Optimization and Validation of Discontinuous Galerkin Code for the 3D Navier-Stokes Equations

by

Eric Hung-Lin Liu

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aerospace Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author . . . .
Department of Aeronautics and Astronautics
January 27, 2011

Certified by . . . .
David L. Darmofal
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by . . . .
Eytan H. Modiano
Associate Professor of Aeronautics and Astronautics
Chairman, Graduate Program Committee

Optimization and Validation of Discontinuous Galerkin Code for the 3D Navier-Stokes Equations

by

Eric Hung-Lin Liu

Submitted to the Department of Aeronautics and Astronautics
on January 27, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Aerospace Engineering

Abstract

From residual and Jacobian assembly to the linear solve, the components of a high-order, Discontinuous Galerkin Finite Element Method (DGFEM) for the Navier-Stokes equations in 3D are presented. Emphasis is given to residual and Jacobian assembly, since these are rarely discussed in the literature; in particular, this thesis focuses on code optimization. Performance properties of DG methods are identified, including key memory bottlenecks. A detailed overview of the memory hierarchy on modern CPUs is given along with discussion on optimization suggestions for utilizing the hierarchy efficiently. Other programming suggestions are also given, including the process for rewriting residual and Jacobian assembly using matrix-matrix products. Finally, a validation of the performance of the 3D, viscous DG solver is presented through a series of canonical test cases.

Thesis Supervisor: David L. Darmofal
Title: Professor of Aeronautics and Astronautics

Acknowledgments

I would like to give my thanks to all of my friends and colleagues who have made this research possible. First and foremost, I must thank my advisor, Professor David Darmofal who has given me a great deal of encouragement and guidance throughout our relationship. He even puts up with my inverted, "night-owl" sleep schedule. Professor Darmofal and I first met when I was a freshman at MIT. He offered me a summer, undergraduate research position with the Project X team. Six years later, I have continued to work on the Project X team, and my efforts have crossed over almost as many different topics. I am of course looking forward to continuing working with Professor Darmofal.

Speaking of the Project X team (who also put up with my inverted schedule), I would not be where I am today without the help, cooperation, and tireless efforts of all of them: Julie Andren, Laslo Diosady, Bob Haimes, Josh Krakos, JM Modisette, Huafei Sun, and Masayuki Yano. I would also like to thank past Project X members whom I worked with previously: Garrett Barter, Krzysztof Fidkowski, Todd Oliver; they were all a huge help when I was getting started. Masa deserves an additional thank you for sharing his deep knowledge of finite elements with me and fielding countless questions from me.

I would also like to thank my close friends, Isaac Asher, Sharmin Karim, Tom Morgan, and Kathy Ye, for providing support throughout this process and providing me with ample distractions when coding frustration left me at my wit's end.

Finally, I would like to thank the U.S. Department of Energy and the Krell Institute for awarding me with the D.O.E. Computational Science Graduate Fellowship. The CSGF Program has been a wonderful opportunity for me, both in terms of my research and in terms of meeting new and exciting people.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The classic approach to analyzing the performance of a numerical algorithm is to count the number of arithmetic operations required. For example, it is well known that the inner product $x^T y$ for $x, y \in \mathcal{R}^n$ requires n floating-point multiplies and $n - 1$ floating-point additions; or that the matrix product AB for $A \in \mathcal{R}^{m \times k}$ and $B \in \mathcal{R}^{k \times n}$ requires mnk multiplies and $mn(k-1)$ additions. For dense inputs, exactly these amounts of computation *must* occur. The basic matrix-matrix product algorithm is a mere 10 lines of code. But the most advanced BLAS implementations involve thousands of lines of hand-crafted assembly.

Clearly, a simple count of arithmetic operations is insufficient. For starters, the multiplication operations are about 5 times as expensive as the additions. Additionally, in order to multiply two array elements, the computer must resolve their memory addresses. This amounts to additional integer multiply and add operations. However, the CPU also needs to access those addresses: memory access is the dominant cost by far.[1] Memory accesses are around 50 times more costly than arithmetic operations for these types of algorithms. Table 1.1 contains the cycle-latencies of a few common CPU instructions (data movement, integer arithmetic, floating point arithmetic) along with the latency incurred by reading from main memory.

If the CPU always waited 200 cycles (memory access) to do 5 cycles (floating point multiplication) of work, the CPU would end up spending most of its time waiting. As

[1] For sufficiently large problems; i.e., problems that do not fit in cache.

Description	x86 ASM	Latency	Throughput
Move from r to r	MOV	1	3
Move from r to m	MOV	2	1
Move from m to r	MOV	3	1
Logic (AND, OR, XOR)	AND, OR, XOR	1	3
`int` add	ADD	1	3
32-bit, signed `int` mult	IMUL	3	1
64-bit, signed `int` mult	IMUL	5	$\frac{2}{3}$
32-bit, signed `int` div	IDIV	14-23	-
64-bit, signed `int` div	IDIV	34-88	-
x87 `double` load/store	FLD/FST	1-3	$\frac{1}{3}$-1
x87 `double` add	FADD	3	1
x87 `double` mult	FMUL	5	0.5
x87 `double` div	FDIV	6-21	0.05-0.2
Packed `int` move	MOVDQA	2	1
Packed `int` add	PADDQ	1	2
Packed `int` mult	PMULDQ	3	1
Packed, aligned `double` move	MOVAPD	1-3	1-3
Packed `double` add	ADDPD	3	1
Packed `double` mult	MULPD	5	1
Packed `double` div	DIVPD	6-21	0.05-0.2
register miss	-	3-5	-
L1 cache miss	-	10-20	-
L2 cache miss	-	≈ 200	-

Table 1.1: Instruction latencies (in CPU cycles) for the Intel Core2 (45nm) processor[26]. The letters r,m,xmm indicate a general register, a memory address, and an XMM (SSE) register, respectively. All x87 instructions assume full precision (80 bits); note that double-precision SSE instructions are 64-bit. Latency indicates the time taken by the appropriate execution unit to complete the specified operation. Unless otherwise specified, all times assume that operands are already in registers and no exceptions (e.g., NAN) occured. Throughput is the maximum number of independent instructions of a type simultaneously in flight; a value of say 0.25 indicates that this instruction can be executed at most once every 4 cycles.

a result, modern CPUs have several levels of *cache*. Cache provides much faster access to data; for example, the L1 cache on an Intel Core 2 has a 3 cycle latency and the L2 cache has a 15 cycle latency. Unfortunately, due to expense, caches are very small compared to main memory; the latency and size of cache, memory, etc. scale inversely. In modern computers, the system of fast caches, main memory, and data storage in general is often called the *memory hierarchy*. Figure 1-1 provides a pictorial overview of the memory hierarchy of modern CPUs, showing some rough estimates for latencies and costs of various cache levels. The memory hierarchy is discussed at length in Chapter 4.

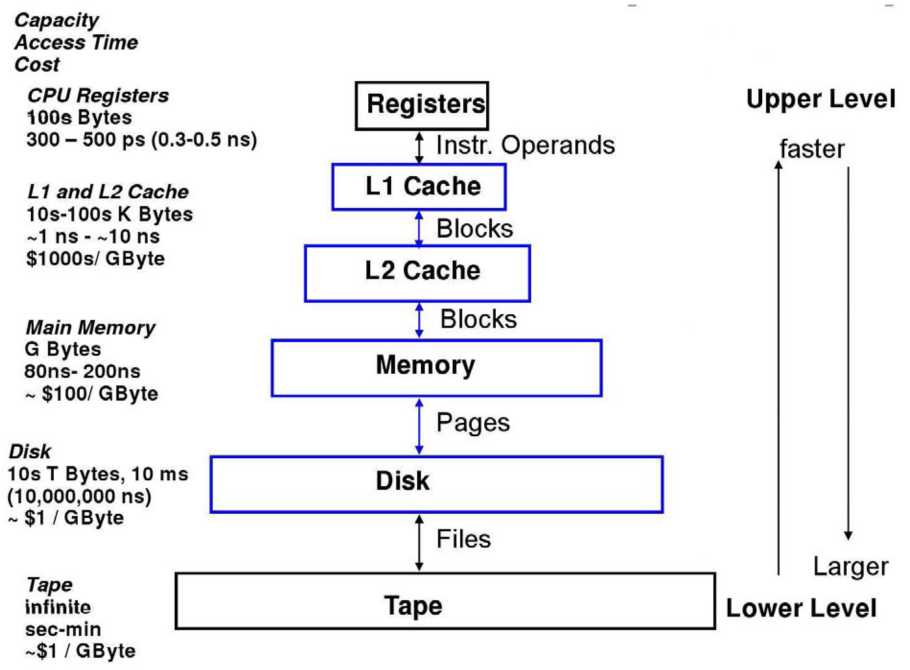

Figure 1-1: A rendition of the cache hierarchy in modern computers[4].

Over the last few years, the Project X group in the MIT Aerospace Computational Design Laboratory has been developing an *hp*-adaptive, Discontinuous Galerkin Finite Element Method (DGFEM). Thus far, research and development have focused on the underlying numerical algorithms. However, as problems of interest become increasingly complex (e.g., viscous flows in 3D), improving computational performance has come into the spotlight. In this thesis, we will survey the aspects of modern computer architecture that are most relevant to DGFEM methods, giving emphasis to the memory hierarchy. Finally, the correctness of the DGFEM method in Project X will be demonstrated through a series of canonical test cases.

The goals of this thesis are as follows:

- Analyze the performance characteristics of a higher-order, DGFEM code (ProjectX) on model problems that emphasize relevant modes of operation. Apply the analysis to guide optimization of the residual and Jacobian assembly processes as well as the linear solve.

- Validate the steady-state, viscous, 3D discretization of ProjectX by performing grid-convergence studies on various test cases.

- Produce this document as a guide and starting point for other researchers and engineers interested in writing high-performance scientific codes. Many examples will be drawn from a DGFEM implementation, but the ideas discussed are relevant to a much broader class of programs.

While the present thesis concentrates on optimization of a serial implementation, most applications will require a parallel implementation. However, for the residual and Jacobian assembly, improvements to the serial performance reflect similarly in the parallel performance. Differences in relative improvements in the serial versus parallel performance would be attributable to the communication overhead, which is small for DG methods. For the linear solver, the same statement may be true, depending on how the parallel system is preconditioned. Thus, while parallel cases are not specifically discussed in this thesis, the serial optimization techniques remain relevant.

This thesis is organized as follows. The remainder of the introduction includes a review of relevant literature and an overview of software tools used in the optimization process. Chapter 2 covers the DG discretization of the Navier-Stokes equations, along with the computational costs of the DG method (residual and Jacobian assembly, nonlinear solver, linear solver). Example cases are provided, and major performance limitations in DG solvers are identified. Chapter 4 covers the memory hierarchy used in modern computers, including discussion of how to utilize it efficiently. Chapter 5 covers particular components of a DG implementation that should be targeted for optimization. Chapter 6 presents the validation of a DGFEM solver for the Navier-Stokes equations in 3D. The appendices cover the implementation of viscous fluxes for the Navier-Stokes equations (Appendix A), the operation and performance of modern DDR memory modules (Appendix B), the operation of modern CPUs (Appendix C), two stopping-criterion choices for iterative linear solvers (Appendix D), and a collection of helpful, performance-enhancing coding practices (Appendix E).

1.1 Literature Review

While the theory of DG methods is well-developed, the literature contains relatively little on the optimization of DG implementations. However, regardless of the application, the need for efficient, high-performance codes is well recognized. This survey covers some relevant developments in CG methods, sparse iterative solvers, and dense linear algebra on traditional CPUs; references for implementing and optimizing DG methods and Krylov methods are also presented. Unfortunately, few works cited here discuss methods that are directly applicable to this thesis. For example, most sparse matrix-vector optimizations are designed for general sparse matrices.[2] But the principles used to guide these optimizations are relevant. The most common theme in the cited works is the importance of properly utilizing the memory hierarchy of the target architecture to improve performance.

As discussed in Chapter 4, modern CPUs (and GPUs) have very deep memory hierarchies. As a result, using algorithms that often access the slowest memory levels is generally sub-optimal. This fact is well-recognized by computer scientists, engineers, etc; thus fast algorithms must find ways to minimize communication between fast cache(s) and slow memory (or even slower hard disks).

The BeBOP group at the University of California, Berkeley is responsible for a great deal of work on communication-minimizing implementations of common linear algebra kernels [10, 34, 53]. [34, 53] discuss a class of communication-avoiding Krylov methods.[3] These

[2]The Jacobian matrix arising from DG discretizations is special in that it possesses natural block structure. Jacobians from other discretizations (e.g., finite difference or finite volume) do not have this structure. Note that the DG block structure is not helpful when the blocks are tiny; i.e., the polynomial order and/or the number of states is low.

[3]In parallel, communication refers to node-to-node data transfer. In serial, communication refers to

methods perform the same (asymptotically) amount of work as a s-step Krylov method, but involve only as much communication as a single step. The methods are divided into several kernels. For example, GMRES involves communication-avoiding matrix powers (computing $b, Ab, A^2b, \ldots, A^sb$), block Gram-Schmidt, QR factorization of a "tall and skinny" matrix, and other relatively minor dense matrix operations. [34] also provides substantial discussion on the numerical stability of these methods. Unfortunately, these methods are currently limited to relatively weak preconditioners (e.g., Jacobi).

One particularly common technique for communication reduction is called blocking or tiling; blocking is important for both sparse and dense linear algebra, though the particulars of its implementation differ. For a (dense) matrix-matrix product, its optimality was proven by Hong for dense problems implemented sequentially (proof later strengthened to the parallel case by Irony). [10] extends previous analysis to include communication lower bounds for other direct linear algebra kernels (e.g., Cholesky, LU, QR, eigen/singular values, etc.) for both dense and sparse matrices, in both sequential and parallel environments. Implementation is not the focus of their work; indeed not all of the analyzed kernels have matching, optimal implementations.

[45] provides a detailed description and analysis of the blocking technique for dense matrix-matrix products. Optimized implementations of the BLAS[4] 3 function `dgemm` take heavy advantage of this optimization; BLAS 2 functions (e.g., `dgemv`) also utilize blocking, but the benefit is less substantial. Since optimized BLAS implementations are commonplace, many researchers attempt to improve memory performance by restructuring their code to utilize these routines [6, 21, 43, 44]. Outside of blocking for matrix-matrix products, [10] provides references for a number of other algorithms for direct linear algebra achieving optimal communication bounds.

For sparse linear algebra, the concept of minimizing memory transfers remains important, but the exact meaning of blocking is different. [35] provides a thorough discussion

memory reads and writes.

[4]The Basic Linear Algebra Subroutines are a set of functions for solving problems that are fundamental in dense linear algebra. The functions are divided into 3 levels: BLAS 1, BLAS 2, and BLAS 3. BLAS 1 performs $\Theta(m)$ work on $\Theta(m)$ data; e.g., operations (scaling, dot product) on vectors. BLAS 2 performs $\Theta(m^2)$ work on $\Theta(m^2)$ data; e.g., matrix-vector operations and outerproducts. BLAS 3 performs $\Theta(m^3)$ work on $\Theta(m^2)$ data; e.g., matrix-matrix operations.

(implementation, analysis, experiment) of the two most common techniques, (sparse) cache-blocking and (sparse) register-blocking. [55] provides an important insight: the effectiveness of (sparse) cache-blocking is primarily governed by TLB misses;[5] [19] also notes the importance of TLB misses on sparse matrix operations (in the context of multigrid). One major issue with these optimizations is that the performance is strongly dependent on the sparsity pattern, in addition to block-size and computer architecture. The issue is often approached with auto-tuning, wherein a software package estimates performance for a given matrix type and generates specialized code accordingly. By comparison, static, architecture-targeted code (e.g., MKL, GOTO) can be generated for dense linear algebra; these routines often outperform auto-tuned code (e.g., ATLAS).

Baker's LGMRES and BLGMRES methods improved iterative solver performance by heuristically augmenting the Krylov subspace used in GMRES [8, 7]. LGMRES adds additional Krylov vectors based on previous restarts; BLGMRES adds additional right-hand sides. The Ax kernel of a BLGMRES solver with 4 right-hand sides could be expected to be 4 times as costly as Ax in standard GMRES, since there are 4 times as many Ax products and 4 times as many Krylov vectors. BLGMRES almost never cuts the number of GMRES iterations by a factor of the number of right-hand sides. The performance of BLGMRES depends on handling all right-hand sides simultaneously (by interleaving the vectors) instead of sequentially. This prevents loading A from memory repeatedly, an operation which is memory bandwidth-limited.

There has also been work implementing GMRES on GPUs for general sparse matrices[46, 70]. The focuses of these works is preconditioning and the sparse matrix-vector product kernel in GMRES, the latter of which is also discussed in [15]. These researchers implemented code on essentially the same hardware, but often saw very different performance gains. The differences are likely due to a combination of differing implementations and differing input matrices. Improving sparse matrix vector products on GPUs centers around selecting the best sparse storage format; while CSR (or CSC)[6] is usually adequate on CPUs. On the

[5]TLB stands for Translation Lookaside Buffer; it is used to cache memory address translations from virtual to physical. See Section 4.3.1 for details.

[6]Compressed Sparse Row or Column.

preconditioning side, the most common choice for GPUs was subdomain-wise block Jacobi with inexact local solves via ILU. Unfortunately this preconditioner does not exhibit weak scaling,[7] so its performance is lacking for higher order DG discretizations of the Navier-Stokes equations [18].

Residual and Jacobian assembly are far less analyzed than linear solvers. [54] summarizes the importance of this component of finite element methods:

> Besides the solution of the set of linear equations, the element evaluation and assembly for stabilized, highly complex elements on unstructured grids is often a main time consuming part of the calculation. Whereas a lot of research is done in the area of solvers and their efficient implementation, there is hardly any literature on efficient implementation of advanced finite element formulations. Still a large amount of computing time can be saved by an expert implementation of the element routines.

They consider CGFEM on a vector machine (SX-6), which lives between modern CPUs and GPUs. The operation reordering they suggest substantially improves performance on a vector machine. However it is not the best choice for CPUs. Their ordering is not what [44] use, but it may have applications on GPUs as well. Also [54] provides examples that demonstrate the effectiveness of the `restrict` keyword; see Appendix E for details.

[44] considers the evaluation of the DG residual on GPUs, for Maxwell's Equations, which form a *linear*, hyperbolic system. They were interested in time-dependent problems; RK4 provided an explicit temporal discretization. Implicit-time solvers and elliptic operators were not considered; Jacobian assembly (especially with viscous terms) adds considerable complexity. Their work was also limited to linear representations of boundaries. The optimizations considered center around stating the residual assembly process using BLAS 3 functions.[8] Due to substantial restrictions on data layout imposed by current GPU BLAS implementations, [44] designs new algorithms and orderings to achieve better performance. Their work focused around how to divide the work of residual assembly across hundreds of GPU threads; how to store solution data and intermediate results to optimize memory

[7] An algorithm scales weakly when increasing the problem size and number of processors such that the degrees of freedom per processor remains constant does not affect the run-time.

[8] The technique of [44] is different than what will be suggested in Section 5.1 because [44] uses optimizations specific to DG discretizations of linear, hyperbolic equations using linear elements.

performance; and how and when to fetch data from memory so that numerous threads could be served in minimal time.

1.2 Optimization Basics

Avoiding premature optimization is always critical. Before proceeding with any code modification, a good first step is to evaluate which parts of the code are consuming the greatest amount of time ("hot spots") and which parts of the code are easiest to improve. Such evaluations are typically case dependent; for example, with flow solvers, steady solutions spend substantially more time in the linear solve than their unsteady counterparts. If a program component only consumes 1% of the runtime, then even if it could be removed completely, the speedup is only 1%.[9] If that 1% gain costs months of development time, it may not be worthwhile. At the same time, if it takes a few hours of work, every little bit helps. For better or for worse, many complex programs do not spend the majority of their time in a single leaf routine; thus performance gains often come through a combination of many small gains, rather than through one large step.

Additionally, improving "wall-clock" performance needs to be weighed against decreases in "performance" measured in terms of code clarity, maintainability, and extensibility. For example, there are very few (if any) instances in finite element methods where developers should be coding in assembly language, even though carefully tuned assembly can provide the best performance as with some optimized BLAS libraries. The time required to develop and maintain code is also a resource, and the balance between performance measures like development and execution time will depend on the objectives of a code project.

A number of tools exist to aid in the profiling process. A few popular examples include OProfile[42], GNU gprof[22, 30], and Intel's VTune[37]. Profiling falls into two main categories: event-based and time-based. Time-based profiling attempts to measure how much time a program spends in each of its member functions.

Time-based profiling can be as simple as inserting system clock (or other timing mecha-

[9]This analysis style follows Amdahl's Law, a well-known formula for computing the maximum possible speedup obtainable through parallelizing routines.

nism) calls around subroutines of interest; e.g., to test the amount of time taken in residual and Jacobian evaluation. Such methods are usually low-resolution, but provide easy access to the "big-picture." GNU gprof provides high resolution timing of every function call across the duration of a program run. It performs the measurement by inserting timing code at the beginning and end of every function. As a result, gprof (and tools like it) incur additional overhead from all the extra timing code; the effect of this overhead depends on both the number of function calls and the size of called functions. Fog[28] details how to obtain high resolution timing more precisely, inserting timing code only where the programmer desires. Intel's VTune package provides another alternative; it obtains timing results by randomly querying the program state and estimating the time distribution across different functions. This method involves much less overhead, but it also leads to inaccuracies.

Timing-based profiling provides information on what program components are the most expensive. But it provides little information on why. Event-based profiling fills this gap. Modern CPUs have a number of *performance counters* (also called *event counters*) on the chip. These counters are incremented whenever certain events occur; e.g., cache misses, instructions retired, branch misprediction, overloading performance-enhancing (hardware) buffers, etc. During the discussion on the memory hierarchy and the inner workings of the CPU, notes will be provided on relevant event counters.[10] These counters can be accessed directly in code[28], but the process is not straightforward. OProfile and VTune both provide the ability to accumulate CPU events across entire program runs. For efficiency, both packages sample randomly. The random sampling does not hinder accuracy in a meaningful way because only functions incurring large numbers of events will be interesting. Programmers wishing for more direct control will need to access the performance counters individually, in code.

[10]Event counter names change from CPU to CPU; the counters discussed here will be taken from the Intel Core 2 only.

Chapter 2

The Discontinuous Galerkin Finite Element Method

In this work, governing equations that take the general form:

$$\frac{\partial}{\partial t}\mathbf{u} + \nabla \cdot \mathbf{F}^{inv}(\mathbf{u}) - \nabla \cdot \mathbf{F}^{vis}(\mathbf{u}, \nabla\mathbf{u}) = 0,$$

such as Euler and Navier-Stokes, will be considered. In the above, $\mathbf{u}$ denotes the conservative state vector (of size N_{sr}); e.g., $\mathbf{u} = [\rho, \rho v_i, \rho E]^T \in \mathcal{R}^{N_{sr}}$, which is $[\rho, \rho v_0, \rho v_1, \rho E]^T$ in 2D. Here, v_i denotes the components of velocity and E denotes the total specific internal energy. The inviscid and viscous fluxes are denoted by $\mathbf{F}^{inv}, \mathbf{F}^{vis} \in \mathcal{R}^{N_{dim} \times N_{sr}}$, respectively. In the following, only viscous fluxes of the form $\mathbf{F}^{vis} = \mathcal{A}(\mathbf{u})\nabla\mathbf{u}$ will be considered. Note that $\mathcal{A}(\mathbf{u})$ is a rank-4 tensor and $\nabla\mathbf{u}$ is rank-2. The form of $\mathcal{A}$ is given in Section A.3. For clarity, the governing equations written with indicial notation are given below:

$$\partial_t u_k + \partial_{x_i} F_{i,k} - \partial_{x_i}(\mathcal{A}_{i,j,k,l}\partial_{x_j} u_l) = 0$$

where k, l range over the state rank (N_{sr}) and i, j range over the spatial dimension (N_{dim}). $F_{i,k}$ denotes the N_{sr} components of $\mathbf{F}^{inv}$ in each coordinate direction. $F^{vis}_{i,k} = \mathcal{A}_{i,j,k,l}\partial_{x_j} u_l$ denotes the components of $\mathbf{F}^{vis}$. See Appendix A for a complete specification of the (continuous) inviscid and viscous fluxes for the Navier-Stokes equations.

2.1 DG Spatial Discretization

The following discussion will only outline the DG spatial discretization. Since this work is concerned with implementation issues and not theoretical developments, results are stated generally without proof. Further discussion of the DG spatial discretization used here can be found in [24, 23, 49, 57].

Begin by triangulating the domain $\Omega \subset \mathcal{R}^{N_{dim}}$ into non-overlapping elements κ. Let $T^h \equiv \{\kappa_e\}$ and $N_{elem} = |T^h|$. The DG method seeks a discontinuous approximation u^h (to the exact solution u) such that $u^h|_{\kappa_e}$ belongs to the function space $\mathcal{V}^{h,p}(\kappa_e)$. The solution and test space is given by:

$$\mathcal{V}^{h,p} \equiv \{v_k \in [L^2(\Omega)]^{N_{sr}} | v_k|_\kappa \circ f_\kappa \in [P^p(\kappa_{ref})]^{N_{sr}},\ \forall \kappa \in T^h\},$$

where P^p denotes polynomials with degree at most p, and f_κ denotes the mapping from the reference element (κ_{ref}) to physical space for the element κ.

Multiplying by the vector of test functions, v_k^h, and integrating, the semidiscrete,[1] weak form of the governing equations follows:

$$\sum_{\kappa \in T^h} \int_\kappa v_k^h \frac{\partial u_k^h}{\partial t} + R^{h,inv}(\mathbf{u}^h, \mathbf{v}^h) + R^{h,vis}(\mathbf{u}^h, \mathbf{v}^h) = 0,\ \forall v_k^h \in \mathcal{V}^{h,p}, \tag{2.1}$$

where $R^{h,inv}, R^{h,vis}$ denote the inviscid and viscous residuals, respectively; and $\mathbf{u}^h, \mathbf{v}^h$ indicate vector-valued arguments, u_k^h, v_k^h. If source terms are present, then $R^{h,sou}$ will also be present in Equation 2.1. Note that if the (strong) source terms contain first or higher derivatives, care is required to maintain the dual consistency of the discretization [57].

[1]The time discretization will be handled later.

