel Nehalem architecture described in Chapter 1.5 use a 64B cache line meaning that no matter the size of a data element, the surrounding 64 bytes of memory will be brought into the cache when that data is accessed \([14, 18]\). One advantage of this scheme is that it takes advantage of spatial locality—when iterating over an array, for example, all of the addresses in the cache line will generally be used in sequence. The performance of this type of memory access can be further enhanced by *prefetching* the cache lines that will be accessed in sequence, bringing them into the cache before they are actually needed. The Intel Xeon CPUs described in Chapter 1.5 all feature prefetching hardware that attempts to detect a regular access pattern and automatically prefetch the next cache line \([19]\). Additionally, these processors provide instructions that allow software to selectively prefetch addresses, which can be useful for access patterns that are too irregular for the hardware system to predict \([19]\).

When the CPU requests a memory transaction, the hardware must be able to efficiently search the entries in the cache to determine if the data is already present. If it is, then a cache *hit* occurs and the data entry can be immediately provided to the processor, while the alternative (called a *miss*) requires a comparatively slow access of main memory. An integral part of this search process is the mapping between entries in the cache and physical memory addresses—the speed at which the entries are searched has a significant effect on the cache access time. A common implementation (used in all current Xeon CPUs) is known as a *set-associative* cache, which partitions the cache into

\(^1\) Cache replacement policy forms an extensive subject that is well beyond the scope of this thesis. Often CPUs use some variation of the well-known Least Recently Used (LRU) paradigm, which tracks the temporal order in which each data entry was last accessed and, when filled to capacity, evicts the entry accessed least recently \([18]\).
equal sized regions known as a *sets* and then maps each cache line to a set—a given cache line in memory can only be stored in exactly one of the sets [18]. The number of cache lines that can simultaneously fit into a set is referred to as the *associativity* of the cache: a cache that stores $N$ lines per set is referred to as $N$-way set associative [18]. Data in a set associative cache is mapped to memory by partitioning the memory address into three parts as shown in Figure 4: the *tag*, *index* and *offset*. The *offset* bits are used to address an individual byte within a given cache line, while the remaining bits (comprising the *tag* and *index*) can effectively represent any cache line in main memory.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4. Division of a 64-bit memory address into the tag, index and offset fields needed for a 256KB 8-way associative cache using a 64B cache line. This is similar to the L2-cache found in the Intel Nehalem and Sandy Bridge microarchitectures [14].*

When the processor generates a memory reference, the *index* bits are first used to determine which set the cache line can reside in [18]. The *tag* bits of each of the $N$ entries in that set are then compared to the memory reference to determine if a cache hit or miss has occurred. This avoids the need to search the entire cache for a data entry, and if a cache line must be evicted to make room for a new entry it is only necessary to apply the replacement algorithm to the $N$ entries in the corresponding set. The example shown above in Figure 4 is analogous to the L2-caches of the Nehalem and Sandy Bridge microarchitectures, which are 256KB in size and 8-way associative. In this particular case, the 64B cache line requires a 6-bit offset, then a 9-bit index is used to represent each of the 512 sets, and the remaining 49 bits become the tag.

Hardware-managed caches can miss for a variety of reasons, but the two with most direct relevance to the INS code are *capacity* and *conflict* misses. Capacity misses occur simply because the cache becomes over-filled and begins evicting entries before they are re-used; this commonly occurs in a stencil code because the data set being iterated over is too large for the cache. One method of mitigating capacity misses is known as “loop blocking” which involves looping over smaller sub-regions of the domain that are chosen
such that the data will fit into the cache. The efficacy of loop blocking for the INS code is analyzed in Chapter 5.2. Conflict misses are notoriously more difficult to predict and manage, and can be a significant problem for structured-grid stencil codes [6, 7]. Conflict misses occur only in set-associative caches when memory transactions are made at addresses separated by a number of cache lines that is also multiple of the number of sets in the cache [7]; for example, if data is accessed at addresses separated by 32KB = 2^{15} \text{ bytes} in the example in Figure 4 (equivalent to 512 64B cache lines) then the addresses will be equivalent for the first 15 bits. This means that the index fields will be identical, so the data entries will all map to the same set. If more memory accesses of this type are made than the associativity of the cache (in this case, eight) then cache lines will be evicted even though the cache is not yet filled to capacity. This scenario is entirely plausible in structured-grid stencil codes\(^1\) because if the grid dimensions are appropriate then the non-unit-stride stencil accesses can be separated by the appropriate distance in memory to generate conflict misses on every iteration. The incidence of conflict misses in the INS code and strategies for reducing them are analyzed further in Chapter 5.2.

1.4 GPU Hardware and Programming

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 operations (or threads), which is two orders of magnitude greater than contemporary CPUs [11]. 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

\(^1\) Conflict misses can be even worse with multiple degrees of freedom arranged in an “array-of-struct” layout, as described in Chapter 3.2. Livesey [7] discusses the importance of address alignment for multi-DOF data structures.
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 [20]). This means the structure of GPGPU programming languages differs from languages traditionally used in CFD, such as C and Fortran, which were originally designed for serial hardware.

Current generation GPUs are strictly coprocessors and require a host CPU to control their operation [20, 21]. In most heterogeneous computing systems, the CPU and GPU memory systems are physically separate and must communicate via data transfers across a PCIe (Peripheral Component Interconnect Express) connection, so GPGPU programming models include functions for explicitly managing data movement between the host (CPU) and device (GPU). 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 [21, 22]. This is an order of magnitude lower than the RAM bandwidths typically seen in the servers that are often used for 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 CPU.

1.4.1 NVIDIA Fermi

The NVIDIA Fermi architecture is a graphics processor intended for both image processing and general-purpose computation, and is notable for prioritizing HPC applications with features such as high double-precision floating-point performance. Fermi consists of up to sixteen streaming multiprocessors (SMs) on a single die, each containing thirty-two 32-bit execution units referred to as “CUDA cores” in NVIDIA jargon. The CUDA cores are used to perform concurrent operations in a manner analogous to 1024-bit wide SIMD, which means most of the floating-point instructions available on Fermi GPUs operate on 32 single-precision (32 bit) or 16 double-precision (64 bit) numbers concurrently and single-precision throughput is nominally twice that of double precision [22]. Each SM also includes 16 load-store units that can generate source or destination addresses for 16 separate data elements per clock cycle, enabling
full gather-scatter type vector addressing (although consecutive data access is still more efficient [20]).

On NVIDIA hardware the basic unit of instruction scheduling is called a “warp”, which represents a single sequence of SIMD instructions that operate on 32 lanes of execution, or “threads” per the CUDA programming model. NVIDIA refers to their architecture as “Single-Instruction, Multiple-Thread” (SIMT), which differs from SIMD in that the full width of the SIMD instructions is not directly exposed in software [20]. The hardware maintains a separate register state and instruction address counter for each thread, thus permitting programmers to write parallel code as if each thread were a fully independent sequence of scalar operations [20]. Full control flow (such as conditional branching) is possible at the thread (1/32 warp) level, but programmers should be aware of how intra-warp divergence is handled by the hardware; as noted in the CUDA Programming Guide:

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. [20]

From a programmer’s perspective, this means conditional statements that cause divergence within a warp should be used sparingly to achieve best performance.

Fermi lacks out-of-order scheduling capability and executes instruction sequences in program order, relying heavily on simultaneous multithreading to make optimal use of its pipelines [20]. Instructions are issued in order from each warp and the dual warp-schedulers in the SMs are capable of queuing and dispatching instructions for up to 48 warps, making Fermi effectively a 48-way SMT architecture. Because of this, Fermi benefits from algorithms showing large amounts of thread-level parallelism in the form of many active warps on each SM [20].

The on-chip memory hierarchy consists of a 128KB register file and 64KB combined shared-memory / L1 data cache local to each SM, as well as a 768KB unified L2 cache.
shared between all the SMs. The register file contains 32,768 32-bit entries which can be
chained together in pairs for 64-bit data types. Registers are allocated per thread context
(i.e., 1/32 of a warp) and each thread is capable of addressing up to 63 32-bit registers
with no provision for accessing registers belonging to other threads. Register spills and
inter-thread communication are handled through the L1 cache and shared-memory
respectively, which are part of a single 64KB structure split into hardware and software
managed partitions that can be configured to either 48KB/16KB or 16KB/48KB sizes. In
addition to handling register spills, the hardware managed L1 partition automatically
caches all load and store requests made to main memory. The shared L2 cache is entirely
hardware managed and like the L1 cache it also services data transfers to and from main
memory, but it is additionally employed in cases requiring data synchronization and
communication between the multiprocessors such as atomic memory operations (e.g.,
read-modify-write operations that permit two threads to safely modify a single data
value) [22].

1.4.2 NVIDIA Kepler

The NVIDIA Kepler architecture is the immediate successor to Fermi and represents
the latest generation of NVIDIA GPU at the time of this writing. Kepler features up to 15
streaming multiprocessors (now abbreviated as SMX) on a single die, each containing
192 32-bit execution units and 64 64-bit double precision floating point units [15]. This
means that the nominal ratio of single to double precision floating point throughput is
now 3:1 compared to 2:1 in Fermi, although the total number of execution units per
multiprocessor has increased significantly (the number of double-precision units has
effectively quadrupled). Kepler has also increased the number of load-store units to 32
(up from 16 in Fermi) and still maintains efficient gather-scatter addressing capability
[20].

Like Fermi, Kepler executes warp-instructions in program order, with each warp
representing 32 threads as is standard on all NVIDIA GPUs to date. The four warp-
schedulers in Kepler can now queue instructions from up to 64 warps per multiprocessor,
making Kepler analogous to a 64-way SMT architecture compared to 48-way in Fermi.
Since Kepler has significantly more execution resources per multiprocessor than Fermi,
Kepler requires more parallelism per SMX to attain optimum performance. This parallelism can take the form of additional active warps on the SMX or increased ILP within each warp (or both)—NVIDIA notes in their *Kepler Tuning Guide* that:

…Kepler GPUs can utilize ILP in place of thread/warp-level parallelism (TLP) more readily than Fermi GPUs can. Furthermore, some degree of ILP in conjunction with TLP is *required* by Kepler GPUs in order to approach peak performance, since SMX's warp scheduler issues one or two independent instructions from each of four warps per clock. [15]

The register file is double the size of the previous generation at 256KB per SMX, with 65,536 32-bit entries that can be used in pairs for 64-bit data types [20]. Each thread can address up to 255 32-bit registers, and there is now a provision for inter-thread communication via “shuffle” instructions that permit threads within a warp to exchange data without resorting to shared memory. The L1 cache and shared memory structure is the same size as before at 64KB, but it is now possible to partition the hardware/software managed sections (L1 cache and shared memory) into three possible configurations: 16KB/48KB, 32KB/32KB or 48KB/16KB. Kepler also makes available a 48KB read-only data cache that is independent from the 64KB L1 cache and shared memory structure. This read-only cache can be particularly useful for applications that exhibit significant spatial locality in their memory access pattern because (unlike on Fermi) data loads from main memory are not implicitly cached in L1—the only method of obtaining hardware managed L1 cache behavior on Kepler is for the programmer or compiler to use specific load instructions that fetch data through this read-only path. The read-only cache actually takes the place of the texture-cache\(^1\) in Fermi, being both larger and able to handle generic data access (meaning data no longer has to be accessed through texture references [15]). Kepler’s L2 cache has also been enlarged compared to its predecessor, becoming a 1536KB unified cache shared by all 15 SMXs. The L2 cache automatically services all data loads and stores [20].

---

\(^1\) NVIDIA CUDA devices permit loading data through the *texture units* for general purpose computation. The texture units are part of the graphics pipeline and provide some additional cache and computational resources specialized for graphics rendering operations [20].
Control flow on Kepler is the same as on Fermi. The fundamental unit of concurrent execution remains the warp, consisting of 32 threads per the CUDA programming model. Divergence between warps has minimal effect on performance, but divergence within a warp requires the sequential execution of each possible code path while instruction predication (deactivating individual CUDA cores) is used to ensure that the correct branch is taken for each thread. As on Fermi, it is best to avoid conditional statements that cause divergence within a warp because such branching can have an adverse effect on performance.

1.4.3 CUDA Programming Model

The proprietary CUDA C language introduced by NVIDIA is a modified form of C that is optimized for stream computing and provides a general-purpose interface to the programmable components of CUDA enabled GPUs. Due to its specialized design, the language differs notably from the standard procedural languages frequently employed in CFD. Programmers interact with the GPU through data parallel functions called kernels, each of which is divided into an abstract computational space of “thread-blocks” that are further composed of “threads” mapping to the elements of SIMD operations (i.e., each thread will be executed by a CUDA core) [23]. Unlike in a standard C function, operations coded within the kernel function are by default applied to all the threads in the grid. Threads are differentiated by their location in the computational space which is accessible through built-in variables that assign each thread a unique identity inside a unique block—this thread/block/grid indexing may be represented in one, two or three dimensions, and the sizes of the grid and blocks are determined by parameters passed to the kernel call [23]. Threads can use their unique ID to conditionally execute program statements or as a means of individually addressing data.
The significance of grouping threads into blocks is that a complete block is always scheduled on a single multiprocessor (SM, SMX), so the threads within are assured of access to the same multiprocessor resources (e.g. shared memory) and their contexts will reside on the multiprocessor at the same time. This enables threads within a block to synchronize (to prevent execution until all threads reach a specific point in the code) and to exchange data via shared memory, which is not generally possible across separate blocks as they may be scheduled on different multiprocessors and at different times. It is important to note that a CUDA programmer has no knowledge or control over the order in which thread-blocks are scheduled, a precept of a language meant to express data parallel algorithms [20, 23].