2.1.1 Inviscid Component

The inviscid discretization arises from multiplying $\partial_{x_i} F_{i,k}$ by the test functions v_k and integrating by parts to obtain the weak form:

$$R_\kappa^{h,inv}(\mathbf{u},\mathbf{v}) \equiv -\int_\kappa \partial_{x_i} v_k^+ F_{i,k}(\mathbf{u}) + \int_{\partial\kappa} v_k \hat{F}_k(\mathbf{u}^+, \mathbf{u}^-, \mathbf{n}^+), \tag{2.2}$$

where the $+, -$ superscripts refer to element κ and its appropriate neighbor, respectively; note that v_k always refers to test functions for κ evaluated over κ. The h superscripts (u_k^h, v_k^h) have been dropped for brevity. The full inviscid residual is formed by summing over the elements: $R^{h,inv} = \sum_{\kappa \in T^h} R_\kappa^{h,inv}$. $\hat{F}_k$ is a numerical flux function (e.g., the Roe Flux). If some faces of $\partial\kappa$ lie on $\partial\Omega$, then $\hat{F}_k$ is modified to enforce boundary conditions weakly.[2]

2.1.2 Viscous Component

The viscous discretization is derived similarly. Here, the viscious, numerical fluxes are from the second method of Bassi and Rebay (BR2) [13, 14, 12]. In this method, the strong form of the governing equations is split into a system of first order equations:

$$\begin{aligned}
\partial_t u_k + \partial_{x_i} F_{i,k}^{inv} - \partial_{x_i}\left(\mathcal{A}_{i,j,k,l} z_{j,l}\right) &= 0 \qquad (2.3)\\
z_{i,k} - \partial_{x_i} u_k &= 0. \qquad (2.4)
\end{aligned}$$

Obtain the weak forms by integrating by parts after multiplying the first equation by $v_k \in \mathcal{V}^{h,p}$ and the second by $w_{i,k} \in \mathcal{V}^{h,p}$:

$$\begin{aligned}
R_\kappa^{h,inv} + \int_\kappa \partial_{x_i} v_k \mathcal{A}_{i,j,k,l} z_{j,l} - \int_{\partial\kappa} v_k^+ \widehat{\mathcal{A}_{i,j,k,l} z_{j,l}} n_i^+ &= 0 \qquad (2.5)\\
\int_\kappa w_{i,k} z_{i,k} + \int_\kappa u_k \partial_{x_i} w_{i,k} - \int_{\partial\kappa} w_{i,k}^+ \hat{u}_k n_i^+ &= 0, \qquad (2.6)
\end{aligned}$$

[2]A boundary state, u_k^b, is established based on outgoing characteristics and boundary data; the boundary flux is determined from these quantities.

where $\hat{z}, \hat{u}$ are viscous numerical fluxes (i.e., traces). Setting $w_{j,l} = \partial_{x_i} v_k \mathcal{A}_{i,j,k,l}$, substituting for $\int_\kappa \partial_{x_i} v_k \mathcal{A}_{i,j,k,l} z_{j,l}$, and integrating by parts again[3] yields the viscous residual contribution,

$$R_\kappa^{h,vis} \equiv \int_\kappa \partial_{x_i} v_k \mathcal{A}_{i,j,k,l} \partial_{x_j} u_l - \int_{\partial\kappa} \partial_{x_i} v_k \mathcal{A}_{i,j,k,l} \left(u_k^+ - \{u_k\}\right) n_j^+ - \int_{\partial\kappa} v_k^+ \widehat{\mathcal{A}_{i,j,k,l} z_{j,l}} n_i^+ \qquad (2.7)$$

Let $\{u\} = \frac{1}{2}\left(u^+ + u^-\right)$ denote the average operator, and let $[\![u]\!] = \left(u^+ - u^-\right)$ denote the jump operator.[4] The BR2 method involves setting $\widehat{\mathcal{A}_{i,j,k,l} z_{j,l}} = \{\mathcal{A}_{i,j,k,l} \partial_{x_j} u_l\} - \eta\{\mathcal{A}_{i,j,k,l} r_{i,k}([\![\mathbf{u}]\!]\mathbf{n}^+)\}$ and $\hat{u_k} = \{u_k\}$. The parameter $\eta > 3$ is required for stability on simplicial elements [24]; in this work, a more conservative $\eta = 6$ is used, as in [56]. Boundary conditions are set by modifying these traces appropriately [24, 23, 49]. The lifting operator $r_{i,k} \in \mathcal{V}^{h,p}$ is given by:

$$\begin{aligned}
\int_{\kappa^+} \tau_{i,k}^+ r_{i,k}^{f,+} + \int_{\kappa^-} \tau_{i,k}^- r_{i,k}^{f,-} &= \int_{\sigma_f} \{\tau_{i,k}\} [\![u_k]\!] n_i^+ \\
\int_\kappa \tau_{i,k} r_{i,k}^{f,B} &= \int_{\sigma_f^B} \tau_{i,k}^+ \left(u_k^+ - u_k^B\right) n_i^+
\end{aligned}$$

for interior and boundary faces, respectively. The B superscript denotes a quantity over a face that lies on $\partial\Omega$; $\tau_{i,k} \in \mathcal{V}^{h,p}$.

2.2 Temporal Discretization

As in the method of lines, Equation 2.1 transforms to system of ODEs. First, a basis $\{\mathbf{v}_m\}$ for $\mathcal{V}^{h,p}$ is needed. Now $\forall \mathbf{u} \in \mathcal{V}^{h,p}$, $\exists U_m \in \mathcal{R}^{N_{bf}}$ such that $\mathbf{u} = U_m \mathbf{v}_m$. Considering all elements and states simultaneously, the ODE for Equation 2.1 is:

$$M \frac{d\mathbf{U}}{dt} + R(\mathbf{U}) = 0, \qquad (2.8)$$

where $\mathbf{U} \in \mathcal{R}^{N_{elem} N_{sr} N_{bf}}$ is the vector of unknowns over all elements; $R_m(\mathbf{U}) = R^{h,inv}(\mathbf{u}, \mathbf{v}_m) + R^{h,vis}(\mathbf{u}, \mathbf{v}_m)$ involves evaluating the previous $R^{h,inv}, R^{h,vis}$ expressions for all N_{bf} basis functions; and $M_{m,n} = \int_\Omega \mathbf{v}_m \mathbf{v}_n$ is the elemental mass matrix. The matrix M is of size

[3]Integrating $\int_\kappa \partial_{x_i} w_{i,k} u_k = \int_{\partial\kappa} w_{i,k} u_k^+ n_i - \int_\kappa w_{i,k} \partial_{x_i} u_k$ completes the terms required for dual consistency.

[4]The average and jump are taken component-wise if u is a vector.

$N_{elem}N_{sr}N_{bf} \times N_{elem}N_{sr}N_{bf}$, composed of all N_{elem} elemental matrices $M_{m,n}$. Note that the mass matrix is block diagonal with at worst N_{elem} blocks of size $N_{sr}N_{bf} \times N_{sr}N_{bf}$. With the optimal ordering of unknowns, $R_{e,k,n}$ (see Section 3.1.1), it improves to $N_{elem}N_{sr}$ blocks of size $N_{bf} \times N_{bf}$; furthermore, the same polynomial basis is usually taken for every state.

From here, any standard ODE solver can be applied. For unsteady problems, this work uses an IRK4 variant [69]. For steady problems, time-accuracy is not important. An implicit method is important to allow for very large time steps; Backwards Euler suffices:

$$\frac{1}{\Delta t} M \left(\mathbf{U}^{m+1} - \mathbf{U}^{m} \right) + R(\mathbf{U}^{m+1}) = 0. \tag{2.9}$$

The time step is controlled by calculating $\Delta t = \min_e \Delta t_e$ where $\Delta t_e = CFL^m \frac{h_e}{\lambda_e}$. h_e, λ_e are per-element measures of grid size and maximum characteristic speed, respectively. The initial value of CFL is set low for robustness (i.e., so that non-physical transients arising from poor initial conditions do not derail the solver). After each successful iteration, CFL is increased by a constant factor to accelerate convergence.

Each time-step requires the solution of a nonlinear problem; this is handled with Newton's Method. Each Newton iteration requires the solution of a linear system. Here, a block-ILU preconditioned, restarted GMRES method with block MDF reordering was chosen as the linear solver.[5] The GMRES implementation used in this work follows the original algorithm [61, 62].

For unsteady problems, Newton iteration continues until the norm of the temporal residual is sufficiently reduced. Since time-accuracy is not important for steady problems, only one Newton step is taken by each nonlinear solve. The following update scheme results after linearizing about $\mathbf{U}^m$:

$$\mathbf{U}^{m+1} = \mathbf{U}^{m} - \left(\frac{1}{\Delta t} M + \left. \frac{\partial R}{\partial \mathbf{U}} \right|_{\mathbf{U}^m} \right)^{-1} R(\mathbf{U}^m). \tag{2.10}$$

Note that this scheme reduces to Newton's Method as $\Delta t \to \infty$. The nonlinear solver halts when $\|R(\mathbf{U}^m)\| \leq TOL$.

[5]ILU and MDF stand for Incomplete-LU (factorization) and Minimum Discarded Fill, respectively. See [18, 60] for details.

Chapter 3

Computational Costs of the DG Method

3.1 Residual and Jacobian Evaluation

The discussion here will be limited to interior faces. Boundary conditions generally make up a negligible portion of the overall residual and Jacobian assembly time, since they are applied on a set of measure 1 less than the problem domain.

In the following discussion, ϕ refers to the basis functions for the polynomial interpolation space. In this thesis, a Lagrange basis (on uniformly distributed points) was used. For p-th order polynomial interpolant on a simplex, $N_{bf} = p+1$ in 1D, $N_{bf} = (p+1)(p+2)/2$ in 2D, and $N_{bf} = (p+1)(p+2)(p+3)/6$ in 3D.

3.1.1 Inviscid Discretization

Galerkin Residual

The basic kernel for the residual over elements due to an inviscid operator involves numerically integrating $\int_K F_{i,k}(\mathbf{u})\partial_{x_i}\phi$. At a single quadrature point (in reference coordinates), computing the residual involves evaluating:

$$R_{g,k,n} = w_g \partial_{x_i}\phi_{g,n} F_{i,k}(\mathbf{U}_n \phi_{g,n}), \tag{3.1}$$

where n ranges over the basis functions, and g ranges over the number of quadrature points, N_{quad}. Note that the boldfaced terms indicate vectors of size N_{sr}. An index is avoided since the flux function requires the full solution (state) vectors as arguments. In particular, at a single quad point, $\mathbf{U}_n\phi_n$ is a N_{sr}-vector, the product of a $N_{sr} \times N_{bf}$ matrix and a N_{bf} vector. The weights w_g include the (reference) quadrature weights as well as a factor arising from the determinant of the geometric Jacobian, $|J|$, due to the change from physical to reference coordinates. Then the complete residual is computed by $R_{k,n} = \sum_g R_{g,k,n}$. The physical gradient of ϕ_n uses the inverse (geometric) Jacobian, $J^{-1}_{i,j}$: $\partial_{x_i}\phi_n = J^{-1}_{i,j}\partial_j\phi_n$, where $\partial_j\phi_n$ are the derivatives of ϕ_n in reference coordinates.

At a quad point, the total cost is $N^3_{dim} + 4N^2_{dim}N_{bf} + 2N_{sr}N_{bf} + 2N_{dim}N_{sr}N_{bf} + C(F)$. Note that these costs assume that the basis functions are precalculated. For now, the cost analysis will focus on the number of floating point operations, which also serve as an upper bound for memory costs. Memory access costs, which are generally more important, will be considered in Section 5.1, after covering the memory hierarchy (Chapter 4) and the CPU (Chapter C). The cost components of Equation 3.1 are:

- (Geometric) Jacobian: $2N^2_{dim}N_{bf}$ to compute $J_{i,j} = X_{i,n}\partial_j\phi_n$, where $X_{i,n}$ are the global coordinates of the element's nodes; N^3_{dim} to compute J^{-1} and $det(J)$.
- $\partial_{x_i}\phi_n$: $2N^2_{dim}N_{bf}$.
- $u_k = U_{k,n}\phi_n$: $2N_{sr}N_{bf}$.
- $C(F)$: cost of computing the (continuous) convective flux; little can be done to improve the efficiency of such evaluations, so we will consider them as a "black box." In general, $C(F)$ should only "scale" with physical dimension (linearly) and state rank (at most quadratically).
- Combining into $R_{k,n}$: $2N_{dim}N_{sr}N_{bf}$.

The previous expressions would be valid for isoparametric geometry representations; in other cases, N_{bf} should be replaced by $N_{bf-geom}$. Additionally, the convention used is that $y = y + Ax$, with $A \in \mathcal{R}^{m\times n}$, costs $2mn$ and $z = z + xy^T$, with $x \in \mathcal{R}^{m\times 1}$ and $y \in \mathcal{R}^{n\times 1}$, also costs $2mn$.

Asymptotic notation was avoided in this analysis. For many problems of interest, many of the cost-components of Equation 3.1 are important. First, the limit of large spatial dimension is not interesting: N_{dim} is 1, 2, or 3. Additionally, the limit of large N_{sr} may be interesting (e.g., for multi-species simulations), but for a given problem, it is a fixed quantity. The limit of large N_{bf} is more interesting, but many simulations are not run with P_5 or P_{10}. At lower polynomial orders (e.g., P_0 or P_1), the relative weights of the residual costs changes substantially. Thus, N_{dim}, N_{sr}, and other numeric factors are given to provide intuition for the relative importance of various factors in the residual computation.

Although the element residual $R_{k,n}$ is commonly thought of as a vector, its components are indexed over two indexes, the state rank and the number of basis functions. It is then convenient to think of $R_{k,n}$ as a matrix of size $N_{sr} \times N_{bf}$. Additionally, $R_{k,n}$ should be stored in (k, n) order; i.e., so that the index over basis functions is the faster-changing index. In this way, elemental mass matrices will be block-diagonal (N_{sr} blocks of size $N_{bf} \times N_{bf}$ each), instead of a "scattered" $(N_{sr}N_{bf}) \times (N_{sr}N_{bf})$ matrix with no useful structure. Inverting the block-diagonal mass matrix is a factor of N_{sr}^2 cheaper.

In general, the index ordering in Equation 3.1 (and in the following discussion) is meaningful. It is suggested that DG implementations store residuals, temporary quantities, etc. in the layout implied by the left-to-right ordering of the indexes. So $A_{i,j,k}$ could be declared `A[N_i][N_j][N_k]` in `C`.[1] In the current discussion, we are summing $R_{g,k,n}$ over g, so we can simply accumulate the added results in a single location.

Jump Residual

The basic kernel for faces (i.e., jump terms) involves numerically integrating $\int_{\partial K} \hat{F}_k(\mathbf{u})\phi$:

$$R^L_{g,k,n} = w_g \hat{F}_{g,k} \left(\mathbf{n}^L_g, \mathbf{U^L}_n \phi^L_{g,n}, \mathbf{U^R}_n \phi^R_{g,n} \right) \phi^L_{g,n} \tag{3.2}$$

The superscripts L and R denote the element on the left and right of a face, respectively. The "sidedness" of an element is dependent on the convention for normal vectors. In the following discussion, n^L_i denotes the normal from the left to the right element; $n^R_i = -n^L_i$

[1]The use of multidimensional arrays in `C` is *only* recommended if the arrays are statically sized.

denotes the opposite, right to left normal.

A matching term to Equation 3.2, $R^R_{g,k,n}$ is present for the contribution to the right-side elemental residual. $\hat{F}_k$ refers to the numerical flux (e.g., the Roe Flux) in the face-normal direction.

At a quad point, the total cost of Equation 3.2 is $2N_{dim}^2 N_{bfL} + 3N_{dim}^2 + 4N_{sr}N_{bfL} + C(\hat{F})$. N_{bfL} denotes the number of basis functions interpolating the left element; recall that an isoparametric geometry representation is assumed. For the right face, the cost expression is identical unless the right element's interpolation order is different; i.e., replace N_{bfL} by N_{bfR}. Here, evaluating the numerical fluxes, $\hat{F}$, will be more expensive than the continuous fluxes. At least for the Roe Flux, the cost "scaling" is linear in spatial dimension and at most quadratic in state rank, as with the continuous flux. But the constants involved are much larger. Lastly, the first two terms of the cost expression arise from computing the normal vector. Technically only the basis functions in dimension $N_{dim} - 1$ are needed for the normals; i.e., the face-normals of a 2D element can be found using 1D basis functions.

Galerkin Jacobian

The basic kernel for the Jacobian of Equation (3.1) at a single quadrature point is:

$$\partial_{U_{a,m}} R_{g,k,n} = -w_g \partial_{x_i} \phi_n \partial_{u_a} F_{i,k}(\mathbf{U}_n \phi_{g,n}) \phi_{g,m} \tag{3.3}$$

Note that $\partial_{U_{a,m}} R_{g,k,n}$ should be interpreted as $A_{g,a,m,k,n}$.

The cost per quadrature point is $2N_{dim}N_{sr}^2 N_{bf} + 2N_{dim}N_{sr}^2 N_{bf}^2 + C(\partial_u F)$. Here, costs that are redundant (e.g., J, $\partial_{x_i}\phi_n$, etc.) with the residual evaluation have been skipped. The new terms arise from:

- $\partial_{u_a} F_{i,k} \phi_m$: $2N_{dim}N_{sr}^2 N_{bf}$.
- Combining into $\partial_{U_{a,m}} R_{k,n}$: $2N_{dim}N_{sr}^2 N_{bf}^2$.
- $C(\partial_u F)$: the cost of computing the state derivatives of the continuous flux.

Unsurprisingly, the Jacobian computation is substantially more expensive, with the (generally) dominant term $N_{dim}N_{sr}^2 N_{bf}^2$ being a factor of $N_{sr}N_{bf}$ larger than the largest residual term, $N_{dim}N_{sr}N_{bf}$.

Jump Jacobian

The basic kernel for the Jacobian of Equation (3.2) at a single quadrature point, with respect to the left states, is:

$$\partial_{U^L_{a,m}} R^L_{g,k,n} = w_g \phi^L_n \partial_{u^L_a} \hat{F}_k \left(\mathbf{n}^L_g, \mathbf{U^L}_n \phi^L_{g,n}, \mathbf{U^R}_n \phi^R_{g,n} \right) \tag{3.4}$$

Differentiating with respect to the right states:

$$\partial_{U^R_{a,m}} R^L_{g,k,n} = w_g \phi^L_n \partial_{u^R_a} \hat{F}_k \left(\mathbf{n}^L_g, \mathbf{U^L}_n \phi^L_{g,n}, \mathbf{U^R}_n \phi^R_{g,n} \right) \tag{3.5}$$

Again, analogous expressions exist for the derivatives of $R^R_{g,k,n}$.

At a single quadrature point, the cost of Equation 3.4 is $2N_{sr}^2 N_{bfL} + 2N_{sr}^2 N_{bfL}^2 + C(\partial_{u^L}\hat{F})$; and the cost of Equation 3.5 is $2N_{sr}^2 N_{bfR} + 2N_{sr}^2 N_{bfL} N_{bfR} + C(\partial_{u^R}\hat{F})$. The costs of computing the left and right derivatives of $R^R_{g,k,n}$ are analogous, but not simply doubled, since some terms are shared between the four Jacobians being computed. $C(\partial_u \hat{F})$ denotes the cost of computing the state derivatives of the upwinding operator (e.g., the Roe Flux). Again, the Jacobian cost is a factor of $N_{sr}N_{bf}$ more expensive than the corresponding jump residual terms; not to mention that there are twice times as many Jacobian terms.

3.1.2 Viscous Discretization

The analysis and derivation of the viscous discretization is more complex. The implementation is also more involved than that of the inviscid terms, and the computational cost is greater. But the process still boils down to a combination of tensor products and tensor contractions.

In the following discussion, only the BR2 viscous discretization is considered; furthermore, it is implemented in a dual consistent fashion. More specifically, the following discussion assumes that quantities computed over an element κ do not influence its neighbors.[2] The

[2]Galerkin terms with such "non-local" influence arise when viscous or source terms (involving derivatives)

computation of the BR2 lifting operator is described first, followed by its use in the viscous jump residual and Jacobian terms.

The BR2 Lifting Operator

On any particular left face $\sigma^{f,L}$, finding the lifting operator involves solving the following equation for $r^{f,L}$ and $r^{f,R}$:

$$\int_{\kappa^L} \tau_i^L r_i^{f,L} + \int_{\kappa^R} \tau_i^R r_i^{f,R} = \int_{\sigma_f} \{\tau_i\} [\![u]\!] n_i^L$$

where $\tau^L \in \mathcal{V}_h^p(\kappa^L)$ and $\tau^R \in \mathcal{V}_h^p(\kappa^R)$. The f superscripts are now dropped since the discussion only considers one face at a time. Due to the choice of basis, $\tau_i^L r_i^L$ has support only in κ^L. Now consider the lift-corrected gradient of the conservative state vector, $\partial_{x_i} u_k$. Choosing $\tau^L = \phi_m^L$ and $\tau^R = \phi_m^R$ (pick the same ϕ for all spatial dimensions i), and expanding $r_{k,i} = D_{i,k,n}\phi_n$ in the polynomial basis,

$$\int_{\kappa^L} D_{i,k,n}^L \phi_n^L \phi_m^L = \int_{\sigma_f} \frac{1}{2} \phi_m^L [\![u_k]\!] n_j^L \tag{3.6}$$

$$\int_{\kappa^R} D_{i,k,n}^R \phi_n^R \phi_m^R = \int_{\sigma_f} \frac{1}{2} \phi_m^R [\![u_k]\!] n_j^L \tag{3.7}$$

These are evaluated using numerical quadrature. Since D^L and D^R are constant, the left-hand sides of the equations above can be rewritten in terms of the mass matrix $M_{mn} = \int_\kappa \phi_n \phi_m$:

$$M_{m,n}^L D_{i,k,n}^L = \frac{1}{2} w_g \phi_{g,m}^L \left(u_{g,k}^L - u_{g,k}^R\right) n_{g,i}^L \tag{3.8}$$

$$M_{m,n}^R D_{i,k,n}^R = \frac{1}{2} w_g \phi_{g,m}^R \left(u_{g,k}^L - u_{g,k}^R\right) n_{g,k}^L \tag{3.9}$$

where $u_{g,k}^L = U_{k,o}^L \phi_{g,o}^L$ (similarly for u^R), as before. These equations are easily solved for D^L and D^R by inverting the mass matrix.

When calculating the Jacobian, finding the derivatives of D^L and D^R (e.g., $\partial_{U_{a,m}^L} D_{i,k,n}^L$),

are discretized in an asymptotically dual consistent manner.

is straightforward (and requires little additional calculation) since the equations defining the lifting operator are linear. The derivatives of $r^L_{i,k}$ are then $\partial_{U^L_{a,m}} r^L_{i,k} = \partial_{U^L_{a,m}} D^L_{i,k,n}\phi_n$. Note additionally that the derivatives of D^L and D^R are independent of state; e.g., $\partial_{U^L_{a,m}} D^L_{i,k,n} = M^{-1,L}_{m,n}\left(\frac{1}{2}w_g\phi^L_{g,m}\phi^L_{g,n}n^L_{g,i}\right) \equiv \delta_L D^L_{i,m,n}$ and $\partial_{U^R_{a,m}} D^L_{i,k,n} = -M^{-1,L}_{m,n}\left(\frac{1}{2}w_g\phi^L_{g,m}\phi^R_{g,n}n^L_{g,i}\right) \equiv \delta_R D^L_{i,m,n}$ only need indices i, m, n.

When the derivatives are not needed, evaluating the auxiliary variable coefficients D^L costs $2N_{dim}N_{sr}N_{bfL}N_{quad}+2N_{dim}N_{sr}N^2_{bfL}$. If derivatives are needed, $\delta_L D^L$ costs $2N^2_{bfL}N_{quad}+2N_{dim}N^2_{bfL}N_{quad}$ and $\delta_R D^L$ costs an additional $2N_{dim}N_{bfL}N_{bfR}N_{quad}$. With the derivatives computed, D^L can be computed as before or as $D^L i,k,n = \delta_L D^L_{i,m,n}U^L_{k,m} - \delta_R D^L_{i,m,n}U^R_{k,m}$ at a cost of $4N_{dim}N_{sr}N_{bfL}(N_{bfL}+N_{bfR})$. Note that geometry costs were not counted since these are duplicate with computations that are already necessary for the jump residual or Jacobian. Additionally, it is assumed that the PLU-factorizations of the mass matrices have been precomputed.

Residual Terms

The Galerkin residual term (Equation 3.10) for viscous operators is standard for finite element methods. However the jump contributions from the left and right faces have their numerical traces determined by the BR2 method. The left face term is given below (Equation 3.11); the right-face term is symmetric.

$$\begin{aligned} R_{g,k,n} &= w_g\partial_{x_i}\phi_n\mathcal{A}_{i,j,k,l}\partial_{x_j}u_l && (3.10)\\ R^L_{g,k,n} &= -w_g\partial_{x_i}\phi^L_n\mathcal{A}^L_{i,j,k,l}[\![u_l]\!]n^L_j && (3.11)\\ &\quad -w_g\phi^L_n\left(\mathcal{A}^L_{i,j,k,l}z^{u,L}_{j,l}+\mathcal{A}^R_{i,j,k,l}z^{u,R}_{j,l}\right)n^L_i \end{aligned}$$

where $\partial_{x_j}u_l = U_{k,n}\partial_{x_i}\phi_n$ is the spatial derivative of the states u_l in physical coordinates, and $z^{u,L}_{j,l} = \partial_{x_j}u^L_l - \eta r^L_{j,l}$ denotes the lift-corrected state derivative.

For the Galerkin residual terms, the additional cost per quad point (over the inviscid residual) is $2N_{dim}N_{sr}N_{bf}+2N^2_{dim}N^2_{sr}+C(\mathcal{A})$. $C(\mathcal{A})$ denotes the cost of computing the viscous $\mathcal{A}$ matrix, which is generally substantially more expensive than computing the inviscid flux. Note that an additional cost of $2N_{dim}N_{sr}N_{bf}$ was not incurred because the viscous flux can be added to the inviscid flux directly.

For the jump residual terms on the left face, the additional cost per quad point is $2N_{dim}^3 + 2N_{dim}^2(N_{bfL} + N_{bfR} + 2N_{dim}^2(N_{bfL} + N_{bfR}) + 6N_{dim}N_{sr}(N_{bfL} + N_{bfR}) + 4N_{dim}^2N_{sr}^2 + 4N_{dim}N_{sr}^2 + C(D^L, D^R) + C(\mathcal{A}^L, \mathcal{A}^R)$, where $C(D^L, D^R)$ denotes the cost of evaluating the lifting coefficients and $C(\mathcal{A}^L, \mathcal{A}^R)$ denotes the cost of computing $\mathcal{A}$ on the left and right faces. As before, the right face costs are similar, but not necessarily double due to some quantities being shared.

Jacobian Terms

Again, handling the derivative $\partial_{u_a}\mathcal{A}_{i,j,k,l}$ is nontrivial. The dominant cost components (i.e., $N_{sr}^2N_{bf}^2$) grow in the same way as the inviscid Jacobian terms, which should be clear from the expressions. For completeness, the Galerkin component, $\partial_{U_{a,m}}R_{g,k,n}$, and the (+ face) jump components, $\partial_{U^L_{a,m}}R^L_{g,k,n}$ and $\partial_{U^R_{a,m}}R^L_{g,k,n}$, are given below:

$$\begin{aligned}
\partial_{U_{a,m}}R_{g,k,n} &= w_g\partial_{x_i}\phi_n\left(\partial_{u_a}\mathcal{A}_{i,j,k,l}\partial_j u_l\phi_m + \mathcal{A}_{i,j,k,a}\partial_j\phi_m\right) && (3.12)\\
\partial_{U^L_{a,m}}R^L_{g,k,n} &= -w_g\partial_{x_i}\phi_n\left(\partial_{u^L_a}\mathcal{A}^L_{i,j,k,l}[\![u_l]\!]n^L_j\phi_m + \mathcal{A}^L_{i,j,k,a}n^L_j\phi^L_m\right) && (3.13)\\
&\quad +w_g\phi_n\left(\mathcal{A}^L_{i,j,k,a}z^{\phi,L}_{j,m} + \partial_{u^L_a}\mathcal{A}^L_{i,j,k,l}z^{u,L}_{j,l}\phi^L_m - \eta\mathcal{A}^R_{i,j,k,a}\partial_{U^L_m}r^R_j\right)n^L_i\\
\partial_{U^R_{a,m}}R^L_{g,k,n} &= w_g\partial_{x_i}\phi^L_n\mathcal{A}^L_{i,j,k,a}n^L_j\phi^R_m && (3.14)\\
&\quad -w_g\phi^L_n\left(\mathcal{A}^R_{i,j,k,a}z^{\phi,R}_{j,m} + \partial_{u^R_a}\mathcal{A}^R_{i,j,k,l}z^{u,R}_{j,l}\phi^R_m - \eta\mathcal{A}^L_{i,j,k,a}\partial_{U^R_m}r^L_j\right)n^L_i
\end{aligned}$$

where $z^{\phi,L}_{j,m} = \partial_{x_j}\phi_m - \eta\partial_{U^L_m}r^L_j$ is the lifted basis derivative.

The additional cost per quad point for the Galerkin Jacobian terms is $2N_{dim}^2N_{sr}^3 + N_{dim}N_{sr}^2 + 2N_{dim}^2N_{sr}^2N_{bf} + C(\partial_u\mathcal{A})$, where $C(\partial_u\mathcal{A})$ denotes the cost of evaluating the state derivatives of $\mathcal{A}$.