CUDA C is a relatively low-level language which has a platform-specific structure and many adjustable parameters that make it effective for writing highly tuned applications on NVIDIA GPUs. This same specialization results in CUDA functions being non-interchangeable with their C-language counterparts, and thus maintaining an application in CUDA C and plain C effectively requires two separate versions of the
Furthermore, the sensitivity of GPU performance to hardware-specific optimizations [20] makes maintaining a low-level CUDA source code challenging given the rapid pace of hardware development; it is for this reason that higher-level programming standards such as OpenACC were developed.

1.4.4 OpenACC

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) [24]. In Fortran, these directives take the form of comment-statements similar to those used in OpenMP:

```fortran
!$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 parallelism 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 [24]. 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 [24]. 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 [24]. 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 [24].

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 [24]. The actual mapping from this representation into a lower-level accelerator programming model is specific to each target platform and 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 [25]. As will be discussed further in Chapter 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 [24]. 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 [24].

### 1.5 CPU Hardware and Programming

The CPUs investigated in this thesis are all Intel Xeon processors implementing the x86-64 instruction set architecture (ISA). This selection is primarily a consequence of what hardware was readily available for testing; however, it should be noted that x86-64 processors are very common in the HPC domain and nearly ubiquitous in desktop and workstation computers, so these CPUs represent a significant share of the hardware available to computational scientists.

Most of the benchmark data in subsequent chapters was collected on two specific generations of Intel microarchitecture: Nehalem and Sandy Bridge. Like nearly all modern x86 processors these are multi-core (MIMD) and superscalar designs, with advanced out-of-order and speculative execution capability that yields very high performance on single threaded tasks [14]. These particular microarchitectures also feature 2-way simultaneous multithreading under the proprietary name *Hyper-Threading Technology* (HTT) which can enhance performance on multithreaded applications having limited ILP [14]. SMT is not common in x86 designs and is restricted to only two threads per core for HTT primarily because the out-of-order execution paradigm is generally so effective in utilizing the superscalar hardware (via ILP) that additional TLP is not required to attain optimum performance [14].

The Sandy Bridge and Nehalem microarchitectures both feature short-vector SIMD FPUs and ALUs within each core that are accessible through extensions to the x86 ISA. Nehalem implements the older *Streaming SIMD Extensions* (SSE) instruction set up to version SSE4.2, which features sixteen 128-bit wide architectural registers and corresponding SIMD instructions that permit vector floating-point operations with up to
two 64-bit or four 32-bit elements [11]. The SSE programming model has gone through several iterations, but since version SSE2 has included arithmetic and bitwise operations for nearly all standard integer and floating-point data types (Figure 6). These are true vector instructions in the sense that the encoded operation is always applied to all data elements in a vector register—there is no instruction predication or branching for individual lanes as there is on the NVIDIA GPUs. To handle conditional code, SIMD comparison instructions are provided which generate masks evaluating to bitwise true or false for each element [17, 19]. These masks are used to nullify data where appropriate to ensure each lane obtains results from the correct code branch. SSE has no provision for scatter-gather addressing, but does include instructions for shuffling the elements in a SIMD register and for packing and unpacking data between registers. This enables somewhat inefficient emulation of scatter-gather by first transferring scalar data from memory into separate registers and then combining them into a single SIMD register (the process is reversed for storing data). Contiguous loads and stores are more efficient, with best performance achieved when the contiguous data is 16-byte aligned (address is a multiple of 16).

Sandy Bridge is backwards-compatible with all variants of SSE but also adds an additional set of SIMD instructions known as the Advanced Vector Extensions (AVX). These instructions extend the SIMD registers to 256-bits wide and permit concurrent operation on four 64-bit or eight 32-bit data elements for most floating-point instructions [17]. AVX also adds a new three-operand instruction format which (uniquely for x86) allows the destination operand to be distinct from either of the sources—in certain cases this might reduce register pressure and yield improved performance, even in scalar applications [26]. Unlike SSE, AVX lacks full-width vector support for many integer arithmetic operations, limiting them to the lower 128-bits. It is also significant to note that while the performance of most floating point operations is doubled, there are certain low-throughput, high-latency FP-instructions (such as square-root and divide) that do not see an improvement with AVX because the hardware implementation of these operations has not been extended from 128 to 256 bits [13]. In most other respects, the AVX instruction set behaves similarly to SSE: conditional code must be handled through data masks and there are no scatter-gather addressing instructions, but there are myriad shuffle and permute instructions to compensate. AVX also adds conditional masking of vector load instructions and a memory-to-register broadcast instruction (which loads a single scalar into all elements of a SIMD register) to further aid non-uniform access patterns [11]. Contiguous data access is still preferred for vector operations, with optimum results obtained when the data is aligned to 32 byte addresses (vs 16 byte for SSE).

Sandy Bridge and Nehalem have very similar on-chip memory hierarchies consisting of three levels of cache. On both architectures this is predominantly hardware managed, although there are software prefetch instructions to selectively bring data into cache as well as SSE and AVX non-temporal store instructions that permit data writes to bypass the cache [19]. Each core contains a private 32KB L1D (data) cache and a private 256KB unified L2 cache, both of which are 8-way associative and have a non-inclusive/non-exclusive relationship. The last level (L3) is 16-way associative, shared by all cores and is inclusive with L1 and L2—all three levels use the same 64B cache line. The total size of the L3 scales with the number of cores and the processor model, but is generally at least 2MB per core for Xeon processors [27].
1.6 Programming for x86 SIMD

The x86 SIMD instructions can improve computational throughput significantly on floating-point intensive applications—Intel claims a near linear performance increase in some of their Math Kernel Library (MKL) routines [28]—but require special handling by either the programmer or the compiler to be used effectively. Common techniques for making use of the SIMD vector instructions in an application include the very low-level approach of applying the instructions directly—either through inline assembly or compiler intrinsics—or the very high-level method of relying on compiler auto-vectorization. The SIMD intrinsic functions provided by C/C++ compilers such as GCC and ICC operate on built-in vector data types that correspond to the SIMD register size, and most of the intrinsics map directly to single assembly-language instructions [19, 29]. Because of this, programming with SIMD intrinsics often requires higher coding effort than plain-C and the results are not portable (intrinsics are akin to writing assembly language). By contrast, compiler vectorization is effectively transparent to the programmer and completely portable, but it is still necessary to ensure that the structure of the algorithm and data is appropriate for SIMD-style parallelism. Efficient data access for SIMD operations often requires specific data layout and alignment, so it may be necessary to re-structure an existing application if it was not originally written with SIMD in mind [29, 30].
2 CFD Code

The CFD code investigated in this thesis solves the steady-state incompressible Navier-Stokes (INS) equations using the artificial compressibility method developed by Chorin [31]. The INS equations are a nonlinear system with an conservation of mass (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 $\rho$ be the density constant and $\nu = \mu/\rho$ be the kinematic viscosity (also assumed constant), the complete system takes the familiar form below.

\[
\frac{\partial u_j}{\partial x_j} = 0 \tag{1}
\]

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

The artificial compressibility method transforms this INS system into a coupled set of equations hyperbolic in time 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:

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

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

In Equation (3), $\beta$ 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, calculating $\beta$ using the local velocity components ($u$ and $v$), a scaling constant ($r_k$) and a reference velocity magnitude ($V_{ref}$) as shown below.

\[
\beta^2 = \max(r_k \cdot V_{ref}^2, u^2 + v^2) \tag{5}
\]
This system of 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 on a Cartesian grid). 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).

\[
\frac{1}{\beta^2} \frac{\partial p}{\partial t} + \frac{\partial u_j}{\partial x_j} - \lambda_j \Delta x_j^3 C_j \frac{\partial^4 p}{\partial x_j^4} = 0
\]  \hspace{1cm} (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. The complete numerical scheme uses approximately 130 floating point operations per grid node, including three floating point 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.
Figure 7. Illustration of a 9-point finite-difference stencil.

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 [32, 33]. 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 thesis 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 $128 \times 128$ to $8192 \times 8192$ nodes. A fully iteratively converged solution is displayed below.
Figure 8. Horizontal velocity component and streamlines for a converged solution of the LDC benchmark on a 512x512 node grid.
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 (Chapter 3.2). 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 the subsequent residual calculation loop. By “fusing” the artificial viscosity loop into the residual loop, and calculating the artificial viscosity when needed (immediately before the residual) at each node, it was possible to completely remove the artificial viscosity array and all associated data movement.

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) because each depended only on the node-local output of the previous operation. This modification resulted in an overall performance increase of approximately 2x compared to the un-optimized version (see Figure 11). It should be noted that fusing loops in this manner is not always straightforward for stencil codes because 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, with the artificial viscosity loop operating on a slightly smaller interior
region than the two others (see Figure 9). This difference in loop bounds necessitated “cleanup” loops along the boundaries for the time-step and residual stencils and slightly increased the overall 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.

![Diagram of original iteration algorithm for the INS code. Grey boxes indicate 2D operations on the rectangular domain, green boxes are 1D loops along the boundary.](image1)

**Figure 9. Original iteration algorithm for the INS code.** Grey boxes indicate 2D operations on the rectangular domain, green boxes are 1D loops along the boundary.

![Diagram of modified iteration algorithm for the INS code with artificial viscosity, time-step and residual operations fused into a single loop. Grey boxes indicate 2D operations on the rectangular domain, green boxes are 1D loops along the boundary.](image2)

**Figure 10. Modified iteration algorithm for the INS code with artificial viscosity, time-step and residual operations fused into a single loop.** Grey boxes indicate 2D operations on the rectangular domain, green boxes are 1D loops along the boundary.
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% (Figure 11). This would be similarly effective in more complex time integration algorithms that require storage of multiple solution levels, such as multistage Runge-Kutta methods.

![Figure 11. 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 Chapter 5).](image)

### 3.2 Data Structure Modification

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 12). 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 [20, 29, 19].

|----------------|------------------|------------------|----------------|------------------|------------------|----------------|------------------|------------------|

**Array-of-Struct**

|----------------|----------------|----------------|------------------|------------------|------------------|------------------|------------------|------------------|

**Struct-of-Array**

Figure 12. 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 [34].

The OpenACC code was constructed from the branch incorporating the high-level optimizations and data-structure refactoring described in Chapter 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 “!$acc data” region, and then using the present clause within the enclosed subroutines to indicate that the data was already available on the device. If the data was needed on the host between iterations (to output intermediate solutions, for example) the “!$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 described in Chapter 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 CFD problems.

By design, applying the OpenACC API to an existing code should be possible without modification of the original source, however there were two instances where restrictions imposed by the PGI compiler forced some refactoring in the INS code. 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 [24]. The original INS code contained a particular subroutine that encapsulated several operations common to the boundary and interior schemes, such as the calculation of the artificial viscosity. The compiler was not able to inline this function in spite of specific requests to do so via compilation flags, so ultimately it had to be inlined by hand, leading to some repeated code where it was previously called. In all other cases the PGI compiler successfully inlined the requisite functions, but this particular case illustrates a potential difficulty that could arise in more complicated codes with many large, complex subroutines. Furthermore, the restriction on device function calls effectively disables function pointers, which are an extremely useful feature of Fortran 2003/2008.

Another inconvenience for modern Fortran programs is the prohibition of allocatables as members of derived types—within an accelerator region, the PGI compiler does not permit accessing allocatable arrays that are part of a user-defined type, although it does allow arrays to consist of user-defined types (Figure 13). 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 best for contiguous memory access on the GPU. In the relatively
simple INS code this was only a minor inconvenience, but in the full scale research code derived types are used extensively to manage allocated data structures, so applying OpenACC would entail significant refactoring.

<table>
<thead>
<tr>
<th>!AOS: ! Permitted in accelerator ! regions</th>
<th>!SOA: ! Not permitted in ! accelerator regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Velocity(i,j)%V1 = 0.0_dp</td>
<td>Velocity%V1(i,j) = 0.0_dp</td>
</tr>
<tr>
<td>Velocity(i,j)%V2 = 0.0_dp</td>
<td>Velocity%V2(i,j) = 0.0_dp</td>
</tr>
</tbody>
</table>

Figure 13. Example AOS and SOA derived types using Fortran 03 syntax. Allocated arrays of derived types are permitted (left), but types containing allocated arrays are not (right).

4.1 OpenACC Performance Testing

In this chapter 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 at the time of the tests represented the latest stable release. Version 13.6 was only capable of generating accelerator code for CUDA enabled devices, which limited the selection of hardware to that made by NVIDIA; Chapter 4.4 presents additional results from Pickering et al. [9] evaluating a newly released version of the compiler (PGI 14.1) capable of compiling for AMD devices. 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 (i.e., Tesla, Fermi and Kepler architectures).