For $\partial_{U^L_{a,m}}R^L_{g,k,n}$, the additional cost per quad point is $4N_{dim}^2N_{sr}^3+2N_{dim}N_{sr}^2+2N_{dim}N_{sr}^2N_{bfL}+ 2N_{sr}^2N_{bfL}^2 + 2N_{dim}N_{bfL}(N_{bfL} + N_{bfR}) + 4N_{dim}N_{sr}^2N_{bfL} + C(\delta_LD^L, \delta_LD^R) + C(\partial_u\mathcal{A}^L, \partial_u\mathcal{A}^R)$. Evaluating $\partial_{U^R_{a,m}}R^L_{g,k,n}$ incurs an additional $2N_{dim}^2N_{sr}^3 + 2N_{dim}N_{sr}^2N_{bfL} + 2N_{sr}^2N_{bfL}N_{bfR} + 2N_{dim}N_{bfL}(N_{bfL} + N_{bfR}) + N_{dim}N_{sr}^2 + 4N_{dim}N_{sr}^2N_{bfR} + C(\delta_RD^L, \delta_RD^R)$.

3.1.3 Quadrature

So far, costs have only been given per quadrature point. Here, the number of quadrature points needed is discussed. The desired quadrature accuracy depends on the polynomial interpolation order for the solution (p_{sol}) and for the geometry (p_{geom}). A particular quadrature formula is characterized by the highest polynomial order that it can integrate exactly. In this thesis, the requested order is $2(p_{sol}+1)+p_{geom}-1$ for elements; for faces, $2(p_{sol}+2)+p_{geom}-1$ is used.

In 1D, Gaussian Quadrature is common; it requires $(p+1)/2$ points to integrate a polynomial of order p exactly. Extensive listings of 2D and 3D quadrature rules have been published [65, 67]. For simplexes, the number of quadrature points scales as $O((\frac{p+1}{2})^{N_{dim}})$ to integrate a polynomial of order p exactly. Exact counts are usually less than the previous, simple formula, which is an upper bound based on tensor product rules (of Gaussian Quadrature) for rectangular elements. The exponentiation by N_{dim} is important here. The number of quadrature points needed for element interiors is much larger than the number of quadrature points needed for element faces, since faces are one dimension "smaller" than interiors. This is especially true for high polynomial orders. However, the expense per quadrature point on faces is higher due to the need to calculate contributions from and to the left and right elements. Additionally, there are more faces than elements.

3.2 Nonlinear Solve

Each nonlinear (Newton) iteration is composed of two principal parts:

- Assembly of the residual and Jacobian
- Solution of a linear system

The cost of the nonlinear solve is dominated by these two components. Outside of the assembly and the linear solve, the nonlinear iteration only requires $O(N_{elem}N_{sr}N_{bf})$ flops through a series of BLAS-1 operations. Further estimation of the nonlinear solver cost is not really possible, since the number of nonlinear iterations is strongly dependent on the degree of nonlinearity in the problem, mesh resolution, etc. Improving the efficiency of the nonlinear solver is a balance between improving robustness and accelerating convergence. For example, using a very conservative line search algorithm, increasing CFL slowly, using highly-accurate linear solves, etc. will lead to more robustness but the number of nonlinear iterations could increase greatly. However, the heuristics used to control the nonlinear solve are not the focus of this thesis; see [59, 63] for overviews of popular globalization techniques and further references.

3.3 Linear Solve via ILU Preconditioned, Restarted GMRES

In most problems using implicit time-stepping, the linear solve will compose a major portion of the total compute time. In steady problems, the linear solve can make up more than 90% of the total cost. This section will focus on the cost of ILU preconditioned, restarted GMRES. However, the discussion here holds for most iterative linear solvers (especially other Krylov methods), since these methods are composed of a combination of sparse matrix-vector products and dense vector-vector products. In the following discussion, $N_b = N_{sr}N_{bf}$ denotes the block-size. DG methods have the advantage of sparsity patterns that are naturally block-sparse, whereas many other discretizations (e.g., CG) often resort to reorderings to create block structure.

As with the residual and Jacobian assembly, the cost per "inner" GMRES iteration (i.e., building the Krylov basis) can be calculated exactly. However, as with the nonlinear solve, estimating the exact linear solver cost is difficult, since it is almost impossible to know the number of inner iterations a priori. If GMRES is restarted, it is similarly impossible to know the number of restarts (i.e., "outer" iterations) exactly. For (restarted) GMRES, the difficulty of the linear problem is dependent on the conditioning (more accurately, eigenvalue distribution) of the preconditioned Jacobian matrix.

ILU preconditioning is implemented in-place [18]. In-place ILU overwrites the Jacobian matrix with its ILU factorization; more commonly, the ILU factorization is stored separately, doubling the memory requirement. The penalty is greater computational cost,[3] since Ax cannot be computed directly; instead $\tilde{L}\tilde{U}x$ must be computed.

The dominant costs of the MDF reordering are: 1) computing $C_{i,j} = \|A^{-1}_{i,p,i,q}A_{i,p,j,q}\|_F$, where i, j range over N_{elem} and p, q range over N_b; and 2) searching over the list of unordered elements for the maximum weight element and updating the weights of that element's unordered neighbors. The former costs $\frac{2}{3}N_b^3 + 2(N_f + 1)N_b^3$ flops per element; note that lower order terms ($O(N_b^2)$) have been dropped. N_f denotes the number of faces of an element; i.e., the number of neighbors. For searching and updating weights, an asymptotically optimal $O(N_{elem} \log N_{elem})$ comparisons is obtainable with an elementary min-heap data structure;[4] see [16] for example. Methods exist for reducing the big-O constant within the binary heap; alternative heap designs with better weight-update performance also exist.

[18] gives flop count estimates for $AM^{-1}x$ kernel in “bare” ILU-GMRES without reordering and ILU-GMRES with MDF reordering. $AM^{-1}x$ is computed once in each inner iteration. Again, note that the ILU factorization is computed in-place. Here, M denotes the preconditioning matrix, leading to the decomposition $A = M + N$. It is more efficient to compute $AM^{-1}x = x + NM^{-1}x$. The asymptotic cost of computing the ILU factorization is similar to that of the MDF reordering: $\frac{2}{3}N_b^3 + 2(N_f + 1)N_b^3$ flops per element. In practice, MDF is slightly more expensive due to the $O(N_{elem} \log N_{elem})$ term. Computing $M^{-1}x$ costs $2(N_f + 1)N_b^2$ flops per element. Lastly, [18] estimates $\frac{2}{3}(N_f + 4)(N_f - 1)N_b^2$ flops per element for computing Nx when no reordering is used.[5] Experimentally, this estimate is too high. With MDF reordering,[6] the estimate is reduced to $\frac{2}{3}(N_f + 4)(N_f - 2)N_b^2$, which is reasonably accurate in practice[18].

The last major cost-component of GMRES is the Arnoldi iteration, excluding the cost of $AM^{-1}x$. This cost is difficult to predict unless the number of inner iterations is fixed. Let N_{inner} be the number of inner iterations and denote a restarted GMRES method by

[3]More flops are required, but the amount of memory accessed is roughly equal. Thus in practice, the in-place method is often faster than its full-storage counterpart.

[4]Brute force searches can require $O(N_{elem}^2)$ comparisons or worse, which can be a substantial cost with as few as 10^4 elements.

[5]Only an estimate is possible here, since the cost of Nx depends on the number of nonzero blocks in N, which is case-dependent.

[6][18] produces an argument for “Line” reordering which reorders elements along lines of strong convective coupling. But for convection-dominated flows, MDF produces similar results.

GMRES(N_{inner}). [61] recommends the Modified Gram-Schmidt (MGS) process for orthogonalizing the Krylov basis, which will be discussed here. Alternatives are covered in Section 5.2. The i-th step of MGS costs $4N_bN_{elem}(i+1)$ flops. With N_{inner} steps, the total cost amounts to $4N_bN_{elem}\left(\frac{N_{inner}(N_{inner}+1)}{2}+N_{inner}\right)$ flops. Lastly, the amortized cost of solving the least-squares problem arising in GMRES is very minor if the Hessenberg matrix generated by the Arnoldi process is QR-factored incrementally [62].

The previous estimate for Arnoldi can be somewhat misleading. First, after each inner iteration, GMRES produces the current residual norm should an update were taken immediately. Thus, the last outer iteration may not have N_{inner} inner iterations; particularly if only 1 outer iteration occurs, the previous MGS estimate could be very high. Additionally, there are many reasons not to have a constant N_{inner}. If a problem could be solved with 10 (outer) iterations of GMRES(100) or 1 iteration of GMRES(1000), the former would be preferred since the cost of its Arnoldi process is cheaper by a factor of 100. However, cases arise where GMRES(100) stagnates and potentially fails to converge. [9] proposes a simple, effective strategy for varying N_{inner} in a user-specified range.

Restarted GMRES implementations also need to deal with stagnation. When GMRES(N_{inner}) stagnates, convergence can slow substantially or even become impossible. A number of heuristics exist to detect stagnation and recover from it. Recovery entails increasing N_{inner} to enrich the Krylov space. [31] discusses a method for dealing with stagnation in restarted GMRES.

Finally, as demonstrated in Section 3.5, optimized BLAS implementations are often poor choices for computing the block-wise matrix-vector products required by GMRES. For many relevant block sizes, compiler optimized versions of a basic matrix-vector product routine outperform the BLAS. Benchmarking is required to determine when BLAS is a better choice for performance.

3.4 Example Problems

This section exhibits one unsteady and three steady example problems. In all cases, the nonlinear solver terminated when the nonlinear residual norm fell below 10^{-12}. Except where noted, the GMRES solver used at most 200 Krylov vectors per restart. The GMRES stopping criterion is the "fixed" criterion described in Appendix D, with $K = 10^{-3}$. This choice is discussed later in this section. All cases used ILU preconditioning with MDF reordering. In steady cases, the nonlinear solver starts with $CFL = 1$ and increases by a factor of 5 with each successful iteration. Note that all steady cases were initialized to converged solutions with interpolation order one less than the reported order; i.e., a P_3 solution starts from a converged P_2 solution. The component-costs are summarized in Tables 3.5, 3.6, and 3.7.

- Case 1: steady, inviscid flow over a Gaussian bump in 2D. The mesh is isotropic and structured, composed of triangular elements generated from a conformal mapping of a rectangle.

Geometry	Guassian Bump
Mach Number	0.20
Angle of Attack	0.0
Spatial Dimension	2
Equation	Euler
Solution Interpolation Order	3
Geometry Interpolation Order	5
Block Size	40
nElement	20480
DOF	819200
Jacobian Size	0.73GB

Table 3.1: Case 1 description

- Case 2: steady, viscous flow over a flat plate in 3D. The mesh is extruded from a 2D, anisotropic, structured mesh with boundary layer packing and element packing toward the leading edge singularity. The 2D mesh has 936 elements. The extruded mesh has 3 layers of elements in the z direction, with each extruded prism being divided into 3 tetrahedrons.

- Case 3: steady, viscous (Reynolds-Averaged) flow over an RAE 2822 airfoil. The RANS Equations are closed with the Spalart-Allmaras turbulence model. The mesh is generated from repeated iterations of a drag adjoint-based mesh adaptation strategy;

Geometry	Flat Plate
Reynolds Number	10^6
Mach Number	0.25
Angle of Attack	0.0
Spatial Dimension	3
Equation	Navier-Stokes
Solution Interpolation Order	2
Geometry Interpolation Order	1
Block Size	50
nElement	8424
DOF	421200
Jacobian Size	0.63GB

Table 3.2: Case 2 description

anisotropy was detected via the local Mach number. The mesh is anisotropic and unstructured. The RANS source terms were evaluated in a dual consistent fashion, following [57].

Geometry	RAE 2822
Reynolds Number	6.5×10^6
Mach Number	0.3
Angle of Attack	2.31
Spatial Dimension	2
Equation	RANS-SA
Solution Interpolation Order	3
Geometry Interpolation Order	3
Block Size	50
nElement	12889
DOF	644450
Jacobian Size	0.72GB

Table 3.3: Case 3 description

- Case 4a, 4b: unsteady, viscous flow over a NACA 0012 airfoil. As with Case 3, the mesh was generated form repeated iterations of a drag adjoint-based mesh adaptation scheme; the adaptation was isotropic in this case. The unsteady solution is periodic, orbiting toward a limit cycle. The adaptation occurred about the stationary point of the solution. In the unsteady solve, the initial condition was a snapshot of the limit cycle. Problem sizes for these unsteady cases were smaller to compensate for the need for comparatively large numbers of time steps. In order to run simulations with thousands (or more) of time steps, the cost per time-step must be kept low. One small (4a) and one large (4b) unsteady problem are shown.

Geometry	NACA 0012
Δt	0.1
Reynolds Number	1500
Mach Number	0.5
Angle of Attack	9.0
Spatial Dimension	2
Equation	Navier-Stokes
Solution Interpolation Order	3
Geometry Interpolation Order	3
Block Size	50
Case 4a	
nElement	2158
DOF	86320
Jacobian Size	0.08GB
Case 4b	
nElement	8632
DOF	345280
Jacobian Size	0.31GB

Table 3.4: Case 4 description

Case	Res Galer	Res IFace	Res BFace	Jac Galer	Jac IFace	Jac BFace
1	0.44	0.29	0.00	1.57	1.91	0.01
2	0.22	1.17	0.13	1.29	6.87	1.29
3	0.71	1.27	0.02	2.91	5.89	0.16
4a	0.02	0.07	0.01	0.11	0.47	0.02
4b	0.09	0.32	0.01	0.43	1.91	0.05

Table 3.5: Per-evaluation costs (seconds) of the residual (Res) and Jacobian (Jac) assembly for the Galerkin (Galer), Interior Face (IFace), and Boundary Face (BFace) terms.

Case	N_{inner}	N_{outer}	MDF	ILU	Ax	MGS
1	603	7	17.93	13.79	165.05	212.56
2	1754	18	32.67	18.05	330.73	275.51
3	603	11	34.51	24.78	147.68	132.90
4a	304	21	5.92	4.22	8.06	0.49
4b	800	23	26.22	18.65	89.90	23.39

Table 3.6: Summary of the total linear solver costs: number of Arnoldi iterations and GMRES restarts, along with the time (seconds) spent in MDF reordering, ILU precalculations, $x + NM^{-1}x$ kernel (called Ax), and Modified Gram-Schmidt orthogonalization. Results for 4a and 4b are given for one full time-step; i.e., 4 IRK4 stages. Steady cases report the aggregate for a full solve.

Case	N_{nonlin}	Res/Jac	GMRES	%-GMRES
1	7	28.04	460.84	95.26%
2	13	132.39	820.62	86.10%
3	11	129.89	353.07	73.10%
4a	20	16.06	12.79	44.33%
4b	20	68.65	159.94	69.97%

Table 3.7: Comparison of total residual/Jacobian and GMRES times. N_{nonlin} indicates the number of nonlinear solver iterations required. Again, the unsteady results are only for one time-step. %-GMRES is calculated by $t_{GMRES}/(t_{res} + t_{GMRES})$. This comparison ignores other costs such as mesh (and other) I/O, wall-distance calculation (where needed), solution updates, etc.

Table 3.5 shows the costs of residual and Jacobian evaluation, broken down into contributions from element interiors, interior element faces, and boundary element faces. Case 2 has an abnormally high percentage of residual and Jacobian time spent on boundary faces. Due to memory limitations, the number of elements in the extruded (z) direction is relatively small, producing a larger number of boundary elements. That is, adding even one more layer of elements in the z direction requires substantially fewer elements in the initial 2D mesh, and the 936 element 2D mesh is already very coarse. As a result, the ratio of interior faces to boundary faces is 6.5:1 for Case 2; this ratio is more than 20:1. However, in this case the costs per interior face and per boundary face are similar. Assuming all boundary faces cost the same as interior faces, the total residual and Jacobian time for Case 2 would decrease by 0.2 seconds.

In general, the residual and Jacobian evaluation on faces proved to be the dominant portion of the residual evaluation process, despite the fact that faces have far fewer quadrature points than elements. However, these cases were primarily $p = 3$ solutions. The additional work per quadrature point on faces compared to elements and the ratio of faces to elements contribute at most a constant factor to the overall assembly cost on faces. But N_{quad} scales strongly with p, so at sufficiently high polynomial orders, the elemental Galerkin terms will dominate. In the inviscid case, which does not involve the viscous flux or the lifting operator, the Galerkin and face assembly costs are already close.

The MGS costs in Table 3.6 are substantial for the steady cases (1-3), and using 200 Krylov vectors is admittedly high. However, for many cases, using fewer (20, 50, 100) Krylov vectors resulted in greatly increased cost due to the need for more overall GMRES iterations. Case 3 often failed to converge with a substantially fewer than 200 Krylov vectors. These cases could get away with fewer Krylov vectors if the accuracy request for the nonlinear solver were not as stringent. Additionally, investigating the trade-off between Ax multiplies and MGS orthogonalization was not a focus in this thesis; roughly equalizing the two costs implies the cost is at worst within a factor of 2 of the optimum. That said, algorithms that avoid the long recurrences of GMRES (e.g., QMR or BiCG-Stabilized) may be preferable. In trade, QMR and BiCG-Stabilized require the Ax and $A^T x$ kernels or two Ax kernels in each iteration.

Table 3.7 compares the relative run-times of the residual and Jacobian assembly process to the linear solve via GMRES. The number of nonlinear iterations is included as point of reference, since the residual and Jacobian are evaluated once per nonlinear iteration, and the resulting linear system is solved once per nonlinear iteration. GMRES was usually the dominant component of the overall DG solver cost. This is particularly true for inviscid problems, where the residual assembly is simple compared to the viscous case. Unsurprisingly, the RANS-SA case has the greatest residual component amongst the steady problems, since its assembly process involves viscous terms as well as source terms from the SA model. Additionally, for much smaller cases (e.g., 4a), the residual cost was dominant. Observe that when scaling up the number of unknowns (e.g., compare 4a to 4b), the linear solver cost grows faster than the residual and Jacobian assembly costs.

In current CFD practice, small cases like Case 4a only arise when solving unsteady problems.[7] Certainly for these cases, considerable efficiency could be gained from improving the residual and Jacobian assembly process. And particularly for the viscous steady cases, residual and Jacobian assembly still play a substantial role in the overall cost. However, these timing results are reliant on having reasonable choices for GMRES parameters–particularly the stopping criterion, as discussed in the next section.

GMRES Convergence Criterion

Table 3.8 shows some timing results for the fixed and adaptive GMRES stopping criterion described in Appendix D with varying K_A, K_F values. The timings shown were taken from runs of Case 3 (from Section 3.4), with all parameters except the stopping criterion remaining as previously described. Also shown is a fixed criterion run with $K_F = 10^{-14}$, which represents asking GMRES to solve down to machine-precision. Briefly, the adaptive criterion asks GMRES to reduce the initial linear residual norm by an amount that is sufficient to obtain quadratic convergence in the Newton solver; K is a multiplicative safety factor. The fixed criterion asks GMRES to reduce the initial linear residual norm by a factor of K.

Appendix D mentions a trade-off between nonlinear iterations and GMRES costs, which

[7]Not mentioned here are explicit-time schemes, where residual assembly is clearly the dominant cost.

is clearly represented in Table 3.8. At $K_A = 0.5 \times 10^{-3}$, the adaptive method is converging in as few nonlinear iterations as possible and at a fraction of the cost of always converging to machine precision. However, the cost is still greater than most of the K_F options with the fixed criterion.

The low K_F fixed criterion solves generally performed very poorly; insufficient linear solves resulted in increased nonlinear iterations. For this case, $K_F = 0.5 \times 10^{-3}$ with the fixed criterion performs the best, but for other cases $K = 1.0 \times 10^{-3}$ (fixed) is preferable. Moreover the more accurate choice is more robust.

Type	K	N_{nonlin}	Ax	MGS	Res/Jac Total	GMRES Total
A	1.0×10^{-1}	12	149.00	181.89	216.24	403.46
A	5.0×10^{-2}	12	155.71	189.73	216.81	421.94
A	1.0×10^{-2}	13	206.08	249.48	258.55	556.59
A	5.0×10^{-3}	10	236.65	282.32	177.66	633.90
A	1.0×10^{-3}	10	247.13	296.53	177.85	662.14
A	1.0×10^{-4}	10	305.40	400.87	177.42	852.14
F	1.0×10^{-1}	18	153.43	98.25	325.28	328.45
F	5.0×10^{-2}	25	868.41	1253.94	470.33	2537.19
F	1.0×10^{-2}	15	155.41	127.50	292.04	360.12
F	5.0×10^{-3}	11	114.50	89.23	193.67	260.51
F	1.0×10^{-3}	11	146.99	132.15	193.71	350.79
F	1.0×10^{-4}	10	154.54	145.19	177.60	374.51
F	1.0×10^{-14}	10	1210.06	1885.93	177.74	3672.95

Table 3.8: Comparison of the costs (seconds) of the fixed (F) and adaptive (A) GMRES stopping criterion, with various K choices. All data were gathered from Case 3. These criterion are described fully in Appendix D. "GMRES Total" includes Ax and MGS costs as well as MDF and ILU precomputation costs.

3.5 Performance Limitations

From the example problems, it is clear that the steady-state solver spends the majority of its CPU time using GMRES to solve linear systems. Unfortunately, the programmer's ability to improve the performance of the linear solver is fairly limited. In Section 5.2, some specific techniques will be discussed for improving performance. However, the dominant GMRES costs (Arnoldi and Ax) are strongly memory bound, since the Jacobian matrix can span several GB. Then most optimization techniques have little effect, especially on Ax. On the other hand, the data for residual and Jacobian evaluation can be made to fit entirely in the largest cache level, meaning that there is much greater potential for improvement over basic implementations. Additionally, the residual evaluation process is a large component cost in viscous, unsteady problems with implicit time-stepping, and it is the only major cost in explicit-time solvers. For these reasons, the optimization discussion in this thesis are is most applicable to the residual and Jacobian assembly.

Figure 3-1 provides evidence for the memory boundedness of the Ax kernel in GMRES. The plot shows the performance results from computing a sequence of *different* matrix-vector products, $y^{(i)} = y^{(i)} + A^{(i)}x^{(i)}$, with $x^{(i)}, y^{(i)} \in \mathcal{R}^m$ and $A^{(i)} \in \mathcal{R}^{m \times m}$, for various values of m. Regardless of matrix size, the working set was 1GB; i.e., all vectors $x^{(i)}, y^{(i)}$ and matrices $A^{(i)}$ fit tightly in 1GB of memory. The $A^{(i)}$ correspond to the $m \times m$ blocks of the Jacobian matrix. Suppose there are N such triplets of $y^{(i)}, x^{(i)}, A^{(i)}$. The testing code creates a random permutation (over $0 \ldots N-1$ taken uniformly at random with removal). Then it computes numerous matrix-vector products by iterating over the N blocks using the permutation; each set N products covers the full 1GB of memory. This process simulates multiplying the Jacobian matrix by a vector. The random permutation is taken to simulate the effect of using a reordering such as MDF.[8] This Ax "simulation" procedure and the $AM^{-1}x$ kernel in GMRES are memory bound because the entire 1GB of RAM is accessed before starting over. Since typical CPU caches are less than 10MB (see Section 4.1), nothing can be saved in the fast CPU caches. As shown in the introduction, floating point computation is much faster than memory access, thus the performance of the Ax kernel is (to first order) bound by how quickly 1GB of RAM can be read.

"Basic" refers to the most direct (column-major) matrix-vector product implementation:

```
t = 0.0, ixn=0;
for(i=0; i<m; i++){ //m columns
```

[8]Randomness is not important here; rather, the simulation needed to emulate the fact that the Ax kernel in GMRES will almost never access A sequentially if reordering and/or preconditioning are present.

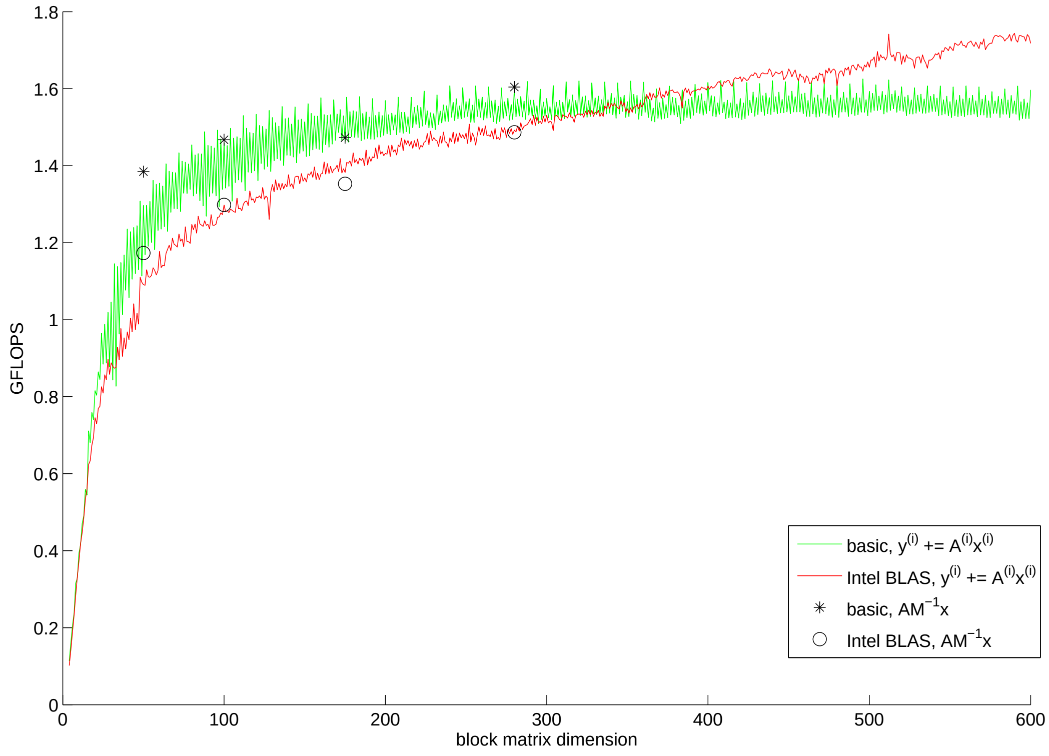

Figure 3-1: The maximum floating point throughput for the $AM^{-1}x$ (from ProjectX) and simulated Ax kernel in GMRES is limited for a wide range of block sizes. The optimized BLAS (MKL) only pulls ahead for very large blocks. The lines indicate results of the $y^{(i)} = y^{(i)} + A^{(i)}x^{(i)}$ program described previously. The individual markers indicate floating point performance measured by the ProjectX implementation of the $AM^{-1}x$ kernel in GMRES. These cases were run on a 3.0GHZ Intel Core 2 processor, on which the maximum floating point throughput for matrix-vector products is around 6 GFLOPS (see Figure 3-2).

```
  t = u[i];
  for(j=0; j<n; j++) //n rows
    y[j] += A[ixn+k]*t;
  ixn += n;
}
```

Note that the "basic" code is compiler-optimized; performance would be much worse if the same simple routine were implemented without optimizations. "BLAS" refers to calls to the BLAS-2 function `dgemv` as implemented in the Intel MKL. The testing code was compiled with `icc`, except where noted. Further results with different BLAS implementations and compilers are shown further in this section. ProjectX was also compiled using `icc`. The specific governing equations (controlling N_{sr}), polynomial order (controlling N_{bf}), and spatial dimension (controlling the number of nonzero blocks per row and N_{bf}) are not important. Results did not vary with these quantities, as long as the final block size, $(N_{sr}N_{bf})^2$, and the total Jacobian size (in GB) remained constant.[9]

For reference, the block sizes arising from common problems are shown in Table 3.9. The first two rows show N_{bf} for Lagrange basis functions over a simplex. The first column indicates N_{sr} for the given equation, with and without PDE-based shock capturing[11]. Observe that only problems with many states and high p even approach the regime where the Intel BLAS is faster than the compiler-optimized, basic implementation; see Figure 3-1. However, in multi-species flows (e.g., chemically reacting flows), the number of states can be quite large. For example, the multi-species, 3D Navier-Stokes equations with 30 states and $p = 3$ has a block size of 600.

First, note that the simple simulation code accurately predicts the performance of the $MN^{-1}x$ operation (called $AM^{-1}x$ for simplicity) in ProjectX. Both the simulation and ProjectX have the same memory-limited behavior. From Figure 3-1, it is clear that the basic routine quickly hits a performance ceiling of around 1.5 GFLOPS. The Intel MKL surpasses 1.5 GFLOPS, but only for very large block sizes. In understanding why the complex and

[9]The total Jacobian size matters to the extent that large portions of it cannot fit in cache simultaneously; e.g., a problem with a 6MB Jacobian will have a flop count that is several times larger than a problem with a 600MB Jacobian.