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)(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 14, 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 chapter, this performance can in some cases be improved upon through tunable parameters that are part of the OpenACC API (Figure 14, green lines).

![Figure 14. Double-precision performance of LDC benchmark on NVIDIA C2075 (left) and K20c (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.](image)

### 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 noted in Chapter 1.4.4 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 [15, 20].

Figure 15. Mapping of CUDA thread-blocks to computational domain. 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. Adapted from NVIDIA Corporation, CUDA C Programming Guide, Version 5.5, 2013. Used under fair use, 2014.

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. 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. This procedure was then used to generate a map of the performance across the full range of thread-block dimensions, as displayed in Figure 16. 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 [20].

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 16, 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%. 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 16. 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 [34].

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 17, 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 reinforces the notion that heuristics alone are insufficient for generating optimum code on the GPU and illustrates the difficulty of tuning for multiple platforms.

![Figure 17. Single-precision performance vs. thread-block dimensions for a fixed-size LDC benchmark. (4097x4097, 50 million DOF). NVIDIA C2075 (left) and K20c (right)](image)

The speedups observed with single precision arithmetic were less impressive than expected based on the theoretical throughputs given in Table A1 of the Appendix. 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 summary of the results is displayed in the tables below.

<table>
<thead>
<tr>
<th>Tesla C2075 (Fermi)</th>
<th>Default Block Size (GFLOPS)</th>
<th>Optimal Block Size (GFLOPS)</th>
<th>Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Precision (GFLOPS)</td>
<td>44.6</td>
<td>47.9</td>
<td>7.4%</td>
</tr>
<tr>
<td>Single Precision (GFLOPS)</td>
<td>64.5</td>
<td>75.4</td>
<td>16.9%</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>44.6%</td>
<td>57.4%</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. OpenACC thread-block optimization results for Tesla C2075.

<table>
<thead>
<tr>
<th>Tesla K20c (Kepler)</th>
<th>Default Block Size (GFLOPS)</th>
<th>Optimal Block Size (GFLOPS)</th>
<th>Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Precision (GFLOPS)</td>
<td>68.5</td>
<td>90.6</td>
<td>32.3%</td>
</tr>
<tr>
<td>Single Precision (GFLOPS)</td>
<td>149.2</td>
<td>153.5</td>
<td>2.9%</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>117.8%</td>
<td>69.4%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. OpenACC thread-block optimization results for Tesla K20c.

<table>
<thead>
<tr>
<th>Optimal Thread-block Dimension</th>
<th>Double Precision</th>
<th>Single Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2075 (Fermi)</td>
<td>16x4 (16x8)*</td>
<td>16x6</td>
</tr>
<tr>
<td>K20c (Kepler)</td>
<td>16x8</td>
<td>32x4</td>
</tr>
</tbody>
</table>

Table 3. Optimal thread-block dimensions obtained for different GPU architectures.
*Note that for the Fermi architecture, two block sizes were optimal for double precision.

4.4 Multi-GPU Scaling

In Pickering et al. [9], a multi-GPU version of the code was constructed that used multiple CPU-threads to manage separate GPUs. Domain decomposition was employed
to divide the solution data amongst the GPUs, and only boundary data was exchanged on each iteration. This technique minimized the overhead of PCIe data transfers and resulted in nearly linear performance scaling on up to 4 GPUs (the maximum available in the test rig), suggesting that a similar implementation using MPI might also provide good scaling on a GPU cluster. A newer version of the PGI compiler (14.1) was used for these tests capable of compiling for AMD hardware. Surprisingly, a top-end AMD 7000 series GPU (7970 or single die of 7990) was seen to produce the best single-GPU performance of any card tested, exceeding even the best Kepler compute cards. Using both dies of the 7990 (effectively two independent GPUs on a single card) produced the highest overall performance of any platform tested. The results are reproduced below.


5 Performance Tuning on x86 CPUs

This chapter investigates hardware aware programming techniques and optimizations tailored to the Intel Nehalem and Sandy Bridge microarchitectures, with the objective of determining how much additional performance can be attained via low-level tuning dedicated to a specific problem and architecture. Initially the purpose was simply to create a strong reference version of the INS code against which the GPU version could be measured, but as the project progressed it was found that this task was much more involved (and interesting) than expected. A variety of techniques for enhancing the performance of the INS stencil algorithm are examined below, including loop blocking, software prefetching and SIMD vectorization.

Although the original INS code was written in Fortran, the language chosen for the CPU optimized version was C++. This decision was made primarily because C and C++ (unlike Fortran) provide the low-level SIMD intrinsic functions that enable “manual” vectorization without resorting to assembly language. C++ is also an object oriented language that has useful high-level features such as operator overloading and templates, making it preferable to C when creating complicated data structures. The compiler used for all optimized CPU cases was GCC 4.7.2, with multithreading handled via the included OpenMP implementation. The only compiler flags set were the optimization level “-O3” along with architecture specific flags when a particular instruction set was needed (e.g., “-march=corei7-avx” for AVX).

5.1 SIMD Vectorization

Vectorization of the INS code was accomplished manually via x86 SIMD intrinsic functions, which are available in the GCC compiler under the headers “emmintrin.h” (SSE2 or AVX-128, depending on compiler flags) and “immintrin.h” (AVX-256). Rather than code the intrinsics directly, C++ container classes were created to encapsulate the built-in vector data types, making it possible to use operator overloading to express SIMD arithmetic with normal mathematical operators. Simple functors were used to transfer data between the vector classes and the solution data array (also a custom class), and functions were defined between scalar floating point values and the SIMD
classes such that the scalars were automatically extended to vector operands. The end result was SIMD C++ code which almost exactly resembled ordinary arithmetic using built-in scalar data types, and facilitated straightforward conversion of the INS code.

As on the GPU, the code was written so that each SIMD lane corresponds to a single grid node. This meant that the innermost loops had to be unrolled by a stride equal to the vector length (e.g., stride 4 for AVX-256 double precision) and in cases where the stride did not evenly divide the size of the domain the inner-loop would terminate in scalar cleanup iterations. This manual SIMD coding slightly increased the complexity of the loop structure, but more significantly it complicated reduction operations such as summation of the iterative residual values. The main problem was that the OpenMP reduction clause, which was used in the original scalar code, cannot handle overloaded operators and user-defined classes, so a manual implementation of the multithreaded reduction was required. Reductions were constructed using a technique that mirrors the standard OpenMP implementation: each thread performs the reduction operation into a private SIMD variable, a horizontal vector operation is applied when the loop completes, and then atomic operations are used to obtain the global scalar value. Because of this coding complexity, SIMD operations were employed only on the interior of the domain and not in the boundary routines. This had only minor effect on application performance since the majority of the runtime is spent executing the interior scheme, and furthermore two of the four boundary loops access memory in the non-unit stride dimension, which would have required a scatter-gather implementation that may have degraded performance.

The solution data was stored in a column-major SOA format that allowed all SIMD memory access to be contiguous, however not all data transfers could be made 16 or 32 byte aligned due to the stencil algorithm. As illustrated in Figure 19, the unit-stride distance between stencil nodes forces at least some of the load operations to be misaligned, so unaligned load and store intrinsics were used for all data transfers. On Nehalem and later Intel architectures the use of unaligned memory instructions on aligned addresses does not have a performance penalty, while attempting to access misaligned data with an aligned instruction will always generate a fault, so it is logical to use the unaligned instructions for all memory access in x86 SIMD code.
Figure 19. SIMD data loads for a 5-point stencil using SSE2. Each register loads two contiguous double precision floating point values (16 bytes) needed for two independent stencils—the inner-loop is unrolled by stride 2, and the data is stored in a column major array. Clearly, the \((i, j)\) load and the \((i \pm 1, j)\) loads cannot both be 16 byte aligned because their starting addresses differ by 8 bytes.

The ideal speedup over the scalar version of the code would be 2x for SSE2/AVX-128 and up to 4x for AVX-256 when using double precision. On both Nehalem and Sandy Bridge the INS code enhanced with SSE2 displays good performance, achieving speedups of 50-60% over the scalar version (see Figure 20 and Figure 21). Even in the cases where performance was being limited by cache effects (a problem discussed further in Chapter 5.2) it appears that the SSE2 code has a significant advantage, which may be due to the fact that SIMD loads more data simultaneously into registers and thus reduces the penalty of inefficient cache use. On Sandy Bridge, setting the compiler to generate AVX-128 instructions instead of SSE2 yielded an average of 5% better performance than SSE2, which is likely due to the 3-operand AVX instructions reducing register pressure.
Figure 20. SIMD instructions applied to C++ INS code on Sandy Bridge. Dual socket Xeon E5-2687W, 16 threads / 16 cores (left), 32 threads / 16 cores (right). Note that software prefetch instructions are being used, but there is no data structure padding in these cases.

Figure 21. SIMD instructions applied to C++ INS code on Nehalem. Dual socket Xeon X5560, 16 threads / 8 cores. Note that software prefetch instructions are being used, but there is no data structure padding in these cases.

Comparing the performance of 256-bit AVX to 128-bit SIMD on Sandy Bridge seems disappointing in light of the impressive speedups observed with the transition from scalar to SSE. On a 3.4 GHz (E5-2687W) workstation, there is only a 13% improvement between AVX-128 and AVX-256 when using one thread per core, and less than 10% improvement when using two threads per core. Running the same cases on a lower-
clocked E5-2660 server (2.7 GHz) yielded a marginally better 15% and 12% respectively, and in general it was observed that the performance gap between AVX-256 and AVX-128 widened as processor clock speed decreased. This could suggest the AVX-256 version is limited (at least in part) by memory bandwidth, so to test this the DRAM clock speed was varied on a low-clocked Sandy Bridge workstation and the performance of the three code versions compared. The results are displayed in Figure 22 and suggest that while the code is not completely memory bound it is near the point where memory effects start to become significant. Certain parts of the algorithm, such as the pressure rescaling function, are definitely memory bound, and as the ratio of CPU speed to memory bandwidth increases these functions exert more influence on the overall application performance.

![Figure 22. Effect of memory clock speed on INS code using three types of SIMD instructions. Xeon E5-2650 @ 2.4 GHz, 16 threads / 8 cores.](image)

Main memory bandwidth cannot be the only factor limiting the performance of the AVX-256 code because a degree of scaling is still observed with changes in core clock frequency. The peak memory bandwidths of the 2.7 GHz and 3.4 GHz Xeons described previously are identical, however the INS code runs approximately 18.5% faster on the higher clocked CPU. This suggests that the upper performance bound on the 2.7 GHz
processor is not exclusively the result of a memory bottleneck and suggests an on-chip influence instead. One possibility is that the cache on Sandy Bridge is not able to sustain the high throughput demanded by the stencil algorithm (which is specifically blocked for the L2 size) because of the large number of misaligned 256-bit loads and stores. The cache line size on Sandy Bridge is 64B, so every misaligned 32B memory access will result in a cache-line split; according to the Intel 64 and IA-32 Architectures Optimization Reference Manual, misaligned 32B accesses will result in twice as many cache line splits as 16B accesses, all of which carry a latency penalty [19]. The manual actually recommends splitting 32B loads and stores into separate 16B transactions (chapter 11.6.2), however when this was implemented in the INS code it yielded a 1-2% performance regression, possibly resulting from increased register pressure and spills.

Another potential cause of the poor 128 to 256 bit SIMD scaling is that the code may actually be compute bound on certain high-latency, low-throughput floating point instructions that do not benefit from AVX-256, such as square root and divide. The INS code performs 130 FLOPs per grid node, three of which are divisions and two of which are square roots. These operations are relatively slow on the Sandy Bridge microarchitecture and are not pipelined, as can be seen when comparing their latency and throughput values to other operations in Table 4. While the VSQRTPD and VDIVPD instructions operate on 256-bit registers, their hardware implementations have not been extended to 256 bits and the vector instruction is actually emulated by executing two 128-bit micro-operations in sequence. The result is that while SSE2 and AVX-128 will see a 2x performance improvement over scalar code, there is no performance increase when moving from AVX-128 to AVX-256 for these instructions, so they may become a bottleneck in a 256-bit SIMD application.
(a) 256-bit packed double-precision floating point instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Latency</th>
<th>Reciprocal Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADDPD/ VSUBPD</td>
<td>addition/subtraction</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>VMULPD</td>
<td>multiplication</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>VMAXPD/ VMINPD</td>
<td>maximum/minimum</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>VDIVPD</td>
<td>division</td>
<td>33-45</td>
<td>44</td>
</tr>
<tr>
<td>VSQRTTPD</td>
<td>square root</td>
<td>45</td>
<td>44</td>
</tr>
</tbody>
</table>

(b) 128-bit packed double-precision floating point instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Latency</th>
<th>Reciprocal Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD/ SUBPD</td>
<td>addition/subtraction</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>MULPD</td>
<td>multiplication</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>MAXPD/ MINPD</td>
<td>maximum/minimum</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DIVPD</td>
<td>division</td>
<td>16-22</td>
<td>22</td>
</tr>
<tr>
<td>SQRTPD</td>
<td>square root</td>
<td>29</td>
<td>22</td>
</tr>
</tbody>
</table>

(c) 64-bit scalar double-precision floating point instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Latency</th>
<th>Reciprocal Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDSD/ SUBSD</td>
<td>addition/subtraction</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>MULSD</td>
<td>multiplication</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>MAXSD/ MINSD</td>
<td>maximum/minimum</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DIVSD</td>
<td>division</td>
<td>16-22</td>
<td>22</td>
</tr>
<tr>
<td>SQRTSD</td>
<td>square root</td>
<td>29</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 4. Latency and throughput for several floating point instructions on Sandy Bridge. (a) AVX-256 (b) AVX-128 (c) Scalar 64-bit. Note that all of these instructions are used in performance-critical sections of the INS code.