Equations	$p = 0$	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 5$
2D Interpolant	1	3	6	10	15	21
3D Interpolant	1	4	10	20	35	56
2D Euler	4/5	12/15	24/30	40/50	60/75	84/105
3D Euler	5/6	20/24	50/60	100/120	175/210	280/336
2D NS	4/5	12/15	24/30	40/50	60/75	84/105
3D NS	5/6	20/24	50/60	100/120	175/210	280/336
2D RANS-SA	5/6	15/18	30/36	50/60	75/90	105/126
3D RANS-SA	6/7	24/28	60/70	120/140	210/245	336/392

Table 3.9: Block sizes for the 2D and 3D Euler, Navier-Stokes (NS) and RANS-SA equations. The notation in cells beyond the first two rows is A/B, where A is the block-size without shock-capturing, and B is the block-size with PDE-based shock-capturing[11]. The first two rows show N_{bf} for 2D and 3D Lagrange basis functions over simplexes. Since $N_{bf} = 1$ in the first column, this also indicates N_{sr} for the given equation.

heavily optimized MKL code falls behind the compiler-optimized basic routine, the first step is to examine the peak performance of the matrix-vector multiply routine in an environment that is not memory bound.

As a comparison, Figure 3-2 plots the performance results for computing a sequence of "isolated" matrix-vector products, $y = y + Ax$ (note the lack of superscripts), with $x, y \in \mathcal{R}^m$ and $A \in \mathcal{R}^{m \times m}$. Here, the working set is $m^2 + 2m$. All problems fit entirely within L2 cache; i.e., the workloads are not memory-bound. The maximum performance is around 6 GFLOPS using an optimized BLAS implementation; this is near the maximum possible performance for the processor used. The `icc` optimized "basic" routine performance peaks around 3 GFLOPS. The basic routine lacks optimizations that improve memory access patterns between cache levels and between cache and registers. Also absent are techniques such as loop unrolling, prefetching, and simple offset array access, which are discussed in this thesis. While sparse linear solvers do not operate in this regime, the residual and Jacobian assembly processes do. Unfortunately, not all components of residual evaluation can be written in terms of calls to optimized BLAS functions; but there is substantial room for improvement over basic implementations.

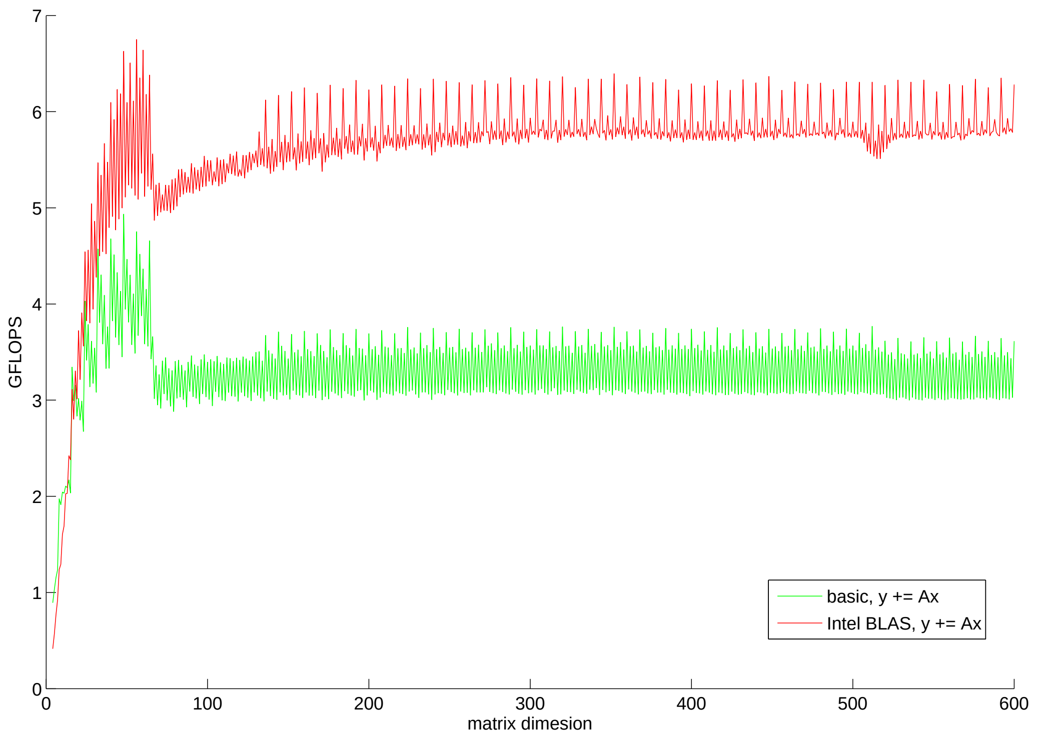

Figure 3-2: The maximum floating point throughput for “isolated” matrix-vector products. The optimized BLAS (MKL) is faster than the basic implementation beyond $m \approx 10$. This case was run on a 3.0GHZ Intel Core 2 processor. The figure indicates that the best possible floating point throughput for matrix-vector products is around 6 GFLOPS for this processor.

In the isolated case, the MKL performance was about a factor of 2 better than the basic routine, unlike in the memory-bound case shown in Figure 3-1. Compared to the isolated case, the peak performance of the basic routine is about a factor of 2 worse than the memory bound case; for the MKL implementation, the difference is more than a factor of 4. Even without prior knowledge of how optimized BLAS implementations work, it should be clear that the optimized BLAS code for a matrix-vector product performs much better when the working set is not memory-bound. Unfortunately BLAS performance is potentially lacking in memory-bound workloads, until the problem size becomes large enough. The following discussion gives some insight into why Figure 3-1 appears as it does. However, the discussion will remain at a high level, since the memory hierarchy (Chapter 4) and the CPU (Appendix C) have not been discussed yet.

In addition to more common techniques like vectorization (Appendix E), loop unrolling (Section C.2), software prefetching (Section 4.1), and several other optimizations), optimized BLAS routines usually employ the blocking technique described in Section 4.1. For now, know that the blocking technique reduces accesses to the vectors $y^{(i)}$ and $x^{(i)}$.[10] However, any matrix-vector algorithm must read all entries of $A^{(i)}$ from memory. Unfortunately the blocking technique can do little to reduce this primary cost. On memory-bound workloads, the BLAS optimizations which are helpful in Figure 3-2 are instead harmful to performance. Many of these techniques incur execution overhead and increase the overall code size, leading to slow-downs. As the problem size increases, the optimized BLAS performance grows. For example, on larger problems, the absolute cost of re-reading $y^{(i)}$ and $x^{(i)}$ is larger. The amount of computation grows like the square of problem-size, offsetting overhead (from optimizations) that scale linearly.

For BLAS 1 and BLAS 2 (shown here) functions, optimized BLAS implementations like the Intel MKL contain a great deal of heavily tuned code that give them unmatched performance when the workload is not memory-bound. In such cases, the BLAS performance degrades substantially due to the extra overhead in running optimizations that are doing little to help. The basic routine is much simpler, and although its performance ceiling is lower, the basic routine reaches that ceiling on much smaller problems.

[10]The basic routine given above reads all of $y^{(i)}$ m times and $x^{(i)}$ once.

The story changes for BLAS 3 operations. Section 4.1 shows that when compared to basic code, blocking asymptotically reduces the amount of communication between levels of the memory hierarchy for a matrix-matrix product. Blocking on BLAS 1 and BLAS 2 routines will at best reduce communication by a constant factor. Thus, when comparing BLAS to basic on computing $C^{(i)} = C^{(i)} + A^{(i)}B^{(i)}$, expect the BLAS to outperform the basic routine even for small matrices.

Note that the compiler (and BLAS implementation) is important. For example, the `gcc` compiled version of the basic routine was generally at least 10% slower than `icc`'s result for the same code. The ATLAS BLAS was often more than 40% slower than the MKL. Figure 3-3 displays the performance difference between different compilers (`gcc` and `icc`) and different optimized-BLAS implementations (ATLAS and MKL), along with the importance of SSE (vectorized) instructions.

Finally, as mentioned in the figure captions, these results were run on a (45nm) Core 2 Duo at 3.0GHZ, with 4GB of RAM. They were mostly compiled with `icc` version 11.1 and `gcc` version 4.4.3 where `gcc` was used. The results shown here are subject to change on other CPUs and with other compiler versions. However, the point remains that there is often overhead associated with using optimized BLAS libraries, and the regimes where BLAS performance is not the best may be quite large. Testing (or auto-tuning) will be required to determine the best settings for each hardware and compiler combination.

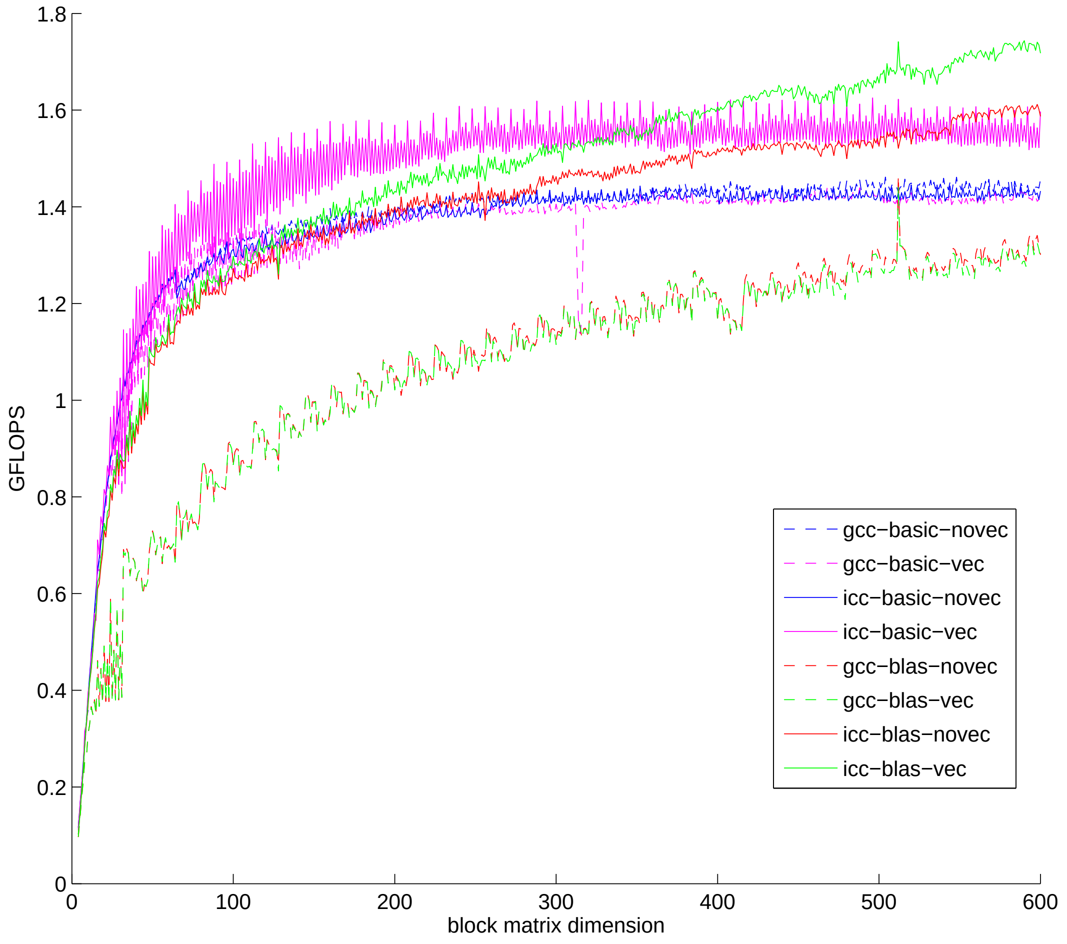

Figure 3-3: The floating point throughput for the simulated Ax kernel in GMRES under various compilers and optimizations. The code running is the $y^{(i)} = y^{(i)} + A^{(i)}x^{(i)}$ computation described at the beginning of this section.

Chapter 4

The Memory Hierarchy

Currently, commodity computers typically possess a central processing unit (CPU) clocked in the range of a few GHZ, a few gigabytes of random access memory (RAM), and a few terabytes of hard disk (HD) space. The notion of a CPU that executes instructions and RAM that stores both data and instructions is called the "Von Neumann Architecture"; it revolutionized digital computer design in the 1940s.

More than twenty years ago, when the x86 family of architectures was created with the Intel 8086, CPU and main memory speeds were roughly equal. In this case, the latency of various arithmetic operations were roughly equal to those of memory accesses. At that time, it was reasonable to account arithmetic operations and memory operations equally as one unit. However, on modern CPUs, arithmetic and memory operations have similar cost only when the data being manipulated is extremely small. Otherwise, that assumption is no longer valid; in particular, memory operations far outweigh the cost of arithmetic computations–especially in scientific computing applications. Reading and writing from RAM takes around 30 times longer than the time needed to multiply two `double` values.

In particular, as computer technology has developed, the rate of growth in CPU performance has far outstripped the rate of growth in RAM performance.[1] Specifically, since the mid-1990s, CPU complexity has been growing at about 60% per year, compared to about

[1]More specifically, SRAM (Static RAM) has kept pace with CPU speeds, whereas DRAM (Dynamic RAM) has not. SRAM is more or less what makes up CPU caches and DRAM makes up the main system memory. However the latter is thousands of times cheaper to produce and power; hence its commonality. Note that the common abbreviation SDRAM indicates Synchronous DRAM, not static.

10% for DRAM[4]. As CPU speeds began to outstrip memory latency, designers realized a need for faster methods to bridge the gap between the two, and CPU caches were born.

Originally, main memory served as a way of bypassing disk accesses. The most often used data would be loaded from disk to memory once, and execution could continue without the hindrance of communicating with the HD. As memory became too slow for constant access, caches served to bridge the new gap. In fact, modern CPUs have as many as 3 levels of cache. The smallest and fastest is enumerated L1 (highest), and the largest and slowest is called L3 (lowest).[2]

In 1988, Aggarwal et al.[2] developed the External Memory Model, which attempts to formalize the notion of a cache. The model only considers a two-level cache hierarchy; e.g., a relatively small and fast cache linking the CPU to a chunk of relatively large and slow main memory as shown in Figure 4-1. It only considers the cost of memory transfers and provides a useful lower bound on performance scaling. Saving the details for later, know that the External Memory Model can be applied between any adjacent cache levels (e.g., L1 and L2), not just cache and main memory.

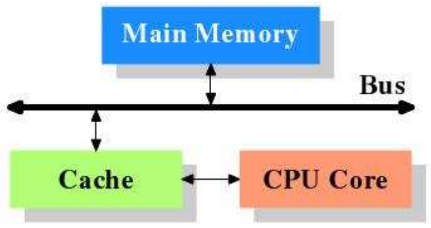

Figure 4-1: A simple 2 level cache hierarchy[20].

Still, many high-level details remain about how caches work and how programs interact with them.

4.1 CPU Caches

Tables 4.1 to 4.3 list some important cache features of the latest processors. Note that the latencies are empirically determined and vary by situation. This is particularly true of the

[2]In a sense, RAM could be considered to be L4 and the HD to be L5 cache.

memory latencies, which vary widely with the type and speed of memory used along with the current memory state.[3] Memory manufacturers advertise their products with information about latencies and bandwdith; see Appendix B for details.

Cache	Type	Size	Line Size	Set Associativity	Latency (CPU cycles)
L1 Instruction	inclusive, per core	32KB	64B	8-way	3
L1 Data	inclusive, per core	32KB	64B	8-way	3
L2	inclusive, per 2 cores	6MB	64B	24-way	15
RAM			64B		≈ 200

Table 4.1: Core 2 (Penryn) cache characteristics. Note that 2MB/4MB, 16-way L2 caches also exist.

Cache	Type	Size	Line Size	Set Associativity	Latency (CPU cycles)
L1 Instruction	Non-exclusive, per core	32KB	64B	8-way	4
L1 Data	inclusive, per core	32KB	64B	8-way	4
L2	inclusive, per core	256KB	64B	8-way	11
L3	inclusive, per 4 cores	8MB	64B	16-way	38
RAM			64B		≈ 100

Table 4.2: Core i7 (Nehalem) cache characteristics.

The CPU has some number of registers[4] that hold operands for use by the various execution units; e.g., `add` the integer in register `rdx` to the integer in `rcx`. If the operands are not

[3]Roughly speaking, a best-case of around 30ns and a worst-case of around 90ns is reasonable.

[4]The x86 standard specifies 8 integer registers (x86-64 expands that to 16), but modern CPUs have at least a hundred, accessed indirectly through "register renaming"; see Section C.4

Cache	Type	Size	Line Size	Set Associativity	Latency (CPU cycles)
L1 Instruction	exclusive, per core	64KB	64B	2-way	3
L1 Data	exclusive, per core	64KB	64B	2-way	3
L2	exclusive, per core	512KB	64B	16-way	12
L3	exclusive, per 2, 4, or 6 cores	2-6MB	64B	32-way	40
RAM			64B		≈ 100

Table 4.3: AMD Phenom (K10) cache characteristics.

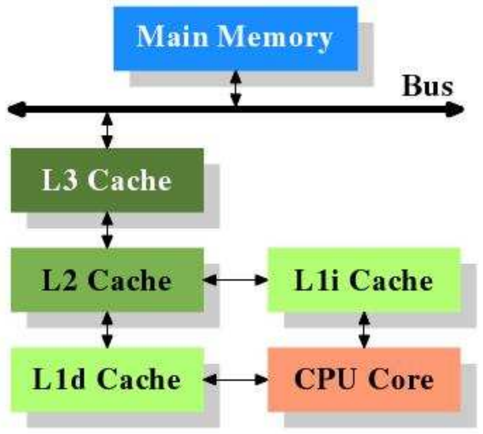

Figure 4-2: Conceptual diagram of the cache layout in Core i7 and AMD K10 processors[20].

in registers, then the CPU must retrieve them from the memory hierarchy and the processor may become *stalled*. That is, the execution must wait until the necessary data is retrieved.

To quantify the benefits of cache on execution time, consider computing the matrix-vector product $y = Ax$ 100 times, where $A \in \mathcal{R}^{100\times 100}$. Suppose that it takes about 200 cycles to read data from main memory and about 10 cycles to read data from L2 cache. Without caching and ignoring accesses to y and x, about 2×10^9 cycles would be spent on memory operations. But if A were sitting in L2 cache, that figure decreases to only about 1×10^8 cycles: a 95% improvement. Considering that x87 or SSE2 floating point operations take at most 5 cycles, it is clear that BLAS-1 and BLAS-2 operations are memory limited. Justifying a 200 cycle wait period requires a fairly tremendous amount of computation (per element loaded from memory), much more work than typically arises in scientific computing applications.

In short, caches are meant to keep "certain data" in a place that is quickly accessible by the CPU. Since interacting with cache and memory stalls CPU cycles, these reads and stores should be performed as infrequently as possible or in parallel with arithmetic computations so that time is not wasted waiting. A *cache hit* occurs when the CPU desires data already in cache; oppositely if the data is not in cache (and must be loaded from RAM), it is called a *cache miss*. In an ideal sense, "certain data" means that the cache should always have the data that the CPU requires. Since the cache size is very small compared to the RAM size, data will have to be loaded and evicted to keep the most relevant elements in cache.

Minimizing cache misses across all memory operations spanning the duration of the program is a nearly impossible task. Solving the problem requires the CPU to possess some oracle that knows all future program states without ever evaluating them; such an oracle would know which data to load and which data to evict to minimize cache misses. Instead, modern caches on x86 CPUs[5] use a paging scheme called "Least Recently Used" (LRU) in which the least recently used data are evicted to make space as new data are loaded into cache. Unfortunately, in the worst case, LRU (and in fact, any online replacement scheme) incurs a factor of k more misses than the magical oracle[64], where k is the number of cache lines fitting in cache. However, LRU performs as well as possible without magic, and it

[5]e.g., ARM processors use pseudo-random replacement.

works quite well in practice. Through a combination of aggressive prefetching by the CPU and programmer awareness, it is possible (and always desirable) to avoid situations where LRU is defeated by the oracle.

Data vs Instruction Cache

In Tables 4.1 to 4.3, two types of L1 caches are listed: L1d (L1 data) and L1i (L1 instruction).[6] Data are the operands passed to arithmetic operations, function calls, etc. Instructions are the operators; e.g., jump, load, store, add, multiply, xor, etc. The compiler converts the `C` code to assembly, which has a one-to-one correspondence with binary machine language. The binary language is loaded into RAM and executed by the CPU. Just as with manipulating bytes of data in RAM, instructions being executed by the CPU also have to be loaded into cache.

The L2 (and L3) caches contain both data and instructions. Modern CPUs separate them at the L1 level because relatively small amounts of code can step through large amounts of data (e.g., matrix-matrix product). Separating instructions and data avoids the immense slow-downs that would occur if every instruction had to be fetched from RAM or even lower cache levels.

In addition, some upcoming processors (e.g., Intel's "Sandy Bridge" architecture) have an additional cache(s) for instructions. Sandy Bridge uses a "L0" cache (sitting between the L1i and the CPU) that caches recently executed μops.

Exclusivity and Shared-ness

The shared-ness of a cache indicates how many processing cores have direct access to all of its contents. A non-shared cache can only be manipulated by a single core.[7] But a shared cache can be manipulated by several cores simultaneously. On modern CPUs, usually only the lowest (i.e., L2 or L3) cache level is shared.

[6]Although the discussion of caching concerned only data thus far, nothing has been specifically tied to the L1d; data simply meant any collection of bytes.

[7]In multicore environments, other cores can "see" non-shared caches, but they do not load or store directly; see Section 4.2.

Exclusivity concerns whether the various cache levels contain duplicate data. In an exclusive cache system (used by AMD), each cache level duplicates no data in the other cache levels. In an inclusive cache (used by Intel), all data in L1 is also in L2 and all data in L2 is also in L3. (In all cases, cached bytes duplicate those in RAM.) Exclusive caches increase the amount of cache available, but their implementation is more complex. The extra complexity incurs additional costs in the form of increased latency, decreased associativity, greater circuit complexity (costs more to manufacture and consumes more power), etc.

So for AMD processors, Figure 4-2 would be somewhat different. Specifically, the connections between L1d/L2, L1i/L2, and L2/L3 would be directed, only moving up the hierarchy; and there would be an additional directed (read-only) edge between memory and the L1 caches.

Cache Lines

Caches are further sub-divided into *cache lines*. As shown above, most current CPUs use a line size of 64 bytes. But this can range from 8B to 1024B. A cache line is the unit of memory that the cache manipulates. Consider an array `double A[1000]`, and then read `A[0]`. If it is not already present, the L1 cache will load `A[0]` through `A[7]`, supposing that `A[0]` sits at the beginning of a cache line. Similarly, stores happen one line at a time. Again, the atomic unit for all load and store operations between various cache levels and between cache and RAM is the cache line.

The existence of cache-lines is a practical consideration. Although RAM is byte-addressable, allowing cache to also be byte-addressable would increase the complexity (and thus cost) of the circuitry in the CPU used to determine what addresses are currently in cache. As a compromise, caches can only address blocks of memory at a time. Additionally, modern RAM does not treat all accesses equally (see Appendix B); specifically, sequential accesses are much faster. Having a cache line as the atomic memory unit encourages more sequential operations.

Cache-lines have an important impact on algorithm design. In particular, since requesting one `double` also loads the other 56B on that cache line, *all* of that data should be used as much as possible before loading another cache line. Using the previous example, `A[0]` through `A[7]` should be processed before touching any other parts of the array `A`. Stepping through data in a regular pattern is called *strided* access. Stride refers to the step size taken between accesses; using a unit stride is also called sequential access. Within a cache-lines, elements 0 through 7 may be processed in any order.

Without knowledge of lines, a programmer might always compute the matrix-vector product $y = Ax$ as a sequence of inner products between the rows of A and the vector x. But if `A` is ordered column-major, the dot product version of Ax would require 8 times more loads from RAM. The situation only worsens if `A` is not an array of `double`s.

Drepper[20] has additional examples that demonstrate the substantial impacts of the cache-line system on a Pentium 4 machine, which were obtained experimentally. In particular, suppose that the per-element cost of processing an array in stride is T. Processing an element involved dereferencing a pointer with the address of the next element; in particular, the processing cost is minute compared to the memory cost. Reading only one element per cache-line costs as much as $5T$; skipping two lines at a time costs as much as $15T$; and skipping four lines at a time costs as much as $32T$. Here, at least accesses were still moving forward in RAM; e.g., skipping two lines at a time means processing every 16th element (in an array of `double`). The situation worsens significantly if elements are accessed at (pseudo-)random: such an access pattern costs as much as $45T$ in Drepper's experiment. The worst cases occur when the working set exceeds the largest cache. When the working set fits entirely in L1, the differences are comparatively small (factors of 2 or 3), but strided access remains optimal. Again, given the design of cache-lines, reading and storing data in stride is absolutely critical.[8]

Note that in an earlier example comparing working with cached and uncached data, the line size was ignored. But it would give the same constant factor reduction to both cases, so the conclusion remains unchanged.

[8]In addition to cache concerns, jumping over large chunks of address space can lead to additional problems; see Section 4.3.1.

Set Associativity

Ideally, any piece of data could go anywhere in cache. The problem then is that another, equally large cache-like space would be required just to figure out whether the contents of some memory address are in cache *and* a fast way of searching that table. This is impractical. The solution used in modern CPUs involves limiting the number of cache positions that a given line from RAM could possibly occupy.

As shown in Table 4.1 and Table 4.2, the L1D cache in a Core 2 or Core i7 processor is organized into 512 lines, each of size 64 bytes. This cache is 8-way associative, so each 64 byte line in RAM can only be assigned to one of eight locations. Thus there are $512/8 = 64$ unique sets in this L1D cache. Recall that on a 64-bit CPU running a 64-bit operating system, memory addresses are 64-bits long. The first 6 bits ($2^6 = 64$) identify each of the 64 bytes in a cache line. The next 6 bits identify which L1D set a particular line falls into. Thus cache lines that are integer multiples of $2^{12} = 4096$ bytes apart map into the same sets. 4096B is called the critical stride for an 8-way associative, 32KB cache. There are clearly an enormous number (2^{52}) of 64-byte memory blocks that map into the same L1D cache line, but there is space for only 8 of those 2^{52}. Finally, note that a 1-way associative cache is also called direct-mapped; and a cache able to store any address at any cache location is called ideal or fully-associative.

Since multiple sets of 64B in RAM map into the same cache lines, it is possible for a 8-way associative cache to incur many more misses than its fully-associative counterpart. In the previous example, suppose 9 `int`s are accessed repeatedly, and they are each located 4096B apart from their nearest neighbors in RAM. Then *every* load will incur a cache miss. In the ideal case, only the first 9 loads incur misses; then all 9 `int` are in cache. Misses due to associativity are called *conflict* misses. Conflict misses greatly reduce the effective cache size and must be avoided. They can be avoided by reordering data in memory and/or working with data dimensions that are not high powers of 2; i.e., if the problem size is a high power of 2, pad the data with extra zeros.

Agner[28] carefully examines a common example of conflict misses: computing A^T in place, where $A \in \mathcal{R}^{m\times m}$ has type `double`. Table 4.4 states his timing results for various matrix sizes. Agner ran these experiments on a Pentium 4 processor.[9] Note the timing spikes occurring for matrices that are large powers of 2; these inefficiencies arise purely due to conflict misses, dramatically demonstrating their danger. The slowdowns at $m = 64$ and $m = 128$ are due to conflict misses in the L1d cache, whereas the slowdown at $m = 512$ is due to conflict misses in the L2 cache. Slowdown from conflict misses in the L1d cache are less dramatic than L2 contentions due to hardware prefetching and the ability of out of order execution to mask L1 misses.

Matrix Size	Total Space (KB)	Time per Element (sec)
63×63	31	11.6
64×64	32	16.4
65×65	33	11.8
127×127	126	12.2
128×128	128	17.4
129×129	130	14.4
511×511	2040	38.7
512×512	2048	230.7
513×513	2056	38.1

Table 4.4: Time to transpose a $m \times m$ matrix for various values of m on a Pentium 4[28]. This test demonstrates the impact of conflict misses for problem sizes that are high powers of 2.