5.2 Cache-Aware Programming

A well-known technique for exploiting data locality in stencil codes is cache or loop blocking, which involves partitioning the operations on a large domain into smaller working sets that fit into cache memory. The finite-difference stencils in the 2D INS
code require a total of 19 data loads, however because the stencils of adjacent nodes overlap most of this data can be recycled through caching, ideally reducing the loads from main memory to only 3 per node (1 per DOF). For a two-dimensional structured grid loop blocking is a straightforward procedure: the hardware managed cache automatically stores data when it is accessed, so as the stencil iterates through the grid (in the unit-stride direction) it caches several “columns” of nodes. The only requirement for effective reuse is that these columns do not become so large that they spill back into main memory before the outer loop increments—thus a 2D block must be defined so that the *inner loop* bounds are appropriate for the given cache size. A range of block sizes were investigated and it was observed that blocking for the L2 (between 40-90KB per thread) seemed most effective, although the blocking scheme was only necessary on the largest grid sizes (Figure 23).

![Figure 23. Effects of cache-blocking on INS code performance. Xeon X5560, 16 threads / 8 cores.](image-url)
Preliminary versions of the optimized INS code, both with and without loop blocking, displayed uneven performance that fluctuated with the problem size. These variations were most pronounced in tests on the Nehalem microarchitecture, but were clearly present on Sandy Bridge as well. A regular pattern can be discerned which is present in tests of both the Fortran and C++ codes. As seen in Figure 24, the most extreme oscillations occur when the \( i \)-dimension of the grid is a multiple of 512, with particularly poor performance at multiples of 4096. Since the grid is a flattened 2D array of double-precision (8 byte) elements, these numbers correspond to multiples of 4KB and 32KB respectively, which equates to exactly 64 and 512 cache lines on the Intel Xeon architectures being tested. This is significant because the 8-way L1 and L2 caches have 64 and 512 sets each, so it suggests that cache effects could be responsible for the pattern. One plausible culprit would be conflict misses caused by reading stencil points that are separated in memory by a multiple of the number of cache sets—as illustrated in Figure 25(a), the distance in linear memory between stencil points that are adjacent in the non-unit stride direction is always equal to the size of the grid in the unit stride dimension, so at particular grid sizes adjacent stencil points will compete for addresses in the cache. The issue is exacerbated by having three degrees of freedom arranged in a SOA layout because there are now three separate arrays being accessed with the same aligned stencil
pattern. Since the L1 and L2 caches are only 8-way associative, three perfectly aligned 5-point stencils would likely cause premature evictions.

![Diagram](image)

Figure 25. (a) The distance in linear memory between adjacent stencil nodes is determined by the number of nodes in the unit-stride dimension. (b) Illustration of the array padding technique.

The conflict miss theory is substantiated in a study of stencil algorithms by Livesey [7], who observed a similar phenomenon in a finite-difference application having 6 equations and using the same SOA data layout as the INS code. In that study it was found that the problem could be mitigated by modifying the data allocation procedure, varying the separation between the arrays to avoid the particular address alignment that led to the conflict misses. Since the INS code actually uses a 3D array to store three DOF in two-dimensions, independently varying the starting address of each DOF in this manner was not practical, however it was found that padding the grid to avoid the sizes which generated conflict misses provided a similar effect. This padding was accomplished simply by allocating a larger grid in memory but restricting the working area to a subset as illustrated in Figure 25(b). The performance results are shown in Figure 26.

### 5.3 Simultaneous Multithreading

On Nehalem, it was observed that at certain grid sizes the performance of the INS code running one thread per core decreased, while the performance with two threads per core actually increased (Figure 26, red line). It is not certain why these specific cases ran faster on this architecture (the pattern was not observed on Sandy Bridge), but it did suggest that overall greater performance might be attained by specifically tuning the code.
for SMT. The first attempt to optimize for SMT involved using the same padding technique discussed above to automatically scale the data structure up to the next largest size at which two threads per core showed increased performance; as expected, this did produce a uniform speedup, but it also required padding nearly every grid size to a very specific set of dimensions. It was later found through trial and error that a similar effect could be achieved through the use of software-prefetch: by emitting three prefetch instructions (1 per DOF) for addresses one column ahead of the stencil in the loop block, the same performance increase was attained (Figure 26, blue line). The use of software prefetch had little effect on the non-SMT cases—performance actually degraded slightly when it was used with one thread per core on Nehalem—but was very significant when two threads per core were used, possibly suggesting the hardware-prefetcher had difficulty tracking the additional memory accesses by the second thread.

The best performance achieved on both Sandy Bridge and Nehalem came from a combination of the software-prefetch and grid-padding techniques when running two threads per core. This was especially true on Nehalem, where the combined application of the padding and prefetch optimizations saw an average of 12% higher performance than either alone. Sandy Bridge saw less effect from the grid padding technique and achieved near peak performance with just prefetch, although padding was needed to avoid cache conflict misses. Through these combined optimizations, SMT-speedups of up to 26% were observed compared to the single thread per core cases.
Figure 26. Effects of array-padding and software prefetch optimizations applied to C++ INS code. Nehalem microarchitecture, Xeon X5560, 16 threads, 8 cores.
6 Comparisons of CPU and GPU Performance

To evaluate the relative performance increase provided by OpenACC acceleration, the results from the GPU benchmarks were compared against both the Fortran INS code re-compiled for multicore x86 CPUs as well as the optimized C++ version described in the previous chapter. It is significant to note that the Fortran version represents a unified INS code base capable of running on either the CPU or GPU. It contains both OpenMP and OpenACC directives which can be switched on or off by passing the appropriate options to the PGI compiler. The compiler flags used to generate the CPU code were: “-O4 -Minfo -Mpreprocess -mp=all -Mvect -Minline”, and the compiler version was PGI 13.6 (latest stable-release at the time of the tests). According to the output from the “-Minfo” flag this compiler was able to automatically generate vector instructions (SSE, AVX) for the target CPU platforms.

As previously noted, the C++ version of the INS code incorporates many low-level optimizations tailored specifically to the target CPU platforms. This version achieves generally superior performance on the Intel Xeon processors relative to the Fortran INS code (particularly on the Sandy Bridge microarchitecture) but cannot be run on GPUs and may be suboptimal on other architectures even if successfully compiled for them. The C++ code should be taken to represent what is possible when significant programming effort is put into a single platform, but it should also be realized that few CFD codes are written to this level of performance optimization because it is simply not practical from a software maintenance perspective.

As described in Chapter 1.5, 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. In Figure 27, the maximum double-precision performance attained by each platform on large grids (with over 4000×4000 nodes) is compared. In Figure 28, the relative speedup of each GPU vs a single CPU core is shown.
Figure 27. Peak double-precision performance attained by test platforms on the LDC benchmark. Details for each test case are given in Table 5. Key: (Green) Intel Xeon CPUs, (Red) NVIDIA GPUs, (Black) AMD GPUs.

<table>
<thead>
<tr>
<th>Version</th>
<th>Platform</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Xeon X5560 (Nehalem)</td>
<td>PGI 13.6 compiler. No performance benefit was observed from hyper-threading with this version, so only 1 thread per core was used.</td>
</tr>
<tr>
<td>CPU1</td>
<td>8 threads / 8 cores</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>Xeon X5560 (Nehalem)</td>
<td>GCC 4.7.2 compiler with SSE2 intrinsics. Approx. 25% performance increase was observed from hyper-threading, so two threads per core were used.</td>
</tr>
<tr>
<td>CPU1</td>
<td>16 threads / 8 cores</td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td>Xeon E5-2687W (Sandy Bridge)</td>
<td>PGI 13.6 compiler. Hyper-threading yielded slight performance regression, so only 1 thread per core was used.</td>
</tr>
<tr>
<td>CPU2</td>
<td>16 threads / 16 cores</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>Xeon E5-2687W (Sandy Bridge)</td>
<td>GCC 4.7.2 compiler with AVX-256 intrinsics. Approx. 20% performance increase was observed from hyper-threading, so two threads per core were used.</td>
</tr>
<tr>
<td>CPU2</td>
<td>32 threads / 16 cores</td>
<td></td>
</tr>
<tr>
<td>OpenACC</td>
<td>NVIDIA C2075 (GF110)</td>
<td>PGI 13.6 compiler. Using thread-block size optimization (16x4 blocks).</td>
</tr>
<tr>
<td>GPU1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenACC</td>
<td>NVIDIA K20c (GK110)</td>
<td>PGI 13.6 compiler. Using thread-block size optimization (16x8 blocks).</td>
</tr>
<tr>
<td>GPU2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenACC GPU3  |  NVIDIA K20x (GK110)  |  PGI 13.6 compiler. Using thread-block size optimization (16x8 blocks).
--- | --- | ---
OpenACC GPU4  |  AMD 7990 (GCN)  |  PGI 14.1 compiler. Only one of the two dies available on the card was used. Thread-block configuration was left at compiler default.

Table 5. Specifications for the test cases displayed in Figure 27.

Figure 28. Relative speedup of GPU platforms vs a single CPU core. (Left) Speedup over single Sandy Bridge E5-2687W core. (Right) Speedup over single Nehalem X5560 core.

The results presented in Figure 27 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 Chapter 4.2 were used, not the default. In the case of the Fortran CPU version, the grid size was selected carefully to return the best results, because the actual performance varied significantly with the size of the grid as noted in Chapter 5.2. This highlights 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 14).
7 Conclusions

The directive-based OpenACC programming model proved very capable in the test CFD application, permitting the creation of an efficient cross-platform source code. Using the PGI 13.6 implementation of OpenACC, it was possible 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 new version of the PGI compiler (14.1) 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 [9].

A comparison of the cross-platform INS code with a second version specifically optimized for the Intel Xeon architectures under investigation revealed some of the difficulties inherent in achieving optimal performance on modern computational hardware. While both versions of the code implemented identical numerical algorithms, the low-level C++ was seen to outperform the Fortran on all of the CPUs, in some cases by almost a factor of two, and to achieve much better results when using SMT (two threads per core). This should not be interpreted as poor performance on the part of PGI Fortran because, as described in Chapter 5.2, a great deal of platform-specific tuning was required to attain the high performance seen with the C++ code. Rather, the discrepancy shows that generating optimal code for a given processor is an extremely complex task that in general cannot always be accomplished automatically at compile-time. Some factors, such as the cache conflict misses observed in both versions of the code, are highly dependent on interactions between the particular algorithm and architecture, and are thus extremely difficult for compilers to predict heuristically. In this particular case, alleviating the problem was only possible through empirical, brute-force tuning of cache block size and data padding parameters. Such optimizations are time consuming and not portable, and it is telling that the multi-platform OpenACC+Fortran code, when simply
recompiled and run on the latest GPU hardware, outperformed even the most-optimized CPU code.

In Chapter 4 it was observed 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 some additional programming effort beyond inserting directive statements, 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 disabled very useful features of modern Fortran (e.g. function pointers) and impeded the use of certain object-oriented programming techniques. These limitations were sufficient to determine 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. Instead, future work will focus on 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 accelerator platforms [35, 36].
Appendix

A.1 Arithmetic Throughput vs Memory Bandwidth

To illustrate the discrepancy between arithmetic throughput and memory bandwidth in modern HPC hardware, consider the 8-core Intel Xeon E5-2660 CPU used in Chapter 5, which has a published maximum memory bandwidth of 51.2 GB/s and is thus capable of reading a maximum of 6.8 billion 8-byte floating point numbers per second [27]. Under ideal conditions\(^1\), each core of this processor is capable of performing one floating point addition and one floating point multiplication per clock cycle [19], giving a maximum arithmetic throughput of \(2.7 \text{ GHz} \times 2 \times \text{FP ops} \times 8 \text{ cores} = 43.2 \text{ billion floating point operations per second}\). If the full 256-bit SIMD capability (AVX) of the CPU is used, the throughput increases by a factor of four to 172.8 billion floating point operations per second, which is 25 times greater than the number of data loads. Even considering that few algorithms will make such optimal use of the superscalar hardware, it is clear that there is a major inconsistency between the memory and arithmetic performance (the ratio for this processor is 3.375 DP FLOPs per byte). A similar trend can be observed in the specifications for the other HPC platforms listed in Appendix A.2; note that the GPUs tend to have a larger ratio of arithmetic to memory performance.

A.2 Hardware Specifications

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Architecture</th>
<th>Compute Capability</th>
<th>Memory (Size)</th>
<th>Memory (Bandwidth)</th>
<th>Peak SP GFLOPS</th>
<th>Peak DP GFLOPS</th>
<th>TDP (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla C2075</td>
<td>Fermi GF110</td>
<td>2.0</td>
<td>6 GB</td>
<td>144 GB/s</td>
<td>1030</td>
<td>515</td>
<td>225</td>
</tr>
<tr>
<td>Tesla K20c</td>
<td>Kepler GK110</td>
<td>3.5</td>
<td>5 GB</td>
<td>208 GB/s</td>
<td>3520</td>
<td>1170</td>
<td>225</td>
</tr>
<tr>
<td>Tesla K20x</td>
<td>Kepler GK110</td>
<td>3.5</td>
<td>6 GB</td>
<td>250 GB/s</td>
<td>3950</td>
<td>1310</td>
<td>235</td>
</tr>
</tbody>
</table>

Table A1. NVIDIA GPUs.

<table>
<thead>
<tr>
<th>Model</th>
<th>Architecture</th>
<th>Memory (Size)</th>
<th>Memory (Bandwidth)</th>
<th>Peak SP GFLOPS</th>
<th>Peak DP GFLOPS</th>
<th>TDP (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD 7970</td>
<td>Sea Islands GCN</td>
<td>3 GB</td>
<td>264 GB/s</td>
<td>3788</td>
<td>947</td>
<td>250</td>
</tr>
</tbody>
</table>

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

\(^1\) In order to achieve maximum floating point throughput, Sandy Bridge requires a nominally equal number of independent addition and multiplication instructions to be interleaved [14]. This occurs only rarely in real applications.
<table>
<thead>
<tr>
<th>Model</th>
<th>Architecture</th>
<th>ISA Extension</th>
<th>Threads / Cores</th>
<th>Clock (MHz)</th>
<th>Memory (Bandwidth)</th>
<th>Peak SP GFLOPS</th>
<th>Peak DP GFLOPS</th>
<th>TDP (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X5355</td>
<td>Core</td>
<td>SSSE3</td>
<td>4/4</td>
<td>2667</td>
<td>21 GB/s</td>
<td>84</td>
<td>42</td>
<td>120</td>
</tr>
<tr>
<td>X5560</td>
<td>Nehalem</td>
<td>SSE4.2</td>
<td>8/4</td>
<td>3067</td>
<td>32 GB/s</td>
<td>98</td>
<td>49</td>
<td>120</td>
</tr>
<tr>
<td>E5-2687W</td>
<td>Sandy Bridge</td>
<td>AVX</td>
<td>16/8</td>
<td>3400</td>
<td>51 GB/s</td>
<td>434</td>
<td>217</td>
<td>150</td>
</tr>
</tbody>
</table>

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