Processing Writes and Reads

The discussion of the previous sections covered the layout and characteristics of modern CPU caches. The following describes the events that transpire when the CPU receives instructions to write or read data (or instructions). The discussion will focus on inclusive caches (currently found on Intel CPUs) in a single-core environment (see Section 4.2 for an overview of the additional complexities involved).

The very first time the contents of a memory address are accessed, they must be loaded into cache. This is called a *cold* or *compulsory* miss. Prefetching may be able to help (e.g.,

[9]The Pentium 4 has an 8KB, 4-way associative L1d cache and 512KB, 8-way associative L2 cache. The critical stride for the L1d is 2KB and it is 64KB for the L2 cache.

if the data is being read in a regular pattern and the bus is not saturated), but in general this cost is unavoidable. When the CPU loads X (more accurately, the entire cache line in which X lives), if X is already in a register (or very recently stored), then nothing needs to be done. Otherwise, the CPU will ask the L1d cache whether it has X. With a 16-way associative L1d, the cache logic checks (in parallel) 16 locations for X. If X is present (a cache hit), then the data is forwarded to the appropriate register, and the cache logic notes that X has been recently used. If X misses in cache, then the query passes on to the L2 cache. Regardless of whether X misses or hits in the L1d, the L1d (for say a Core i7) takes 4 cycles to respond. The L2 performs the same check: if X hits, then the L2 passes X back to the L1d (since the caches are inclusive); if X misses, then the query proceeds to L3 or main memory. Once X passes back to the L1d, the L1d must find space (in one of 16 possible slots) for X; thus some older data will be evicted.[10] Loads of instructions work the same way, except the highest cache level is the L1i. Note that misses in the instruction cache are generally more expensive, since the CPU will have no work to do. Misses in the data cache are potentially cheaper, since the CPU can execute instructions that do not depend on the missed data (at least in the out-of-order model).

At a high level, stores are similar to loads, but a few important details remain. First, before a store of X can occur, X must already be in cache.[11] So a store operation can trigger a load. On the surface, it may seem like a write miss is more expensive than a read miss. However modern CPUs have a *store buffer* associated with the CPU's execution units that buffers all outgoing stores. Store buffers exist between registers and the first cache level, and they can be conceptualized as a queue. It serves to increase efficiency in several ways. For one, stores to the same address can be coalesced as long as they occur close to each other in time. Additionally, load requests for data in the store buffer can be returned immediately, bypassing interaction with the memory hierarchy. The store buffer also greatly aids the performance of the out of order engine, since many memory operations are not reordered.

[10]Evict takes a different meaning for exclusive and inclusive caches. In the former case, eviction causes the cache line to be sent down to the next cache level. With inclusive caches, the data moves nowhere since it must already exist in lower cache levels, unless the cache line was modified in a write-back environment; see next paragraph.

[11]With SSE instructions, there is an exception to this rule; see below.

The cache line at the head of the store buffer will be written out when the cache is ready. How the data is handled once it leaves the store buffer depends on the cache properties. The two major options here are a *write-through* (WT) cache and a *write-back* (WB) cache. In a WT cache, the write is reflected in the cache and immediately forwarded to the next-lowest cache level. In a WB cache, the write is only reflected in that cache. The altered cache line will only be reflected in the lower memory levels once the altered line is evicted. WB caches are much faster, but they give rise to complex consistency issues. The latest Intel (Core 2, Core i7) and AMD (K10) processors employ WB caches at all levels. Earlier processors (e.g., pre-Core 2 for Intel) employed WT L1 caches.

A large number of event counters exist for various cache events. For example, `L1D_REPL`, `L1I_MISSES`, and `L2_LINES_IN` are among the most commonly examined events on the Core 2. Examine documentation carefully when using these counters. For example, `L1D_REPL` counts the number of cache lines loaded into the L1d, while `MEM_LOAD_RETIRED.L1D_LINE_MISS` counts the number of misses to the L1d cache. Accesses to the same cache line in rapid succession each count separately in the first event,while filling the missed cache line counts only once in the second event.[12] Additionally, the cache performance counters are not a sufficient measurement on their own, particularly in the L1 cache. If cache misses are not causing a significant delay, then removing them should not be a high priority. On the Core 2, there are additional `RESOURCE_STALLS.*` events for this purpose that measure the number of stalled cycles due to various causes; the most important ones are `RAT_STALLS`, `RESOURCE_STALLS.ROB_FULL`, `RESOURCE_STALLS.RS_FULL`, and `RESOURCE_STALLS.LD_ST`. The first three have to do with out of order execution (see Section C.4), and the last event measures stalls resulting from the filling the load buffers, store buffers, or both. The last event will also be discussed further in Section C.4, since these buffers exist largely to enable out of order execution. Stalled cycles due to these three events, coupled with large numbers of cache misses, indicate a memory-bound workload.

[12]Suppose that a program wants to read values from the same cache line 16 times in rapid succession; e.g., reading all 16 (32-bit) integers on a cache line. If the desired line is in memory, then potentially all 16 reads will miss, since loading a line from memory has high latency. Then `MEM_LOAD_RETIRED.L1D_LINE_MISS` is incremented by 16, whereas `L1D_REPL` is only incremented by 1 when the desired line is loaded into memory. Alternatively, if the desired line is in L2 cache and only 4 read attempts occur before that line is loaded, then the first event increments by 4 while the latter still increments by 1.

It also is worth noting that non-cached writes (`MOVNTI`, `MOVNTQ`, `MOVNTPS`, `MOVNTPD`, `MOVNTDQ`, which write various sizes of data directly to RAM without invoking the cache) exist via SSE instructions.[13] As a rule of thumb, non-cached writes should only be used when the data exceeds roughly $\frac{1}{2}$ of the largest cache level[68, 29]. Cache-limited reads (`PREFETCHNTA`, PREFETCH Non-Temporal Access, meaning that data prefetched with this method will be given the highest priority for eviction by the LRU scheme) are also available through the SSE instruction set.

Prefetching

Prefetching is broken into two categories: hardware and software. Hardware prefetching is handled automatically by the CPU, which can and will prefetch both instructions and data. Software prefetching comes in the form of specific assembly instructions issued by the compiler or the programmer, *requesting*[14] that the CPU prefetch data; programmers and compilers have no way of prefetching instructions. The following discussion focuses only on data prefetching; hardware instruction prefetch is handled later in Section C.3.

Hardware prefetching is triggered by specific events. When cache misses occur and the CPU thinks it can predict upcoming misses, prefetching is initiated. Usually, 2 or more cache misses following a recognized pattern will start prefetching[20]. Hardware prefetching technology remained mostly the same between the Core 2 and Core i7 processors and represent the current industry best practices, so details are only provided for the prefetch mechanism in those two CPUs. Substantial changes are upcoming in Intel's Sandy Bridge and AMD's Bulldozer architectures, but details are not yet available.

Prefetchers are able to automatically prefetch several data streams with different strides for all cache levels[20, 27]. The lower cache levels generally have more advanced prefetching mechanisms, since misses are more costly. Strides can involve both stepping forward and backward through memory. Strides can also be non-unit as long as they are constant; e.g.,

[13]The non-cached write instructions do not interact with the previously discussed store buffer. Instead, some CPUs have a write-combining buffer to coalesce uncacheable writes; the existence and function of these buffers is highly CPU dependent[39, 5].

[14]The CPU may ignore software prefetch commands.

reading every 19th byte.[15] Streams correspond to memory accesses due to reading or writing from different regions of RAM; e.g., in computing $y = Ax$, three streams are involved.

Hardware prefetch also has a few potential disadvantages[36]. A few cache misses are needed before prefetching starts, reducing efficiency; this is more problematic for very short data. Prefetching will also continue past the end array boundaries, causing superfluous loads that unnecessarily burden the cache system. Lastly, another potentially substantial weakness of hardware prefetching is that it cannot cross page boundaries.[16]

Table 4.5 lists and describes currently available prefetch commands.

Name	Purpose	Amount of Data Moved
`MOV`	Move data into a register (which forces the data into cache as well).	1 cache line
`PREFETCHT0`, `PREFETCHT1`, `PREFETCHT2`	Architecture dependent; usually prefetches data into the largest cache	1-2 cache lines
`PREFETCHNTA`	Prefetch data into L2 cache with no intent to reuse it	1-2 cache lines
`PREFETCHW` (AMD only)	Prefetch data into L2 cache with intent to modify it	1-2 cache lines

Table 4.5: The various prefetch assembly commands available on current x86 CPUs. Note that `MOV` is the assembly command for loading and storing data; it is not explicitly a prefetch command. Unlike the other `PREFETCH*` commands, `MOV` cannot be ignored by the CPU.

As a rule of thumb, prefetch data that will be used in computations about 200 cycles[17] in the future[29]. AMD suggests an alternate rule: prefetching 6 to 8 cache lines ahead[68]. Additionally, be careful not to prefetch multiple data that would be loaded onto the same cache line, which would cause superfluous conflict misses. However, these are only general guidelines,[18] and any kind of manual software prefetching will require extensive testing to

[15]While the prefetch mechanism can recognize non-unit strides, efficiency is lost by not utilizing all data on a cache line.

[16]Page boundaries, the TLB, and the virtual memory system are described in Section 4.3.1.

[17]More specifically, approximately the latency of a cache miss in the lowest cache level.

[18]Predicting exactly how and when the CPU will require certain data is virtually impossible due to out of order execution; see Section C.4.

maximize performance. Additionally, code with software prefetching optimized for a certain architecture is unlikely to be successful on a different architecture.

Note that the improper use of prefetch commands will likely hurt performance. Prefetching data too late (i.e., prefetching data already in cache) wastes instructions performing null operations. Prefetching data too early could evict presently useful data from cache, leading to extra capacity or conflict misses. In general, automatic hardware prefeteching coupled with compiler-issued software prefetches are generally sufficiently strong for programmers to simply rely on them; i.e., manually issuing prefetch commands is usually unnecessary. But when the program's memory access pattern is irregular (either temporally or spatially), software prefetch can lead to performance improvements. Irregular spatial patterns occur when accessing array elements with non-constant stride; e.g., navigating an array-backed heap. Irregular temporal access occurs (for example) when indirect addressing and/or additional array accesses are required to find the position and size of a matrix block before performing a matrix product. While the CPU is computing these quantities, the hardware prefetcher will not realize that the block matrix entries are needed for the upcoming matrix product. Thus, prefetching the first several columns of the matrix (for smaller block sizes) will prevent processor stalls when the matrix product is initiated and the data is unavailable.[19] Lastly, unlike hardware prefetch, software prefetch can cross page boundaries; but neither type may cause a page fault.

CPU event counters are additionally useful for evaluating the performance of software and hardware prefetching. For example, `SSE_PRE_EXEC.NTA` and `SSE_PRE_MISS.NTA` describe the number of `PREFETCHNTA` commands executed and the number of commands fetching useful lines, respectively; `LOAD_HIT_PRE` describes the number of useless `PREFETCHNTA` commands– they fetched already-cached data. A large number of unneeded prefetch commands corresponds to either hardware prefetching misinterpreting access patterns or improper use of software prefetches. In the former case, access patterns should be changed to prevent misprediction; in the latter case, the use of software prefetch should be re-examined.

[19]The effect is reduced for larger block sizes, since the potential stall at the start of matrix operations (which is avoided by prefetching) becomes a negligible portion of the overall execution time.

Cache Performance: Locality

Revisiting cache-lines, code that loads a cache-line, processes all the data in that line, and never revisits it again is said to have good *temporal locality* and *spatial locality*. Unfortunately, it is not always possible to produce both of these properties. Temporal locality is the notion that some data (or instructions) are (re-)used for several operations that are very close in time. Spatial locality is the corresponding idea in address space; e.g., `A[0]` and `A[1]` are processed together (as opposed to `A[0]` and `A[1000]`).

Locality is extremely important in maintaining good cache performance. In particular, having good temporal and spatial locality means that the cache will nearly always have the data required for computation available. The performance of the LRU replacement scheme will be nearly optimal. Since data (and instructions) are readily available in cache, very little time will be wasted waiting for loads or stores to complete.

It is easy to see that Drepper's non-unit strided and random array access examples lack both spatial and temporal locality; hence their relative inefficiency. Additionally, linked lists, trees, and other pointer-based data structures typically have relatively poor locality.[20] Code with poor spatial and temporal locality will have numerous *capacity* misses. Capacity misses occur when an element that is needed now was evicted earlier because too much additional data has passed through the cache since the initial load. The program has effectively bitten off more than it can chew, and if possible, code should be re-organized to operate on a smaller set of data so that misses do not occur. Oftentimes, having redundant computation is a worthwhile trade-off for reducing the working set size; profiling should be used to evaluate the trade.

Matrix-matrix multiply is another classic example of the benefits of spatial and temporal locality[28]. Consider the problem of computing $C = AB + C$, where $A, B, C \in \mathcal{R}^{m\times m}$. Assuming that row-major ordering is used, a basic implementation follows:

```
for(i=0; i<m; i++){
  for(j=0; j<m; j++){
```

[20]Except when the heap is managed manually, i.e., all nodes are allocated in a contiguous chunk. This is particularly helpful for linked lists or breadth-first type tree operations.

```
    for(k=0; k<m; k++){
      C[i*m + j] += A[i*m + k]*B[k*m + j];
    }
  }
}
```

Accesses to the output `C` have good temporal and spatial locality. The spatial aspect for `C` is not that significant, especially when m is large: then the cost of accessing m elements of `A` and `B` for the inner product heavily outweigh the cost of accessing one element of `C`. But the temporal locality is helpful: in the inner-most loop, only one element of `C` is accessed. `A` is also accessed in stride. In the inner-most loop, adjacent entries of `A` are used one after the other, giving it good spatial locality.[21] However, `B` has no locality.

Giving `B` the same locality as `A` is achieved by transposing `B` before executing the multiply. This leads to about a 4x speedup on a 3.0GHZ Core 2 on a matrix of 1024×1024 `double`. But there is still plenty of room for improvement, even when only taking cache considerations into account. Currently, for each row of `A`, every element of `B` is accessed. If the matrices are very large, the act of accessing `B` causes capacity misses in all memory operations, nullifying the benefit of caching. To restore locality, a technique called *blocking* is applicable. Blocking to improve cache efficiency is a well-known and well-analyzed technique in the scientific computing community; for example, see [45]. Proceeding, divide the matrices into submatrices such that roughly three such blocks (one from each of A,B,C) fit in cache simultaneously. Determination of the exact block size requires experimentation. Then perform the block-matrix product.[22] Now, cache misses only occur when a new block is fetched. Blocking leads to another factor of 3-4x improvement, and it is one of the most important features of the BLAS.

The code below gives a sample implementation of the blocked matrix multiply algorithm when working with square matrices of size m. Note that this code is only meant to illustrate

[21]For smaller values of m, `A` also has some temporal locality since the entire matrix `B` is navigated before a new row of `A` is read.

[22]For the purposes of cache analysis, any matrix product algorithm will suffice for computing the product of blocks. However, for performance, additional techniques such as loop unrolling, vectorization via SSE instructions, and register-blocking are needed.

ideas; it is neither general nor optimal, and it should not be applied directly. The idea of the algorithm is to divide $m \times m$ matrices into $N_b \times N_b$ blocks, each of size $b \times b$. Then carry out the basic matrix-matrix product routine, multiplying one block-row of A by one block-column of B at a time.

```
for (j = 0; j < m; j += BLOCK){
  for (k = 0; k < m; k += BLOCK){
    for (i = 0; i < m; i+=BLOCK){
      for (ii = i; ii < i+BLOCK; ii++){
         for (jj = j; jj < j+BLOCK; jj++){
            for (kk = k; kk < k+BLOCK; kk++){
              C[ii*m+jj] += A[ii*m+kk]*B[jj*m+kk];
            }
         }
      }
    }
  }
}
```

Here, A is taken in row-major order and B in column-major; if this is not the case, then transpose B, as mentioned above. Note that it is assumed that `m` is a multiple of `BLOCK`. While this assumption may be valid if the code runs on very specifically sized matrices, `m` is generally not a multiple of `BLOCK`. When this happens, the division of matrices into blocks leaves several "fringes" that are of size $p \times q$, where $p, q <$`BLOCK`. Special care is required to handle the fringes efficiently.[23] Additionally, `BLOCK` is left unspecified. The choice of `BLOCK` depends on what level of cache the blocks should fit in, which depends on typical problem sizes and performance bottlenecks of particular cases. In general, matrices can be blocked for registers, L1, L2, or any level of the memory hierarchy. Fast matrix-matrix product implementations (e.g., BLAS) use nested blocking, covering all levels of the hierarchy.

[23]For example, one common optimization is to identify common fringe sizes and write specialized code to handle them quickly. See open source, optimized BLAS implementations like the Goto BLAS for state-of-the-art examples.

The external memory model provides a nice framework for analyzing the amount of memory transfers required for these three matrix-matrix product methods. The results are summarized below:

- Basic: $\Theta(m(m(m/s_L + m))) = \Theta(\frac{m^3}{s_L} + m^3)$
- Transposing B: $\Theta(m(2m^2/s_L)) = \Theta(\frac{2m^3}{s_L})$
- Blocked: $\Theta(\frac{2m^3}{s_L\sqrt{s_C}})$

Here, s_C is the cache size and s_L is the cache line size. The amount of memory communication for the first two routines is straightforward; the argument for the blocked routine follows. In the blocked routine, each block of A and each block of B is read N_b^3 times; recall that $N_b = \frac{m}{b}$.[24] A and B are accessed in stride, so these blocks cost $\frac{b^2}{s_L}$ each. The overall cost is then $2N_b^3\frac{b^2}{s_L} = \frac{2m^3}{bs_L}$. The block size, b, is chosen so that $b^2 = \Theta(s_C)$, giving $\Theta(\frac{2m^3}{s_C s_L})$ overall. Note that all three algorithms incur an additional $O(\frac{2m^2}{s_L})$ for reading and writing the output matrix, C.

Note that by a theorem of Hong and Kung[41] (later strengthened by Irony et al.v[40]), the blocked matrix product is *optimal* in the sense than any data reordering cannot obtain better than an $O(\sqrt{s_C})$ speedup over the basic routine.[25] These results only apply to matrix multiply methods requiring $\Theta(m^3)$ flops; they do not apply to $o(m^3)$ methods like Strassen's or Coppersmith and Winograd. The extension to $o(m^3)$ flops methods is an open problem. Lastly, note that cache-oblivious matrix multiply algorithms[26] exist, but their performance is worse than optimized BLAS routines primarily due to an inability to effectively prefetch.

[24]If b does not divide m, then $\lceil \frac{m}{b} \rceil$ captures (but overestimates) the work required for the fringes.

[25]Here, "basic routine" refers to any algorithm that computes $C_{i,j}$ as a sum of products $A_{i,k}B_{k,j}$.

[26]A cache-oblivious algorithm is one that takes advantage of the memory hierarchy without knowing any of its characteristics. Cache-oblivious algorithms are effectively self-tuning, and the same code would perform reasonably on many architectures. Contrast this with BLAS routines, which involve heavy optimization based on the exact size and latency of the cache system. These characteristics must be determined beforehand, requiring different BLAS implementations for each target architecture.

4.2 A Note about Multicore CPUs

In multicore (or more generally, multithreaded), shared memory environments, the issue of cache efficiency becomes even more muddied. In multicore, shared memory environments, many processing cores can read and write from the same block of memory concurrently. The matter is complicated further by the fact that modern CPUs reorder memory reads and writes (see Section C.4). Cache coherency becomes a significant issue; e.g., if core 0 writes to address X and core 1 reads from X immediately after, what value will core 1 read? A formal definition can be found in[33]; in short, coherency refers to the notion that as soon as a write is issued, *all* copies of that data immediately reflect the update. When core 0 writes to X (in cache), it makes that cache line *dirty*; the cache line remains dirty until it is written back to memory.

Cache coherency is an issue on modern CPUs for two main reasons: 1) the use of WB caches; and 2) the existence of load and store buffers between registers and the first cache level. In both cases, buffering is the issue: a dirty cache line is not written out to the next level of the memory hierarchy until it is evicted (either from cache or from a buffer). Thus cores can easily lose coherency from being unable to see the latest memory state. Disabling these performance-enhancing features is one possibility, but an undesirable one in most situations. Instead, processor manufacturers have chosen to weaken consistency models and add assembly instructions that locally strengthen consistency at the request of the programmer.

Handling cache concurrency issues is a topic of active research, but all current x86 implementations follow the MESI (Modified, Exclusive, Shared, Invalid) protocol[39, 5]). However, as previously discussed, some parts of the load and store process exist outside of the cache hiearchy–load and store buffers in particular. Thus MESI (and other cache coherency protocols) only guarantee the coherency of data once that data has entered cache. Before this point, processor consistency defines the correct behavior. Below, the discussion proceeds to MESI first and then introduces processor consistency. Following the MESI protocol allows us to solve the first issue; the hardware implementation allows cores to *snoop* other cores' caches for data of interest.[27]

As shown in Figure 4-3, the MESI protocol defines four possible states for a line of cache[20, 39] in each core:

- **Modified**: The line is dirty; it only exists in the local core's cache.
- **Exclusive**: The line is clean; it only exists in the local core's cache.
- **Shared**: This line is clean; it may exist in other cores.
- **Invalid**: This line cannot be read, since its value is invalid; i.e., the value does not match the most recent update by another core.

Figure 4-3 also denotes the possible state transitions.[28] Local read and write transitions denote when the local core reads or writes X; remote reads and writes denote when some other core reads or writes X. The cores maintain the correct states by snooping other cores' cache activities. Snooping involves directly monitoring other cores' accesses to cache lines held by the local core.

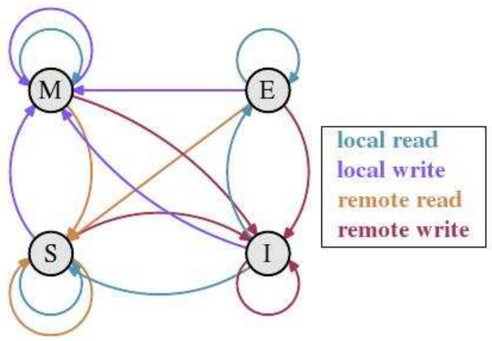

Figure 4-3: The states of the MESI Protocol[20].

Under the MESI protocol, it is possible to generate many more cache misses (called sharing misses) than an equivalent sequential program would incur. That is, some state transitions trigger cache fills, which may unnecessarily load the memory system. For example, if core 0 has cache line X in the modified state and core 1 attempts to read X (not

[27]In caches that write-through to main memory, MESI is not necessary.

[28]One transition is omitted. If a modified line is written out to RAM (e.g., due to eviction), then that line becomes exclusive.

in core 1's cache), then core 0 must send X to core 1 (X is also written to memory). In particular, core 1's actions put extra load on core 0. If core 0 has line X in the shared state and modifies X, then all other cores currently caching X must invalidate their versions; and core 0's version becomes modified. Thus if one such core wants to access X, even though X is already in cache, a long wait will occur wherein X must be transferred from core 0 to the local core. As with single core cache misses, CPUs also have performance counters for sharing misses, e.g., `L1D_CACHE_LOCK.MESI` and `CMP_SNOOP` on the Core 2.

Multicore x86 CPUs follow the (weak) *processor consistency* model. This model is much weaker than strict or sequential consistency (detailed below), which are more intuitive. *Strict consistency* occurs when reading address X fetches the result of the most recent write to X[1]. Strict consistency is a very strong requirement. But the value in X after a write occurs is irrelevant until the next read of X. *Sequential consistency* captures this notion by slightly weakening strict consistency: it allows reads and writes to occur in any order, as long as (serialized) operations occur in *program order*[1]. Program order is the order implied by the code–it is the ordering of instructions resulting from executing code sequentially, one line at a time. Returning to processor consistency, under that model, a sequence writes done by core 0 are received by all other cores in that same order, but different cores may see writes from other cores in different orders. Specifically, if core 0 writes out A,B,C, then no other core will receive that data in a different order (e.g., C,A,B); all cores will receive the data in the order A,B,C. However if core 0 writes A,B,C and core 1 writes D,E,F, then core 2 may see D,A,B,E,C,F while core 3 sees A,B,D,E,F,C. This is often drastically different from sequential consistency.

Programmers often desire sequential consistency, which they can achieve with additional tools, at the cost of performance. Specifically, the assembly commands `LFENCE`, `SFENCE`, and `MFENCE` are available in the x86 instruction set; they force the serialization of code, giving programmers more control over the sequence in which memory loads and stores occur.[29] In particular, the fence commands force all load and store buffers to flush, bring out of order operations back into program order, and generally serialize execution. So after fencing on all threads, every process will see the same memory state. As a result, these commands should

[29]For readers familiar with MPI, these operations are analogous to barriers.

be issued sparingly as they hinder processor performance. Another possibility under various multithreading models (e.g., POSIX) is the use of locks, which only allow one thread to have write access for the duration of the lock. Locks also lead to performance losses and add the additional danger of race conditions. A better approach is to side-step (when possible) the consistency issue by disallowing cores from manipulating the same blocks of memory in a temporally close manner.

The example in Table 4.6 provides a somewhat unsatisfying answer to the question posed in the beginning of this section. That is, if two cores are reading and writing from the same location in memory, what behavior results?

Proc 0	Proc 1
Store X <- 1	Store Y <- 2
Load Y	Load X
Y: 0 or 2	X: 0 or 1

Table 4.6: Assuming X and Y are 0 initially, the processor consistency model used in x86 multicore, shared memory environments leaves the value of Y on Proc 0 and the value of X on Proc 1 indeterminate.

4.3 Paging and Virtual Memory

So far, our discussion has focused on how data (and instructions) pass into and out of the memory system when the CPU makes various requests. The working model has revolved around accessing a specific, physical memory address X (and its associated cache line). However, at least in modern commodity computers (i.e., Intel or AMD machines running Microsoft Windows or Linux/Unix variants), there is another layer of abstraction above physical memory addresses: the virtual address space. Allocating memory to processes in physical space is restrictive; it would be difficult for modern computers to run so many processes simultaneously, because every process would require its own unique block of memory. The advantages of a virtual memory system are not particularly relevant, but its presence adds another layer of complexity to the cache model explained thus far.

Virtual memory allows programs to be designed as if they have sole access to a single, contiguous block of memory. Additionally, “memory” is some abstract hardware device– it may involve several components or only one, but the program should remain unaware. At least for functionality, programs need not worry about whether its assigned block is in memory or somewhere (paged out) on the hard disk, or on some other storage device entirely.[30] As a result, the OS does not allocate physical memory space to the processes running on it.[31] Instead, the OS works with a virtual address space.[32]

For example, the function `malloc` works with virtual memory. At initialization, it asks the OS for some pages of (virtual) memory (e.g., via the `sbrk` or `morecore` system calls) that span enough space to fill the initial request. It then keeps a pool of various unused memory chunk sizes so that future requests can be placed into as small a space as possible (in a greedy attempt to minimize fragmentation), as quickly as possible. If no pooled space is large enough for the current request, the OS is prompted to grow the segment allotted to the current process.

The implementation of virtual memory is something of a cooperative effort between hardware (CPU) and software (operating system). In particular, on the hardware side, modern CPUs have a memory management unit whose sole purpose is to deal with virtual memory; i.e., virtual to physical address translation, cache control, memory protection, etc. As a matter of practicality and performance, almost all current virtual memory implementations divide the virtual address space into *pages*. Pages are the atomic unit of memory for the OS. Programs requesting memory space can only receive in page-sized chunks; even if only 10 bytes are needed, the OS allocates an entire page. In the x86-64 architecture, standard pages span a 4KB address space. 2MB pages are also available on all x86-64 processors; AMD’s latest CPU (K10) additionally supports 1GB pages; see Table 4.7.[33] The OS maintains a

[30]Of course for performance, the location of the current working set in the memory hierarchy is important.

[31]Some segments of memory are allocated directly in physical space; these are usually low level processes fundamental to the operation of the OS. For example, page tables, certain data buffers, certain drivers, interrupt handlers, etc. cannot be paged out.

[32]The size of the virtual space is usually the size of the available DRAM combined with any additional page-file space allowed on hard disks or other storage media. This additional space is referred to as the page file.

[33]Some more specialized processors and operating systems, like those found in certain supercomputers, support page sizes in the range of 16GB or larger.

set of *page tables* that provide the mapping from virtual to physical addresses, if the virtual address corresponds to data that is currently in RAM. If not, then the page table contains a flag indicating that this virtual page is *paged out* of RAM; e.g., it is currently on stored on the hard disk.

Virtual addresses can be conceptualized into two parts: 1) page table information; 2) physical address offset. When the CPU decodes an instruction requiring it to manipulate some data in memory, the MMU uses the page table information along with the page tables to identify which memory page owns a particular virtual address along with the base physical address of that page. The offset is then added to the base page address to find the physical address of a virtual address. The structure of the page tables is quite complex, but the details are not important here. Drepper[20] provides a more complete description.

The process is different if the desired virtual page is not currently in physical RAM. In this scenario, a *page-fault* exception occurs; these are costly events that completely interrupt current execution and then prompt the OS to locate the desired virtual page in the page file and load it into main memory. Usually, bringing a faulted page into memory will page out another page of memory to the page file.[34]

The page tables are always held in a known location in physical RAM. The CPU has a special register (set by the OS) that stores the base address of the page tables. Given all the effort made to cache memory values for faster access, it would make little sense if translating a virtual address always required accessing the page tables in main memory. The problem is worsened because page tables are typically several levels deep, requiring multiple memory accesses to complete the translation. Additionally, a nontrivial amount of computation is required to perform the address translation, even if the entire page table were in fast cache. The solution to this issue is the Translation Lookaside Buffer (TLB).

4.3.1 The Translation Lookaside Buffer

The TLB is a special cache that holds a small subset of precomputed virtual to physical address translations relevant to the latest memory operations. As with CPU caches, TLBs are

[34]The memory-bound workloads found in most scientific computing applications must avoid page faults at all costs.

often divided into instruction and data buffers: ITLB and DTLB. The TLB is implemented as content-addressable memory (CAM). Given an address, RAM will quickly return the data located there. Given a word of data, CAM will quickly return the address(es) where the data were found; additional values associated with the search key may also be returned. CAM is even more expensive to implement and use than CPU caches, so TLB space on modern CPUs is very limited. Table 4.7 summarizes TLB sizes for the latest Intel and AMD CPUs. The layout of the TLB differs from processor to processor; modern CPUs often have different TLBs for instructions and data, just as in the memory hierarchy. Generally, TLB misses on commodity CPUs incur latencies that are somewhat worse than those associated with L2 cache misses. When a TLB miss occurs, the CPU accesses all page table levels in main memory and computes the address translation. The newly computed translation is cached in the TLB, evicting some older translation. Due to this expense, L1 caches are usually physically addressed; i.e., they work directly with physical memory addresses. Larger caches tend to be virtually addressed as a practical measure.

Cache	Entries (4KB/2MB)	Set Associativity
Core 2		
L1 Data TLB[35]	16/16	4-way
L2 Data TLB	256/32	4-way
Inst. TLB	128/8	4-way
Core i7		
L1 Data TLB	64/32	4-way
L1 Inst. TLB	64/7	full
L2 Unified TLB	512/0	4-way
AMD K10		
L1 Data TLB	48/48	full
L1 Inst. TLB	32/16	full
L2 Data TLB	512/128	4-way
L2 Inst. TLB	512/0	4-way

Table 4.7: TLB characteristics for the Intel Core 2 and Core i7 and the AMD K10 architectures. Note that the number of entries are listed as A/B, where A is the number of entries using 4KB pages and B is the number of entries with 2MB pages. AMD K10 also supports 1GB pages, which are not listed here.

Table 4.7 lists several different page sizes: 4KB and 2MB.[36] Currently, 4KB pages are the default in commodity operating systems. The 2MB alternative are often referred to as "huge pages." However, here the default choice is not always the best choice. Consider the Core 2 DTLB: it has 256 entries in 4KB mode and 32 in 2MB mode. In standard mode, the DTLB spans 1MB of data; in huge mode, it spans 64MB of data. 1MB is smaller than the L2 cache on Core 2 processors. Thus, even if the working set is 4MB (so it fits entirely in L2 cache), DTLB misses will occur. For example, if a program steps sequentially over all 4MB of data, 0 L2 cache misses occur, but 1024 TLB misses occur. If the data spans several GB (as would the Jacobian matrix on large simulations), the situation only worsens. Using 2MB pages, the number of TLB misses decreases by a factor of 64.

Why are 2MB pages so uncommon? First, making all pages 2MB in size is not practical. Many processes do not require more than a few KB of RAM; allocating 2MB to each of these would be wasteful. For scientific computing applications, this is a non-issue: wasting an extra 2MB is hardly a concern when the working set is measured with GB. Additionally, the 4KB size is a long-standing standard. As a result, both CPU and OS support for 2MB pages is currently sub-optimal; but both are improving. On the CPU side, TLB sizes are increasing, especially in huge mode. On the OS side (at least in Linux), recent conferences have been discussing operation with multiple page sizes simultaneously; the OS would automatically determine which page size is appropriate depending on the size of memory requested[47].

Unfortunately, currently huge pages are somewhat awkward to use in Linux. They must be allocated beforehand through kernel commands. Once allocated, only programs written to use huge pages can access those regions of memory. Resources do exist on huge page interfaces and administration in Linux environments; see [32] for a recent guide. Even with these tools, finding several hundred MB or a few GB of contiguous physical memory is difficult once memory has become fragmented, requiring that large sets of huge pages be allocated immediately after boot. Furthermore, memory allocated with huge pages is unavailable for use by the rest of the system.[37] On the implementation side, huge pages are not allocated

[36]These are relevant to x86-64 environments. In 32 bit mode, the page size options are 4KB and 4MB.

[37]This should not be an issue for users running large simulations, since nominally the simulation is the only major program running.

through familiar calls to `malloc`. Instead they are accessed with either `mmap` or `shmget`, depending on the huge page interface chosen. In short, using huge pages may not be as easy as using 4KB pages. But the performance benefits are often worthwhile in environments accessing large amounts of data, particularly if those accesses are not in stride.

Some libraries exist to simplify the use of huge pages. The library `libhugetlbfs` [48] in particular can make huge pages much more user-friendly. This library includes functions that allow programmers to allocate data on pages of varying size (depending on hardware and OS support) without having to interact with functions like `mmap` directly. The library can also make `malloc` use huge pages by overriding the `morecore` (also known as `sbrk`) function via the `LD_PRELOAD` environment variable. That is, all dynamically allocated memory will be allocated through huge pages; statically allocated memory can also be allocated through huge pages by linking against other `libhugetlbfs` components. Finally, `libhugetlbfs` includes system tools that simplify the setup and administration of huge pages.

Milfeld[52] provides data on the improvement of 2MB pages (over 4KB) in a simple test environment (strided array access). No misses with 2MB pages are seen over a wider range (due to the larger span), and once TLB misses occur, the maximum latency is 15% to 30% less (depending on the architecture). Gorman[32] provides some benchmark data on the IBM PowerPC architecture, showing the superiority of 2MB pages over a set of benchmarks (mostly some variation on strided array access) for various working sets. Several authors have also examined TLB performance on more real-world applications. McCurdy[51] studies the effect of page size on various scientific computing oriented benchmarks as well as some real-world applications of interest to them. Their results are older, run on the AMD K8 architecture,[38] but still show 2MB pages outperforming 4KB pages by as much as 60% in overall runtime (in the worst case, 2MB and 4KB pages performed equally). Oppe[58] examined 4KB versus 2MB pages on supercomputing applications (run on a Cray XT5, which uses the AMD K10). Results varied by compiler, but across 256 to 1280 processor runs, 2MB pages were 10% faster on average.

There is a fair amount of data supporting the use of 2MB pages for many scientific computing applications. Figure 4-4 estimates the effect on the linear solver aspect of DG

[38]The AMD K8 had only an 8 entry DTLB for 2MB pages; this is very small.

codes. The figure uses the same test code as Figure 3-1 in Section 3.5, except that now the 1GB block of memory is allocated using 2MB pages. The huge pages produced more than a 15% flops improvement in the best case. Figure 4-5 shows the relative performance improvement from using huge pages on smaller block sizes.

2MB pages are the most effective when matrix blocks occupy much less than 4KB of space, as shown in Figure 4-5. 4KB corresponds to a block size of about 22. A TLB miss occurs at least once for every unique 4KB of data read from memory. TLB misses are minimized if the data is read sequentially. Thus, when accesses are random (as in Figure 4-4), it often occurs that one access is separated from the next access by more than 4KB. For small blocks, less than 4KB of data is read, thereby increasing the relative impact of TLB misses.

Finally, note that the page size choice also affects the effectiveness of hardware prefetching. As mentioned in the discussion about prefetching, hardware prefetchers cannot cross page boundaries.[39] It is required that a cold miss occur in the new page region before hardware prefetching may continue, even if the CPU would have otherwise known to prefetch data. As a result, using huge pages has an added benefit in that hardware prefetching will become more effective. In future processor designs, the limitations on prefetching across page boundaries may be lifted; this is currently a topic of research[3]. Lastly, if huge pages become more prevalent, a corresponding improvement in hardware prefetching will occur. Currently, hardware prefetching is somewhat limited in complexity because it designed to perform best in small, less than 4KB environments[20].

[39]The reason has to do with the design of virtual memory systems, but it is unimportant here. See [20] for details.

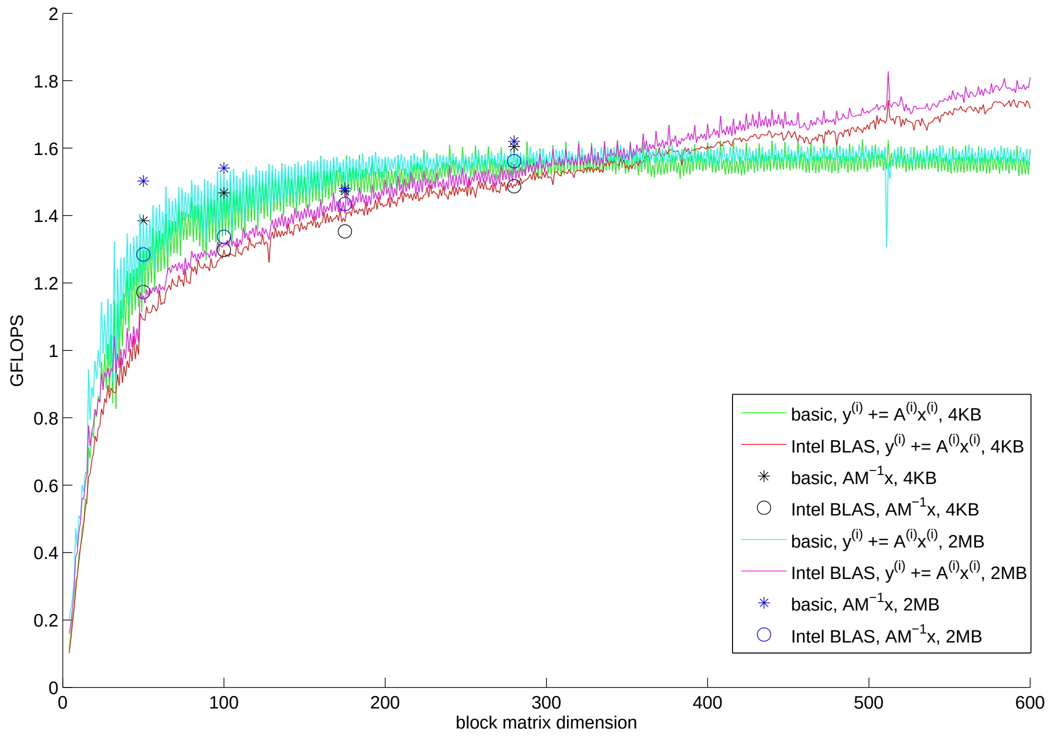

Figure 4-4: A demonstration of the potential gains possible when using huge pages (2MB) on a sequence of matrix-vector multiplies, where the data spans 1GB. The cases shown are running the same setup as in Figure 3-1. The 4KB data are the same as the data plotted in Figure 3-1; the 2MB data are new, demonstrating the benefits of huge pages. For very small blocks, huge pages show more than a 70% improvement (see Figure 4-5). For larger blocks, 5% is more reasonable for the Intel BLAS and 2% for the basic implementation. Note that on newer Intel and AMD processors (designed with larger data sets in mind, e.g., in server environments) supporting larger TLBs in huge page mode, the performance difference may be even more substantial.

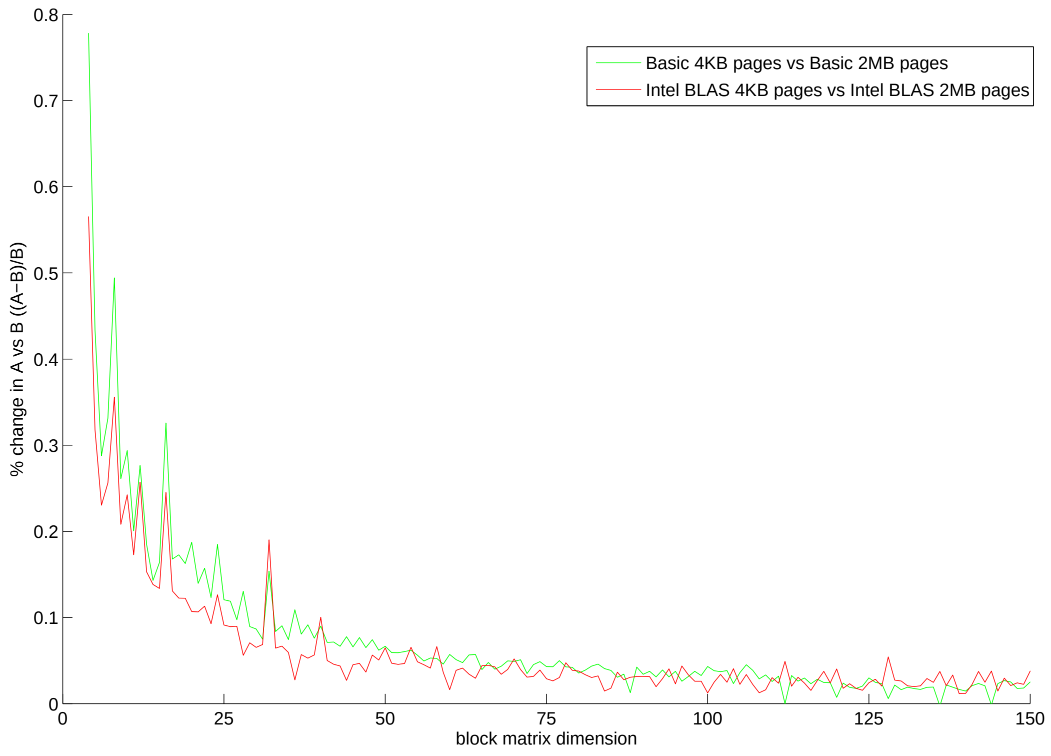

Figure 4-5: The relative performance difference between 4KB and 2MB pages. The data is taken from the “simulated” Ax results shown in Figure 4-4. Only block sizes up to 150 are shown to emphasize the improvements for smaller blocks. For larger blocks (not shown), the Intel BLAS performs around 3% to 6% better with huge pages. The basic implementation performs 0% to 2% better for the larger block sizes, except at size 512 where the large performance loss can be seen in Figure 4-4. Thus the Intel BLAS sees a greater performance improvement due to 2MB pages.

Chapter 5

Optimizing a DG Implementation

This chapter will focus on the application of the computer architecture concepts and optimization techniques discussed previously to the two main parts of a DG flow solver: the residual and Jacobian evaluation and the linear solver.

5.1 Residual and Jacobian Evaluation

- In the initial description of the DG discretization given in Chapter 3, the residual (and Jacobian) evaluation process described evaluating the contribution at each quadrature point (e.g., $R_{g,k,n}$) and summing into the residual vector. In the end, the residual vector, $R_{k,n}$ contained the sum, $\sum_{g=1}^{N_{quad}} R_{g,k,n}$. The inviscid Galerkin term (Equation 3.1) is reproduced here:

  $$R_{g,k,n} = w_g \partial_{x_i} \phi_{g,n} F_{i,k}(\mathbf{U}_n \phi_{g,n}).$$

  More simply, $\partial_{x_i} \phi_n F_{i,k}$ can be interpreted as the sum of N_{dim} outerproducts each of the form $p_n H_k$; these outerproducts occur at each quadrature point. The viscous contribution also has the same form. Each of the resulting matrices is then summed into $R_{k,n}$. In effect, N_{dim} separate matrix-matrix products have been computed. That is, $R_{k,n} = \Phi_{i,g,n} G_{i,g,k}$, where $\Phi_{i,g,n}$ indicates a matrix of size $N_{Dim} N_{quad} \times N_{bf}$ storing every basis function derivative at every quadrature point; and $G_{i,g,k}$ is a matrix storing the flux (in each spatial direction) for every state at every quadrature point (all scaled by w_g). Emphasizing the point further, $\Phi_{i,g,n}$ contains $\partial_{x_0}\phi_{g,n}$ ordered first, then the x_1 derivatives, etc; similarly the matrix of fluxes, $G_{i,g,k}$ orders all x_0-components first, then x_1, then x_2. For residuals (e.g., jump terms) that do not involve summation over spatial directions, then only $\Phi_{g,n}$ and $G_{g,k}$ are needed.

The outerproduct approach reads $N_{quad}(N_{bf} + N_{sr})$ `double` from cache and writes $N_{quad}N_{sr}N_{bf}$ `double` to cache. As appropriate, these operations are once per spatial dimension. The basic matrix-matrix product involves the same number of reads but only $N_{sr}N_{bf}$ writes. As described in Section 4.1, a blocked algorithm (e.g., as found in the optimized BLAS) will perform even better.[1] By reorganizing the outerproducts into a matrix-matrix product, the performance can be improved by more than a factor of 2.[2]

However, the matrix-matrix version has a disadvantage over the outerproduct version. In both cases, Φ should be precomputed. In the outerproduct case, $F_{i,k}$ can be overwritten at each subsequent quadrature point. However, the matrix-matrix version will require the storage of $G_{i,g,k}$, which spans $N_{quad}N_{dim}N_{sr}$ `double`. The situation is worse with Jacobian assembly, when the extra storage is $N_{quad}N_{dim}N_{sr}^2N_{bf}$. Nonetheless, the extra storage requirement is easily offset by the efficiency improvement when these pieces all fit in cache. However, danger arises when the combination of these matrices does not fit in cache, as discussed below.

Jacobian assembly can be reorganized in a similar fashion. For example, one of the face residual terms can be written:

$$\partial_{U^L_{a,m}} R_{k,n} += \phi_m B_{k,i,a} \partial_{x_i} \phi_n.$$

Here, it is possible to compute $C_q = \phi_m B_{k,i,a}\partial_{x_i}$ or $D_q = B_{k,i,a}\partial_{x_i}\phi_n$ first; q ranges over $N_{sr}^2N_{bf}$. As pointed out below, the latter choice (using D) is preferable. To motivate reorganizing the Jacobian assembly into a matrix-matrix product, note that

[1] If the matrix dimensions are so small that blocking is ineffective, then the optimized BLAS may be outperformed by the basic routine. This may also occur if the input matrices are too "tall and skinny."

[2] As observed on a 3GHZ, Intel Core2 CPU.

the tensor on the right hand side of the definition of D has been "collapsed" into a vector, D_q. Similarly, let $S_{m,q} = \partial_{U^L_{a,m}} R_{k,n}$ also refer to the Jacobian matrix. Then $S_{m,r} = \Phi_{g,m} D_{g,q}$. Organizing the data layout in $D_{g,q}$ and/or $\partial_{U^L_{a,m}} R_{k,n}$ requires care, since the collapsed indices a, k, n must align properly with the data ordering implied by the single index q.

Depending how the data is ordered in the Jacobian matrix and the temporary D matrix, it may also be necessary to handle the matrix-matrix products corresponding to summation over N_{quad} one state at a time. By comparison, the previous discussion involved handling all states simultaneously. This corresponds to whether the original assembly process is expressed in terms of OPH or OPV (see below) calls. Using more than N_{sr} separate matrix-matrix products is not recommended. Generally, a single matrix-matrix product is preferable. Unfortunately, this is not always possible. But as described in the next paragraph, generating an intermediate matrix with the wrong data ordering and reordering via a single transpose-like operation is a reasonable alternative.

Additionally, observe that with this optimization, the previously recommended Jacobian matrix layout $\partial_{U^L_{a,m}} R_{k,n}$ is not possible. In particular, the index m must be first or last, since the matrix-matrix product only allows $S_{m,q}$ or $S_{q,m}$, regardless of how a, k, n are organized within q. Thus, an appropriate intermediate ordering could sequence the indices k, n, a, m. Transposing that intermediate result as a square matrix of dimension $N_{sr} N_{bf}$ recovers the suggested $\partial_{U^L_{a,m}} R_{k,n}$ order. With other intermediate orderings such as m, n, k, a, a more complex "transpose" is required. Nonetheless, experiments in ProjectX have shown the matrix-matrix optimization remains worthwhile.

The worsened locality mentioned above requires further discussion. Generally, the extra storage requirement is worthwhile to increase the use of BLAS-3 operations. However, if the size of the temporary matrices exceed the largest level of cache,[3] performance will suffer greatly. To compensate, the quadrature points should be partitioned into

[3]The last cache on some CPUs is very large and relatively slow. In such cases, it may be desirable for the problem to fit entirely in a smaller cache.

(contiguous) chunks sized so that all data for each fits entirely in cache. This requires little programming effort, but the residual and Jacobian evaluation will have to be explicitly cache-aware, and thus processor-aware, since cache performance varies from CPU to CPU.

Lastly, note that the performance of optimized BLAS libraries will depend on the data layout and argument ordering. For example, depending on whether $R_{k,n}$, $\Phi_{g,n}$, and $G_{g,k}$ are ordered row-major or column-major, $R = \Phi G$ or $R = G\Phi = (\Phi G)^T$ may be appropriate. The optimized BLAS may perform ΦG so much faster than $G\Phi$ that it is worth manually transposing R if the resulting ordering on R is not correct.

- Precomputing: precomputation is a way of trading time for space. Precomputation also avoids redundant operation, since there is no need to precompute quantities that are only needed once. Precomputed quantities only need to be computed once and usually have very little effect on the overall run-time. However, extra space is required to store the results. Generally, it is almost always worthwhile to precompute quantities that are shared across elements such as basis functions and mass matrices (and their inverses). Quantities that vary from element to element, such as geometry-related quantities, may not be worth precomputing in steady-state solvers since the savings are relatively small. But these too should be precomputed in an unsteady solver or if their cost is deemed prohibitive. When evaluating element or face residuals and Jacobians, elements and faces that share precomputed quantities should be handled together temporally. This improves the temporal locality of accesses to the precomputed quantities. The efficiency gains from precomputation will be lost if those quantities must be repeatedly loaded from main memory, incurring unnecessary conflict misses.

  The precomputation steps may be implemented lazily, but it is more efficient to identify exactly which quantities are needed, compute them, and never have to check whether some quantity has already been computed before using it. Lookup tables are ideal for tracking precomputed quantities; the tables should be indexed on a minimal set of distinguishing features. For example, basis functions could be indexed over element shape,[4] quadrature rule, and interpolation order. Indexes should range over small sets of integers starting at 0. If indexes must range over large sets of integers, multilevel hashing would be more appropriate.

- Evaluate tensor contractions as early as possible. Delaying contractions results in carrying around more data and performing unnecessary calculations. For example, there are 3 ways to compute

$$\partial_{U^L_{a,m}} R_{k,n} += \phi_m B_{k,i,a} \partial_{x_i} \phi_n, \tag{5.1}$$

  but $\phi_m(B_{k,i,a}\partial_{x_i}\phi_n)$ is the most efficient, involving $2N_{dim}N_{sr}^2N_{bfL}+2N_{sr}^2N_{bf}^2$ flops. Evaluating $(\phi_m B_{k,i,a})\partial_{x_i}\phi_n$ costs $2N_{dim}(N_{sr}^2N_{bfL}+N_{sr}^2N_{bfL}^2)$ flops. The first order similarly has a substantially reduced number of memory operations.

- Outside of flux and $\mathcal{A}$ evaluation, the residual and Jacobian evaluation process is primarily composed of tensor contractions (inner, matrix-vector, and matrix-matrix products) and tensor products (outer products). The contraction operations should be familiar, but the tensor products fall into two categories that may not be immediately obvious. In ProjectX, they are called the "horizontal" and "vertical" (block) outer products (OPH and OPV). Note that the following discussion considers row-major matrix storage.

  OPH takes as input $v_k \in \mathcal{R}^n$ and $B_{i,j} \in \mathcal{R}^{b_1\times b_2}$ and produces $H_{i,k,j} = B_{i,j}v_k$, where $H_{i,k,j} \in \mathcal{R}^{b_1\times b_2 n}$. Alternatively, H may be written, $H = [Bv_0, Bv_1, \ldots, Bv_n]$. A simple implementation follows:

```
double * pB = B, * pH = H;
double t;
for(i=0; i<b1; i++){
  for(j=0; j<n; j++){
    t = v[j];
```

[4]If multiple shapes (e.g., simplex, quadrilateral) are used.

```
    for(k=0; k<b2; k++){
      pH[k] += pB[k]*t;
    }
    pH += b2;
  }
  pB += b2;
}
```

The inputs of OPV are the same as those of OPH, but OPV produces $G_{k,i,j} = B_{i,j}v_k$, where $G_{k,i,j} \in \mathcal{R}^{b_1 n \times b_2}$. OPV can be rephrased as the outer product $G_{i,j} = b_i v_j$, where $G_{i,j} \in \mathcal{R}^{b_1 n \times b_2}$, $b_i \in \mathcal{R}^{b_1 b_2}$, and $v_j \in \mathcal{R}^n$.

```
double * pV = V;
double t;
for(j=0; j<n; j++){
  t = v[j];
  for(k0; k<b1*b2; k++){
    pV[k] += B[k]*t;
  }
  pV += b1*b2;
}
```

Many tensor products (i.e., OPV and OPH) and tensor contractions range over small quantities like N_{sr}. Code implementing these operations should involve fully unrolled loops for common, small parameter values. Unroll factors of 2 to 4 are more appropriate for loops over quantities like N_{bf} that could range over a much wider range of values.

- The flux cost quantities denoted $C(F)$, $C(\hat{F})$, $C(\mathcal{A})$, etc. (and their state derivatives) can easily dominate the overall residual and Jacobian evaluation times if these functions are not implemented carefully. Specifically, unless N_{dim} and N_{sr} are known at compile-time, writing these functions with loops over N_{dim} and N_{sr} leads to substantially worse performance. Flux evaluation may be slowed by as much as a factor of 5-10. If the loop variables' ranges are known at compile-time, the compiler may automatically unroll these loops; the programmer should check to ensure this is happening correctly.

- Merge tensors with the same dimensions as early as possible. For example, $A_{i,j,k,l}u_{k,l} + B_{i,j,k,l}u_{k,l}$ should be generally computed as $(A_{i,j,k,l} + B_{i,j,k,l})u_{k,l}$. Here, one set of N_{dim}^2 matrix products is avoided.

- Taking advantage of sparsity in $\mathcal{A}$ and/or $\partial_u\mathcal{A}$. $\mathcal{A}$ has many zeros, as shown in Appendix A. As a result, its state derivative tensor will have at least N_{sr} times as many zeros. The full rank-5 tensor $\partial_{u_a}\mathcal{A}_{i,j,k,l}$ is never needed in its entirety. Rather, Jacobian evaluation only requires the result of contractions like $\partial_{u_a}\mathcal{A}_{i,j,k,l}\partial_{x_j}u_l$. Thus, only the smaller rank-3 product should be stored; furthermore, computing this result should take advantage of the sparsity in $\partial_u\mathcal{A}$.

- Matrix inversion, matrix determinants, and eigenvalues over small problems (specifically, $N_{dim} \times N_{dim}$ matrices) should be coded explicitly. Relatively simple, analytic formulas exist for all of these operations when the problem size is sufficiently small. Direct implementations like these can be more than 10 times faster than calling BLAS or LAPACK routines. However, be aware that solving small eigenvalue problems with analytic root-finding formulas may be ill-conditioned; LAPACK (or other safe routines) should be used if a loss of accuracy is detected.

5.2 GMRES

- The GMRES algorithm uses Arnoldi Iteration to form an orthogonal basis for the Krylov space. As suggested by [61], Arnoldi is implemented with the Modified Gram-Schmidt (MGS) process. MGS only uses BLAS-1 operations and is inherently sequential. The cost of the Arnoldi process varies widely with the number of inner GMRES iterations used, but in ProjectX (which specifies a maximum of 200), Arnoldi composes (at worst) about $\frac{1}{3}$ to $\frac{1}{2}$ of the total GMRES cost. If the required Krylov space could be formed and then orthogonalized, the process could utilize LAPACK's QR factorization routines (which are written with BLAS-2 and BLAS-3 calls) instead of the all BLAS-1 MGS version. Problematically, the monomial basis $b, Ab, A^2b, \ldots$ is extremely ill-conditioned. By using a (Newton) polynomial basis, [6] were able to form the full Krylov space and orthogonalize it afterward[6] using the QR functionality of LAPACK. LAPACK's QR routine predominantly uses BLAS 2 operations, making it more efficient than the BLAS 1 MGS orthogonalization.

- If a reordering is used, linear algebra operations ($a = x^T y$ and $x = \alpha x$ in particular) often do not need to be aware of the reordering. For multiplication by a scalar, the ordering never matters. For an inner product, as long as x and y are stored with the same ordering, the product should be computed as if no reordering were present–with one loop sweeping over the elements. If x and y are stored with different orderings, then clearly the inner product must match elements that refer to the same physical quantity. Such operations should be avoided whenever possible. They have very little memory locality and are much more expensive (by as much as 2-3x) as a result.

- Using huge pages (see Section 4.3.1) will also improve GMRES performance. This modification is relatively simple since the underlying GMRES implementation does not need to change. At worst, the memory allocation process must be altered to utilize huge pages, since the `malloc` implementation in standard `C` libraries can only access 4KB pages. At best, a library like `libhugetlbfs` can provide an alternative `malloc` implementation that uses huge pages. As shown in Section 4.3.1, the benefit of huge pages is most significant for small block sizes, but huge pages remain beneficial even for very large blocks.

- Prefetching can be effective when Jacobian blocks are very small; e.g., $N_{sr} \leq 8$ and $N_{bf} \leq 4$ (P_0 or P_1). For small blocks, the overhead of "random" access is very noticeable. Accessing the Jacobian sequentially is difficult or impossible unless very simple preconditioners are used. Thus with more powerful tools like ILU, the speed-up in the Au kernel can be around a factor of 2 if upcoming blocks are prefetched appropriately.

For larger block sizes, prefetching can be detrimental since undue load is placed on the cache and memory system, and the penalty of non-sequential access is mostly masked by the large individual problem sizes.

Chapter 6

Validation

The validation problems were chosen to test the 3D, viscous discretization of ProjectX; the Navier-Stokes Equations were of particular interest. The test scheme involved performing grid-convergence studies to check that optimal convergence rates were obtained. The problems scaled up in complexity, first isolating the residual evaluation process using the diffusion and convection-diffusion equations. Then tests were conducted on Navier-Stokes problems. Cases also included flows with boundary layer-like features, since high Reynolds Number flows are of interest. In all cases, source terms were used so that an analytic solution would be available. As such, boundary conditions were set using the known exact solution for diffusive operators; for convective operators, the exact solution was enforced weakly via an upwinding scheme. The metric for correctness was the L_2 norm and H_1 norm (or seminorm) of the solution error. The DG method obtains an L_2-error convergence rate of $O(h^{p+1/2})$ for purely convective problems, but for problems of interest, $O(h^{p+1})$ is observed. Using the BR2 viscous discretization, the DG method recovers the optimal $O(h^{p+1})$ rate seen in continuous Galerkin Finite Element methods. The H_1-error convergence rate is expected to be 1 order lower than L_2: $O(h^p)$.

The test cases were solved on a sequence of uniformly refined meshes.[1] If applicable (i.e., Case 2 and Case 4), the initial meshes were graded to match the underlying flow features.

[1]Cases 2 and 4 use unstructured meshes generated by specifying an underlying metric. Here, uniform refinement refers to uniformly "refining" the metric field, which is not the same as uniformly refining the starting mesh by subdividing elements.

Additionally, the nonlinear residual norm was driven to at most 1.0×10^{-14} whenever possible; exceptions are noted in the case discussions.

6.1 Case 1: Diffusion

Case 1 involved solving the steady diffusion equation in 3D:

- $-\nabla \cdot (\mu\nabla u) = G(u)$ with $\mu = 0.1$.
- $G(u)$ was chosen so that $u(x, y, z) = QuarticPoly(x, y, z)$ was the exact solution. $QuarticPoly$ was a quartic polynomial containing all terms $x^a y^b z^c$ such that $a+b+c = 4$ with $a, b, c \geq 0$.
- Grids: 162, 1296, 10368, 82944, 663552 uniform tetrahedra covering the cube $[-2, 2] \times [-2, 2] \times [-2, 2]$.

Convergence results are shown in Figure 6-1 (L_2 error) and Figure 6-2 (H_1 (semi) error). Rates are given in Table 6.1. Note that $p = 4$ and $p = 5$ results are not shown because the solution is an element of the approximation space in these cases. Thus, setting $p > 3$ obtains the exact solution, which was confirmed experimentally. Optimal rates were obtained for this problem, except at $p = 0$. Setting a constant $\eta = 6.0$ leads to an inconsistent discretization at $p = 0$, explaining the observed rates of 0.00.

Error Type	$p = 0$	$p = 1$	$p = 2$	$p = 3$
L_2	0.00	1.96	2.99	4.00
$H_1 semi$	0.00	0.97	1.99	3.00

Table 6.1: Rate of convergence for Case 1. The table displays the exponent k where the order of convergence is $O(h^k)$.

6.2 Case 2: (Steady) Convection-Diffusion

Case 2 involved solving the steady convection-diffusion equation in 3D. The exact solution is meant to simulate a boundary layer, testing the solver with highly graded, unstructured

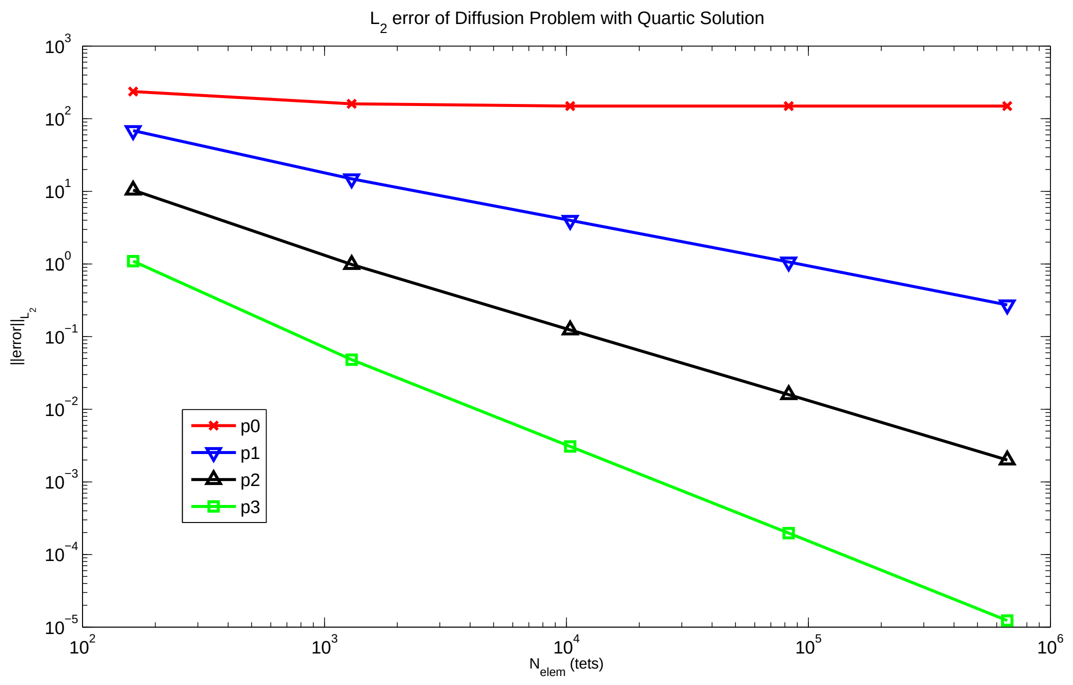

Figure 6-1: L_2-norm error convergence for Case 1.

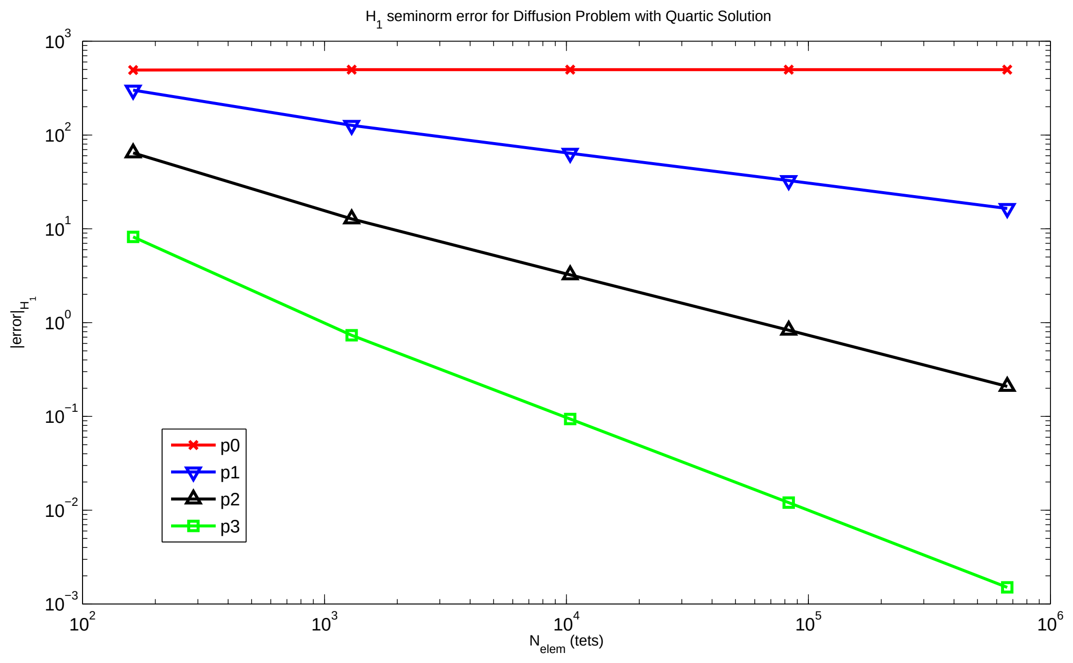

Figure 6-2: H_1-seminorm error convergence for Case 1.

meshes.

- $-\nabla \cdot (\mu \nabla u) + a \cdot \nabla u = G(u)$ with $\mu = 1 \times 10^{-6}$ and $a =< 1, 0, 0 >$.

- $G(u)$ was chosen so that $u(x, y) = e^{(-y/\sqrt{x\mu})}$ was the exact solution.

- Grids: 794, 6416, 51261, and 409591 tetrahedra stretched exponentially in the y direction so that the anisotropy is appropriate for the boundary layer. These meshes are unstructured. The domain was $[0.1, 1.1] \times [0, 1] \times [0, 1]$; the x-component was shifted to avoid the singularity at $x = 0$.

Convergence results are shown in Figure 6-3 (L_2 error) and Figure 6-4 (H_1 (semi) error). Rates are given in Table 6.2. Optimal rates were obtained for this problem.

Error Type	$p = 0$	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 5$
L_2	0.50	2.10	3.07	4.09	5.08	6.09
$H_1 semi$	0.00	1.04	2.05	3.04	4.03	5.02

Table 6.2: Rate of convergence for Case 2. The table displays the exponent k where the order of convergence is $O(h^k)$.

6.3 Case 3: Poiseuille Flow

Case 3 involved solving the 3D Compressible Navier-Stokes Equations with Poiseuille flow serving as the exact solution.

- Compressible Navier-Stokes Equations with $\mu = 1.0 \times 10^{-3}$ and $\gamma = 1.4$.

- $G(u)$ was chosen so that the exact x-velocity distribution was quadratic (i.e., arising from a channel flow with a linear pressure gradient) with $\rho = const$. The exact solution is effectively 1D, with no variation in the x and z directions.

- Grids: 162, 1296, 10368, 82944 uniform tetrahedra covering the cube $[0, 1] \times [0, 1] \times [0, 1]$.

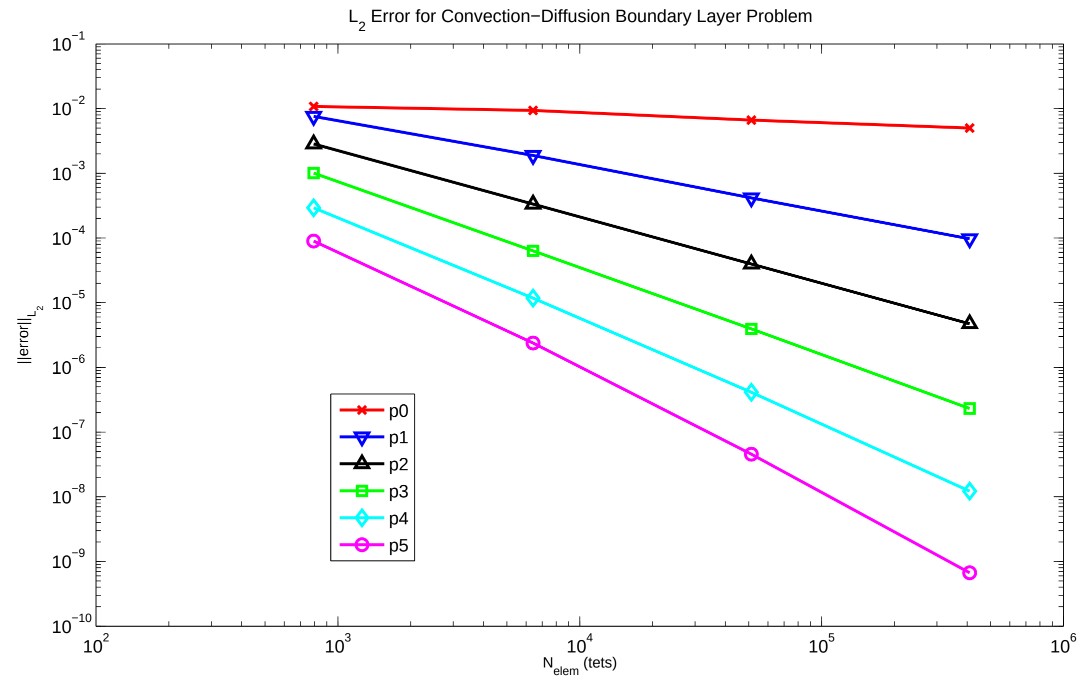

Figure 6-3: L_2-norm error convergence for Case 2.

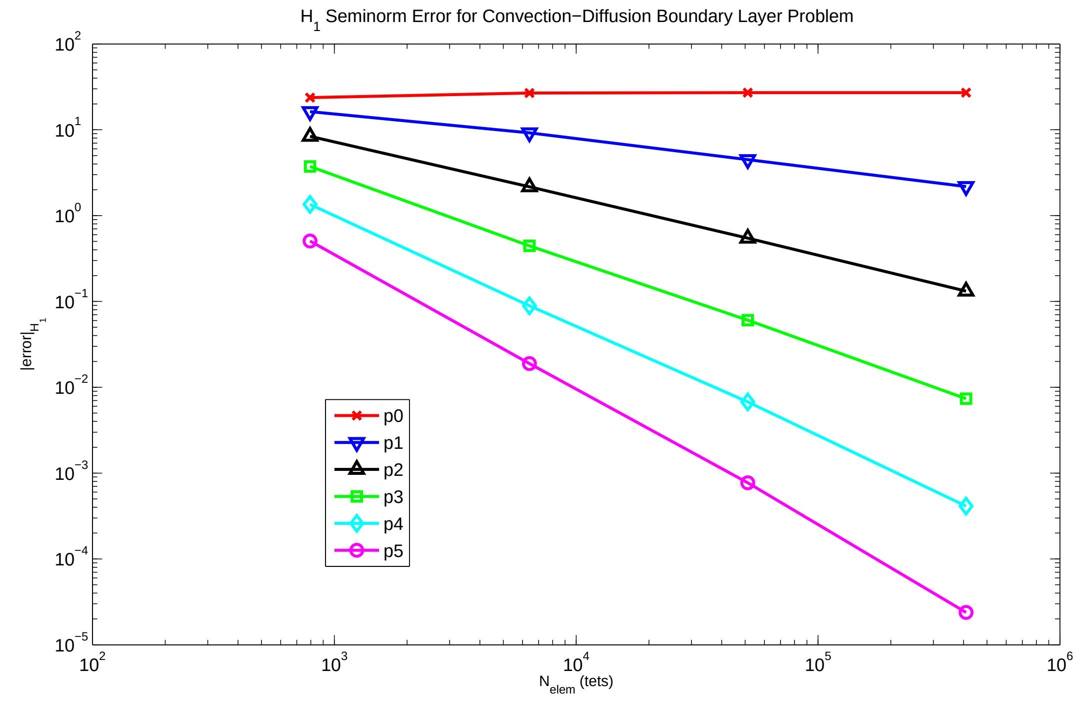

Figure 6-4: H_1-seminorm error convergence for Case 2.

Convergence results are shown in Figure 6-5 (L_2 error). Rates are given in Table 6.3. Note that $p = 4$ and $p = 5$ results are not shown because the solution is an element of the approximation space in these cases. Thus, setting $p > 3$ obtains the exact solution, which was confirmed experimentally. Optimal rates were obtained for this problem.

Error Type	$p = 0$	$p = 1$	$p = 2$	$p = 3$
L_2	0.95	2.05	3.03	3.98

Table 6.3: Rate of convergence for Case 3. The table displays the exponent k where the order of convergence is $O(h^k)$.

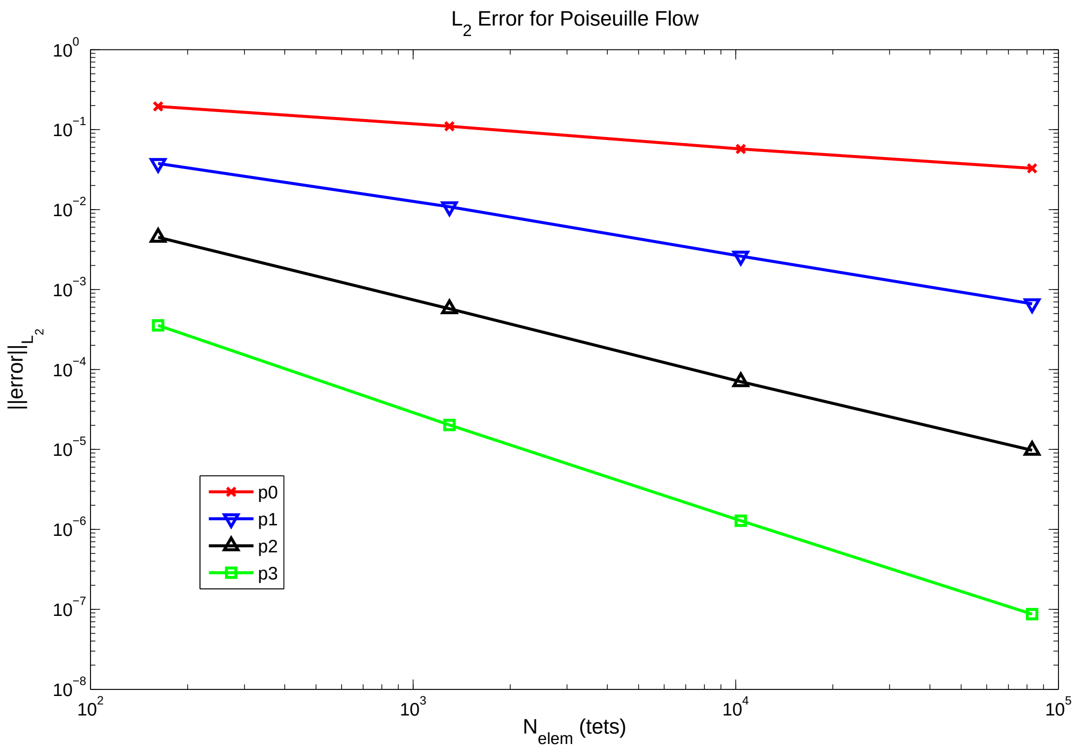

Figure 6-5: L_2-norm error convergence for Case 3.

6.4 Case 4: (Steady) Navier-Stokes

Case 4 is another 3D Compressible Navier-Stokes problem. This time, the exact solution has the same velocity profile as Case 2. Again, the exact solution simulates a boundary layer while avoiding the leading edge singularity seen in the more common problem of flow over a flat plate.

- Compressible Navier-Stokes Equations with $\mu = 1 \times 10^{-6}$ and $\gamma = 1.4$.
- $G(u)$ chosen so that $u(x, y) = e^{(-y/\sqrt{x\mu})}$ is the exact velocity profile and the flow has constant density and pressure: $\rho = 1.0$ and $p = \frac{8}{7}$.
- Grids: 1587, 12792, 40030, 102733 tetrahedra stretched exponentially in the y direction so that the anisotropy is appropriate for the boundary layer. These meshes are unstructured. The domain was $[0.1, 1.1] \times [0, 1] \times [0, 1]$; the x-component was shifted to avoid the singularity at $x = 0$.[2]

Convergence results are shown in Figure 6-6 (L_2 error) and Figure 6-7 (H_1 error). Rates are given in Table 6.4. Optimal rates were obtained for $p \leq 3$ only.

Some results (e.g., for $p \geq 4$) are influenced by linear solver limitations and ill-conditioning. The residual norms on the finer meshes dropped no further than 10^{-12}, whereas the coarse mesh and low p residual norms are all at most 10^{-14}. Further progress does not occur because GMRES stalled and it was unable to produce linear residuals lower than around 10^{-12}. Using a greater number of Arnoldi vectors may alleviate this problem, but memory limitations prevented verification. In Case 2, it was found that using data from solutions with nonlinear residual norms of around 10^{-12} lead to absolute errors of about 0.2 in the observed convergence rates. This result leads to the conclusion that the code is performing properly, but limitations in the parallel linear solve are preventing the obtainment of optimal rates. Additionally, the $p = 5$ results are less reliable since there are only three data points.

[2]The 40030 element mesh is not a uniform refinement, but it is shown so that the $p = 5$ line has a relatively fine result plotted.

Error Type	$p = 0$	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 5$
L_2	0.30	2.17	3.17	3.90	4.60	6.29
$H_1 semi$	0.00	1.07	1.97	2.79	3.54	5.23

Table 6.4: Rate of convergence for Case 4. The table displays the exponent k where the order of convergence is $O(h^k)$.

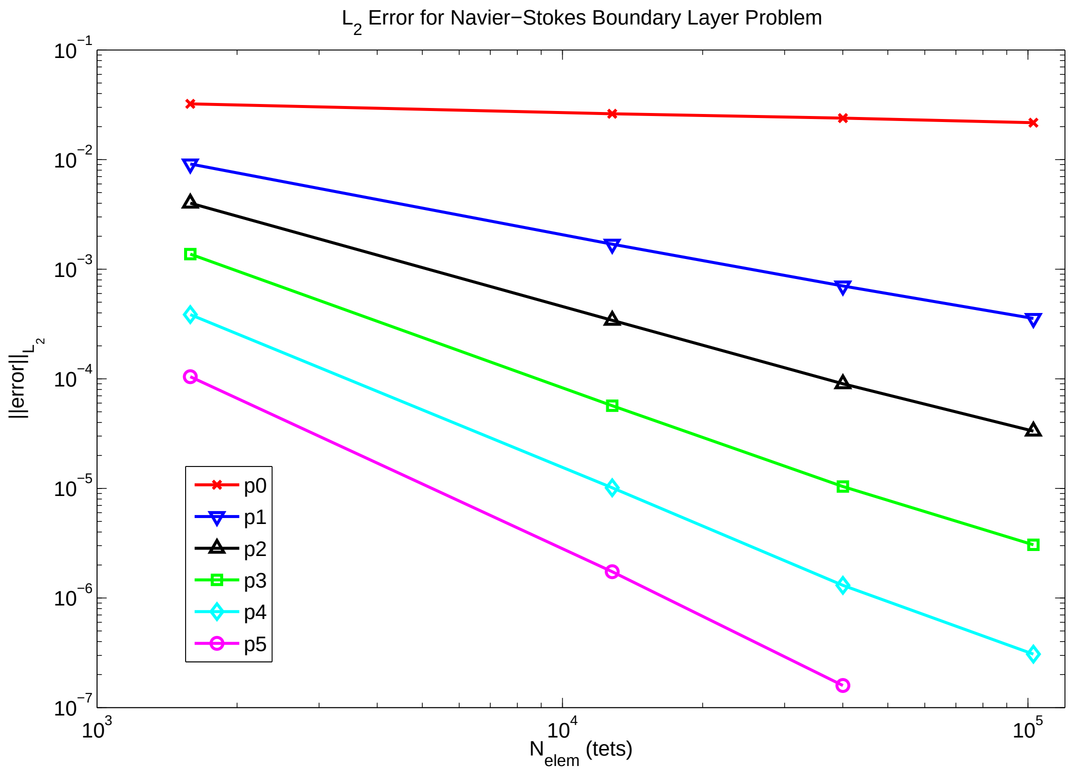

Figure 6-6: L_2-norm error convergence for Case 4.

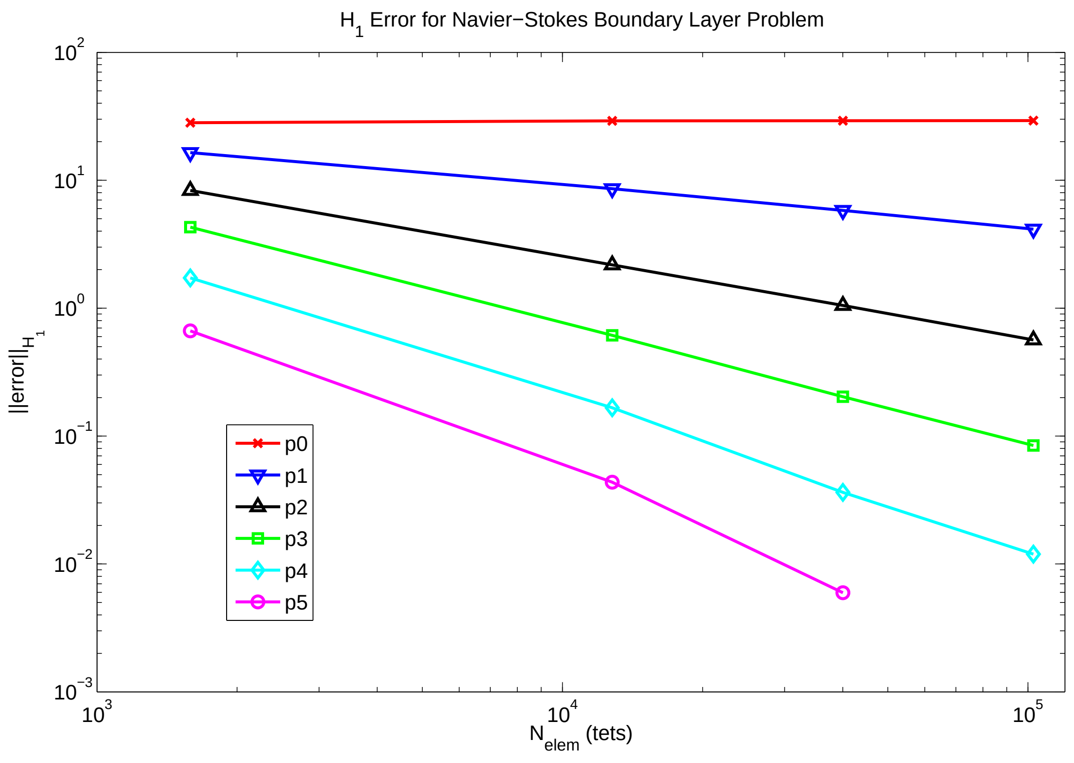

Figure 6-7: H_1-norm error convergence for Case 4.

Chapter 7

Conclusion

This thesis has discussed many of the major features of a fluid dynamical solver based on the Discontinuous Galerkin Finite Element Method. The residual and Jacobian assembly processes were detailed fully, since this topic was often skimmed previously. Major features of iterative linear solvers (GMRES in particular) were also covered, with references provided for the remaining details.

The discussion moved toward features of modern computers relevant to achieving high performance in a DG code. Due to the large working sets often associated with fluid solvers, memory accesses can often be a bottleneck. Even in working sets that fit in CPU cache, communication between cache levels is often several times slower than floating point computation. Thus, a thorough description of the memory hierarchy followed, including discussion of common techniques for utilizing it as efficiently as possible. Also included was an overview of the inner-workings of modern CPUs, which is useful for reasoning about what code optimizations will be helpful in various scenarios.

In the process, simple code was developed that accurately predicts the performance of the Ax kernel in GMRES for block-sparse matrices. The simple "simulated" Ax code identified important features of the actual Ax kernel while removing the unnecessary complexity of a full fluid solver framework. Specifically, working sets much larger than CPU caches were considered with varying block-size. A surprising result arose–optimized BLAS libraries did not outperform basic matrix-vector implementations for a wide spread of problem sizes.

Optimized libraries are often the best way to solve problems quickly and simultaneously avoid re-inventing the wheel. The recommended residual and Jacobian assembly process somewhat non-intuitively reorders summation over quadrature points into a matrix-matrix product in order to take advantage of the optimized BLAS. However, before diving in, programmers should certify that optimized libraries of interest are the best option, perhaps through numerical experiment or otherwise.

Finally, this thesis validated of a 3D fluid solver implementing the DG method (ProjectX) through numerical experiment. In particular, the BR2 lifting operator, viscous residual and Jacobian implementation, and viscous $\mathcal{A}$-matrix evaluation were tested. Test cases were built up, starting from simple diffusion problems with polynomial solutions and ranging to the full Navier-Stokes Equations in 3D. Solver performance was validated by checking convergence rates (across a series of uniform mesh refinements) against theoretical bounds.

7.0.1 Future Work

Residual (and Jacobian) assembly in finite element methods is an embarrassingly parallel problem. This is particularly true of DG methods due to minimal amounts of inter-element communication. With so many instances of executing the essentially same code on different data, the problem is ideal for GPUs. [44] has shown that substantial speedups are possible with GPU implementations of the residual evaluation process. However, their work only covered inviscid, linear problems with explicit-time stepping. Implementing the DG residual and Jacobian evaluation on a GPU would be a natural and nontrivial extension of their work. With current GPU technology, speedups as impressive those achieved by [44] are unlikely, due to greatly increased temporary storage requirements for Jacobian evaluation.

Additionally, current GPU implementations of GMRES have not obtained speed-ups as substantial as those seen in other applications. Preconditioning has been a common limiting factor with some groups [46] using weaker preconditioners like subdomain-wise block Jacobi with inexact local solves via ILU, which has been shown to possess poor parallel scaling [17]. Stronger, minimally-coupled parallel preconditioners could help alleviate this issue while maintaining GPU scalability. GPU implementations of matrix-vector multiplication on block-sparse matrices (such as the DG Jacobian) have also not yet been considered.

One of the issues identified in this thesis is the memory-boundedness of the linear solve in DG problems. Focusing on GMRES, some new techniques are promising but not fully mature yet, as discussed in Section 1.1.

Fundamentally, the performance of GMRES (and other iterative linear solvers) can only be improved by increasing the amount of computational work done for each memory access. Currently, in one $m \times m$ block of a block-sparse matrix vector multiply, m^2 `double` are loaded, $2m^2$ work is done, and the data tossed aside until the next iteration. The efficiency would increase if fewer sparse matrix vector products could be used to solve the same problem. Alternatively, doing more work before tossing aside each Jacobian block would be effective; i.e., making the current BLAS 2 operations closer BLAS 3. So far, the CA-GMRES method [34, 53] addresses the first approach, and the LGMRES and BLGMRES methods [8, 7] addresses the latter. Both were introduced in Section 1.1.

Unfortunately, neither method is a silver bullet, yet. With BLGMRES (and other blocked GMRES methods), results indicated little improvement in some examples from fluid dynamics, including one CG FEM discretization [7]. The technique appears to be hit-or-miss, with its a priori performance remaining difficult to predict, since it is based on a heuristic argument for augmenting the Krylov space.

Additionally, LGMRES and BLGMRES have high start-up costs. The Krylov-augmenting vectors chosen in each method depend on the GMRES solution estimate after a full restart cycle. Thus there is no speed up in the first restart cycle. For guaranteed numerical stability, CA-GMRES has a similarly high start-up cost.

CA-GMRES can form the Krylov basis using a monomial, $b, Ab, A^2b, \ldots$, which resembles the numerically unstable Power Method. Preliminary results indicate that equilibration reduces the penalty for many practical problems [34]. However, the monomial basis is guaranteed to fail for extremely poorly conditioned A. Polynomial bases exist (Newton, Chebyshev) that avoid the conditioning problem altogether, but they require estimates of the eigenvalues of A. These values are readily available through the Ritz values that are almost naturally generated by GMRES. However, as with BLGMRES, generating these values could cost as much as a full restart cycle of computation.[1]

Additionally, CA-GMRES currently does not work with most complex preconditioners due to strong rank restrictions on all off-diagonal submatrices. However, the focus of existing work has not been on preconditioning[34]. Furthermore, these methods have not been applied to the block-sparse Jacobians arising from DG discretizations.

Finally, it may be the case that GMRES is not the best solver choice; other Krylov solvers were not investigated in this thesis but they should be analyzed in the future. The other common Krylov solvers for nonsymmetric matrices could be split into two categories: those requiring an Ax and A^Tx kernel per iteration, and those requiring the Ax kernel twice per iteration. Examples of the former include QMR and BiCG; examples of the latter include TFQMR, BiCG-stab, and CGS. In all cases, these alternatives avoid the long recurrences seen in GMRES, thereby eliminating the Arnoldi iteration entirely. Instead, these methods tend too use the Lanczos iteration.

For methods requiring Ax and A^Tx, the two results may be computable simultaneously by interleaving the data. This would make the additional cost of A^Tx small compared to the initial cost of Ax, but its usefulness depends on how the preconditioner is stored. On the other hand, the multiplies in methods requiring two Ax products cannot be interleaved, since they are related: $y = Ax, z = f(y), w = Az$. Lastly, the work of [34] could be relevant to these other Krylov solvers; he has developed a communication-avoiding BiCG method as well. Other communication avoiding Krylov kernels do not exist now, but may be upcoming.

[1]Or more, as [34] points out that with the Newton basis, if s inner iterations are required, then sometimes $2s$ Ritz values are needed for convergence.

Appendix A

Inviscid and Viscous Fluxes for the Navier-Stokes Equations

The Navier-Stokes equations can be written:

$$\partial_t u_k + \partial_{x_i} F^{inv}_{i,k}(\mathbf{u}) - \partial_{x_i} F^{vis}_{i,k}(\mathbf{u}, \nabla \mathbf{u}) = 0.$$

Following [23], the upcoming sections specify F^{inv} and F^{vis} as well as the decomposition $F^{vis}_{i,k} = \mathcal{A}_{i,j,k,l}(\mathbf{u})\partial_{x_j} u_l$.

A.1 Inviscid Flux

Component Name	Indices	$F^{inv}_{i,k}$
Cons. of Mass	$k = 0$	ρv_i
Cons. of Momentum	$k = 1 \dots N_{dim}$	$\rho v_{k-1} v_i + \delta_{i,k-1} p$
Cons. of Energy	$k = N_{dim} + 1$	$\rho v_i H$

Table A.1: Inviscid (Euler) flux components.

The pressure (p), total specific enthalpy (H), and static temperature (T) are given by:

$$\begin{aligned} p &= (\gamma - 1)\left(\rho E - \frac{1}{2}\rho v_i v_i\right) \\ H &= E + \frac{p}{\rho} \\ T &= \frac{p}{\rho R} \end{aligned}$$

where γ is the ratio of specific heats and R is the gas constant. Lastly, $\delta_{i,j}$ is the Kronecker delta function, which takes value 1 if $i = j$ and 0 otherwise.

A.2 Viscous Flux

Component Name	Indices	$F^{vis}_{i,k}$
Cons. of Mass	$k = 0$	0
Cons. of Momentum	$k = 1 \ldots N_{dim}$	$\tau_{i,k-1}$
Cons. of Energy	$k = N_{dim} + 1$	$\kappa \partial_{x_i} T + v_j \tau_{i,j}$

Table A.2: Viscous flux components.

$\tau_{i,j} = \mu\left(\partial_{x_i} v_j + \partial_{x_j} v_i\right) + \lambda \delta_{i,j} \partial_{x_m} v_m$ denotes the stresses (shear and normal) for a Newtonian fluid. $\kappa = \frac{c_P \mu}{Pr} = \frac{\gamma \mu R}{(\gamma-1)Pr}$ is the thermal conductivity. The bulk viscosity is $\lambda = -\frac{2}{3}\mu$. The dynamic viscosity is given by Sutherland's Law, $\mu = \mu_{ref}\left(\frac{T}{T_{ref}}\right)^{1.5}\left(\frac{T_{ref}+T_s}{T+T_s}\right)$, with $T_{ref} = 288.15K$ and $T_s = 110K$.

A.3 Deriving the $\mathcal{A}$ Matrix

Observe that $F^{vis}_{i,k}$ depends on the state gradients $\partial_{x_j} u_l$ linearly. In particular, $F^{vis}_{i,k} = \mathcal{A}_{i,j,k,l}(\mathbf{u})\partial_{x_j} u_l$; utilizing this fact simplifies the implementation. All that remains is to derive $\mathcal{A}$. For simplicity, the following example will be in 2D. First consider the $F^{vis}_{0,k}$ term

(x-direction viscous flux) in the Navier-Stokes equations:

$$F_{0,k}^{vis} = \begin{bmatrix} 0 \\ \frac{2}{3}\mu\left(2\frac{\partial v_0}{\partial x} - \frac{\partial v_1}{\partial y}\right) \\ \mu\left(\frac{\partial v_0}{\partial y} + \frac{\partial v_1}{\partial x}\right) \\ \frac{2}{3}\mu\left(2\frac{\partial v_0}{\partial x} - \frac{\partial v_1}{\partial y}\right)v_0 + \mu\left(\frac{\partial v_0}{\partial y} + \frac{\partial v_1}{\partial x}\right)v_1 + \kappa\frac{\partial T}{\partial x} \end{bmatrix} \tag{A.1}$$

where v_i denote the spatial components of velocity.

Since $F_{0,k}^{vis} = \mathcal{A}_{0,0,k,l}\partial_{x_0}u_l + \mathcal{A}_{0,1,k,l}\partial_{x_1}u_l$, it is clear that $\mathcal{A}_{0,0,k,l}\partial_{x_0}u_l$ only involves $\frac{\partial \cdot}{\partial x}$ terms. Similarly, $\mathcal{A}_{0,1,k,l}\partial_{x_1}u_l$ only involves $\frac{\partial \cdot}{\partial y}$ terms. Additionally, Equation A.1 is written in primitive form. Converting to conservative variables requires rewriting the spatial derivatives; for example, $\frac{\partial u}{\partial y} = \frac{1}{\rho^2}\left(\rho\frac{\partial u_1}{\partial y} - u_1\frac{\partial \rho}{\partial y}\right)$, where $u_1 = \rho u$ refers to the second entry of the conservative state vector, $\mathbf{u}$. Then the matrices $A_{0,0,k,l}$ and $A_{0,1,k,l}$ follow:

$$A_{0,0,k,l} = \frac{\mu}{\rho}\begin{bmatrix} 0 & 0 & 0 & 0 \\ -\frac{4}{3}v_0 & \frac{4}{3} & 0 & 0 \\ -v_1 & 0 & 1 & 0 \\ \left(\frac{\gamma}{Pr} - \frac{4}{3}\right)v_0^2 + \left(\frac{\gamma}{Pr} - 1\right)v_1^2 - \frac{\gamma E}{Pr} & \left(\frac{4}{3} - \frac{\gamma}{Pr}\right)v_0 & \left(1 - \frac{\gamma}{Pr}\right)v_1 & \frac{\gamma}{Pr} \end{bmatrix}$$

$$A_{0,1,k,l} = \frac{\mu}{\rho}\begin{bmatrix} 0 & 0 & 0 & \\ \frac{2}{3}v_1 & 0 & -\frac{2}{3} & 0 \\ -v_0 & 1 & 0 & 0 \\ -\frac{1}{3}v_0v_1 & v_0v_1 & -\frac{2}{3}v_0 & 0 \end{bmatrix}$$

where κT has been expanded to $\frac{\gamma\mu}{Pr}\left(E - \frac{1}{2}v_iv_i\right)$ for convenience. A similar process produces $A_{1,0,k,l}$ and $A_{1,1,k,l}$:

$$A_{1,0,k,l} = \frac{\mu}{\rho}\begin{bmatrix} 0 & 0 & 0 & \\ -v_1 & 0 & 1 & 0 \\ \frac{2}{3}v_0 & -\frac{2}{3} & 0 & 0 \\ -\frac{1}{3}v_0v_1 & -\frac{2}{3}v_1 & v_0 & 0 \end{bmatrix}$$

$$A_{1,1,k,l} = \frac{\mu}{\rho} \begin{bmatrix} 0 & 0 & 0 & 0 \\ -v_0 & 1 & 0 & 0 \\ -\frac{4}{3}v_1 & 0 & \frac{4}{3} & 0 \\ \left(\frac{\gamma}{Pr} - 1\right) v_0^2 + \left(\frac{\gamma}{Pr} - \frac{4}{3}\right) v_1^2 - \frac{\gamma E}{Pr} & \left(1 - \frac{\gamma}{Pr}\right) v_0 & \left(\frac{4}{3} - \frac{\gamma}{Pr}\right) v_1 & \frac{\gamma}{Pr} \end{bmatrix}$$

Appendix B

Double Data Rate Random Access Memory

As of this writing, modern commodity DRAM modules are some type of DDR (Double Data Rate) memory. The original DDR is essentially obsolete. Many existing computers use DDR2 RAM, but newer platforms have transitioned to DDR3. DDR4 is expected to arrive in 2012. Other DRAM types (e.g., RDRAM, XDRAM) exist but are not as common. When purchasing DDR memory, it is common to see the following:

$$DDRxyPCxwa - b - c - d.$$

The x marks the type of DDR (DDR, DDR2, DDR3, etc.); it may be absent after the PC symbol. The y indicates the effective memory frequency (usually in MHz).[1] The w indicates the maximum theoretical bandwidth (in MB/s) of a single module of this memory. The sequence $a - b - c - d$ indicate the RAM timings, which contribute to its latency (see below). For example, DDR3 1600 PC 12800 5-5-5-16 is a stick of DDR3 memory, clocked at 1600MHz, with 12.8GB/s of bandwidth, and 5-5-5-16 timings.

RAM is composed of a large number of microscopic capacitors gated by transistors. If a capacitor holds sufficient charge, its value is 1; otherwise its value is 0. Due their

[1]The RAM is clocked "independently" of the CPU. Specifically, the RAM frequency is derived from the memory controller (described below) frequency.

small size and capacitance, the capacitors leak electrons and must be periodically refreshed. On modern DDR components, the refresh period is 64ms [20]. Note that this quantity is sometimes reported as 64ms divided by the number of capacitor rows (see below), resulting in values like 40 or 50ns.

To read a bit, the capacitor storing that bit is discharged. That is, the transistor completes a RC circuit with a "sense amplifier." The discharge time must be long enough for the sensing circuit to ascertain the difference between a 0 and a 1. The result is passed back to the memory controller. The amplified signal is also fed back into discharged capacitor, since a value of 1 must remain 1. Writing a bit involves discharging or charging a capacitor, depending on its initial state. Modeled as an RC circuit, charging require time. The relative slowness of these operations is the primary limitation in the speed of modern DRAM modules.

With one capacitor for every bit of data capacity, organizing the capacitors into a linear array is impractical [20]. Instead, the capacitors are organized into rows and columns and read in row-major order, like a two-dimensional `C` array. If there are N bits of memory, the row and column addresses only range over $\sqrt{N}$. When a read or write request arrives, it arrives with the row and column address of its data bits. To access a given capacitor, the appropriate row must first be opened, incurring a delay; only 1 row can be open at a time. This is analogous to dereferencing the first pointer in a 2D array. Then the appropriate column can be read or written, much like dereferencing the second pointer. Unlike array accesses, multiple columns can be read simultaneously. When a new row is desired, the currently open row (if there is one) must be closed; this process is called precharging, wherein the sense amplifiers are charged back to their idle, charged state. Precharging also incurs a delay. Memory operations are pipelined (not to be confused with CPU pipelining) so that different rows and columns can be accessed in subsequent cycles and to hide latency. This covers the time needed to read and write data, as the results are not immediately available.

Each set of rows and columns composes one DRAM chip; each DRAM module has several chips. DRAM communicates with the memory controller over the memory bus, which is usually 64 bits wide. Current DDR modules are organized into one or two *ranks* of 8 chips each, with each chip owning 8 bits of the 64 bit bus width.[2] To saturate the bus, as many column accesses as possible must happen per row access. All of the time taken for charging and discharging capacitors, opening rows, accessing columns, etc. contribute to the observed memory latency, which is described below. Note that the previous discussion is substantially simplified, but it captures all of the major features of DRAM operation.

B.1 Bandwidth and Latency

From a performance standpoint, all RAM types have two primary properties: bandwidth (in bytes per second) and latency (in *memory clock* cycles). The time required to communicate with memory follows $t = ax + b$, where x is the amount of data, a is the inverse bandwidth, and b is the latency. The inverse-bandwidth is typically much smaller than the latency, but this depends on the access pattern, as described below.

B.1.1 Bandwidth

The maximum theoretical bandwidth can be calculated from several fixed quantities combined with the effective memory frequency. The maximum theoretical bandwidth is reported by the manufacturer. Note that the maximum theoretical bandwidth is rarely attained in practice. Some synthetic benchmarking tools come very close but the access patterns used are contrived.

Bandwidth is the product of the memory frequency, the bus width (in bits), and the number of transmissions per cycle. First note that the effective memory frequency is double the actual frequency. So DDR3 1600 RAM is physically clocked at 800MHz. The effective frequency of 1600MHz arises from the ability of the various types of DDR to transmit data twice per memory clock cycle. As mentioned previously, the bus width is 64 bits (8 bytes). Thus the bandwidth of DDR3 1600 is $2 \times 800 \times 8 = 12800$ MB/s. The frequency of DDR modules varies, with higher frequency units being more expensive.

[2]Physically, single rank DRAM modules only have chips on one side, while double rank modules have memory chips on both sides. The memory controller can only access one rank of chips per cycle.

Modern RAM technology incorporates another performance-enhancing feature: multi-channel communication. Currently, dual and triple-channel technologies are available. Using RAM in multi-channel mode accesses several RAM modules simultaneously by interleaving the data (in the same fashion as RAID0 arrays). This technique doubles the theoretical bandwidth, although the practical performance improvement is nowhere near a factor of 2. It is important to use matched memory modules when running in multi-channel mode to avoid hardware incompatibility. Manufacturers sell memory meant for multi-channel use in matched sets.

B.1.2 Latency

Memory latency is usually broken down into four "timings": T_{CL}, T_{RCD}, T_{RP}, and T_{RAS} (described below). For a given memory operation, the relevant timings depend on the current state of the RAM module(s). These timings are controlled by the memory controller. Manufacturers report recommended timings, which are automatically detected by the memory controller.[3] The timings are reported as a sequence of four numbers, written as $T_{CL} - T_{RCD} - T_{RP} - T_{RAS}$. Their descriptions follow:.

- CAS latency, T_{CL}: Assuming the appropriate row is already open, this is the time between the RAM module receiving an access request and the request being fulfilled; e.g., the first bit of data being transmitted back.

- Row-Address to Column-Address Delay, T_{RCD}: The time required to open a new row of capacitors. If no rows are currently open, the time until the first bit of data is transmitted is $T_{RCD} + T_{CL}$.

- Row Precharge Time, T_{RP}: The time to precharge (close) a row. If the desired memory address requires changing rows, the time until the first bit of data is transmitted is $T_{RP} + T_{RCD} + T_{CL}$.

- Row Active Time, T_{RAS}: The number of cycles required to refresh a row. T_{RAS} is

[3]On some motherboards, the timings are user-selectable. Decreasing the timings too far can lead to system instability and data corruption.

usually equal to the sum of the previous three values plus a small integer. This form of latency affects performance if the controller wants to read from the row being refreshed.

The manufacturer recommended settings for these timings also varies by product. Lower latency modules are more expensive.

Each new generation of DDR memory has roughly doubled the previous generation's bandwidth. Latency is also doubled, but memory frequency doubles as well, meaning that the absolute latency (in seconds) has remained nearly constant from DDR to DDR2 to DDR3.

B.2 Memory Performance

As noted previously in Section 4.1, sequential memory access is much faster than random memory access. Assuming the row is already open, the first bit arrives T_{CL} memory cycles after the request. Sequential-read performance is then in large part governed by the bandwidth, assuming that the memory controller knows to sustain reading beyond the initial cache-line. Sustained reading hides the latency through pipelining. If not (e.g., access with stride larger than a cache-line), subsequent, sequential line-reads usually have latency T_{CL}. Infrequently, $T_{RP} + T_{RCD} + T_{CL}$ is also triggered by changing rows. When accessing randomly, the latency is always $T_{RP} + T_{RCD} + T_{CL}$ in the worst case and almost no speedup from bandwidth is possible.

Memory performance can vary widely even within the same class of RAM. These latency and frequency variations affect the latency (in CPU cycles) observed by the CPU. These variations partially explain why only approximate values could be reported in Tables 4.1 to 4.3; the other half of the issue lies with the memory controller.

The RAM latency is not the only factor in measuring the average latency from a miss in the largest cache level. When the CPU cache realizes that it does not have the desired data, it sends the request to the memory controller, a "chip" which interacts directly with the RAM. The memory controller may be part of a separate chip (called the Northbridge) sitting off-die, on the motherboard; however most new CPU designs integrate the memory controller directly onto the CPU die. Integrated memory controllers reduce memory latency for two reasons. First, the physical distance traveled by electrical signals decreases. Second, integrated memory controllers are clocked at (or near) the CPU frequency. Off-chip controllers are clocked at the same frequency as the Front-Side Bus (FSB).[4] Thus the integrated controller responds more quickly (and can also handle more operations).

AMD processors starting with the Athlon64 and Intel processors starting with the Nehalem (Core i7) use integrated memory controllers. The CPUs in Tables 4.1 to 4.3 all use the same RAM. Comparing the Core 2 (Table 4.1) to the K10 and Core i7 (Tables 4.3 and 4.2), one can see that the average latency difference between the off-chip memory controller (Core 2) and the integrated memory controller (K10, Core i7) is around 100 (CPU) cycles. Note that some part of this variation comes from the difference in CPU clock speeds, but these three CPUs are clocked similarly.

[4]On modern computers that still use an FSB, the FSB frequency is around 5-10 times slower than the CPU frequency.

Appendix C

The CPU

Modern CPUs have several performance enhancing features. Current CPUs are capable of Out Of [program] Order (OoO) execution, and execution is both superscalar and pipelined. Additionally, substantial effort has gone into the development of techniques such as branch prediction and prefetching to make the previously listed features more efficient. Briefly, a superscalar CPU is one capable of executing more than one instruction per clock cycle; this is handled by having multiple, redundant copies of certain circuits (e.g., floating point multiplier). Pipelining is often related to a factory assembly line: the CPU can be working on many instructions at the same time, using the same resources. OoO execution is the ability to reorder instructions on the fly when appropriate; i.e., programs need not be executed sequentially. We will now take a brief, high-level glimpse at how the CPU executes instructions.

C.1 Basic Data Flow

For a CPU executing instructions in-order, a few steps must occur for each instruction. Again, the instructions that the CPU is able to execute have a one-to-one mapping with the compiler's assembly output. The CPU has a special register called the *program counter* (PC) indicating the next instruction up for execution. The execution steps are then:

1. Fetch the next instruction from the L1i cache. Increment the PC by 1 instruction.

2. Decode the instruction.[1]

3. The instruction will have one or more operands. If these are available, then execute the instruction. Otherwise *stall* until the operands finish loading from cache into registers.

4. If the instruction was a load or store (to RAM), interact with cache, RAM, etc.[2] Otherwise, continue.

5. Write the result of the instruction to a register if needed.

These steps are diagrammed by a single row of Figure C-1.

Already a few points of inefficiency might stand out. First, each step in this process requires some amount of time. So while the execution step is doing say, a costly division operation, the instruction fetch, decode, etc. components are idle. Pipelining allows the potential for every component to be in use simultaneously. Secondly, in the execution phase, if the operands are not in registers, then our model CPU stalls. During the stalled time, the CPU is effectively doing nothing. OoO execution solves this problem by using several buffers to store stalled instructions, thereby allowing the CPU execution units to work on other code. This is incredibly beneficial, given how slow cache or even worse, RAM is.

C.2 The CPU Pipeline

The main idea behind the CPU pipeline is straightforward, as described previously: we want as many CPU components as possible to be constantly busy.[3] As diagrammed in Figure C-1, when the program starts, instruction A is fetched on the first clock tick and the PC is incremented. At the next clock tick, A is decoded and the fetcher gets instruction B.[4] Each step of the pipeline (fetch, decode, etc.) is called a stage. Modern CPU pipelines

[1]Assembly commands are broken down internally by the CPU into μops (micro-ops); μops are the atomic instruction unit used by the CPU. For example, an instruction for adding the contents of a register to the contents of a memory address is broken into two μops: load memory into a temporary register; then add two registers. Micro-ops improve the performance of the out of order engine (discussed below); they would not always be used in the simple model discussed here.

[2]In the execution step, load and store instructions calculate the memory address of the desired data.

[3]Long CPU pipelines are also necessary for higher clock speeds.

[4]B is the instruction immediately after A in the assembly code, since the fetcher only fetches an instruction and increments the PC.

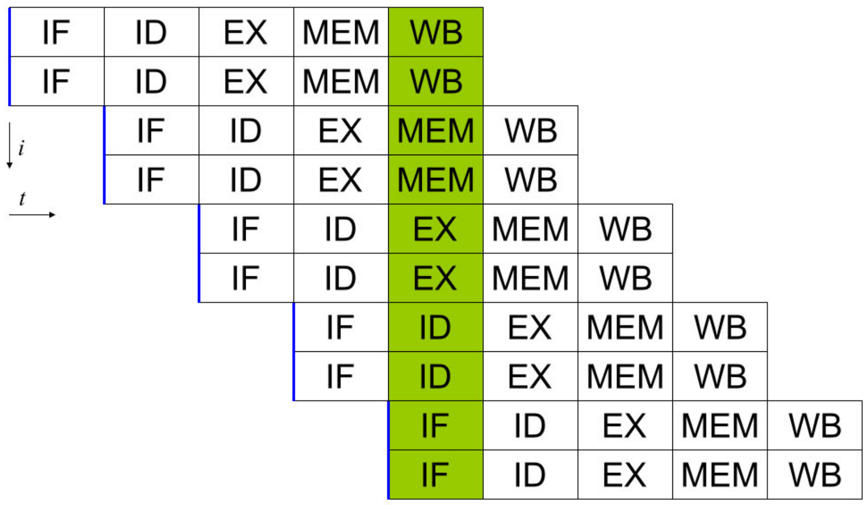

Figure C-1: A simple superscalar CPU pipeline with five stages that can execute at most two instructions per clock cycle. IF = Inst. Fetch, ID = Inst. Decode, EX = Execute, MEM = Memory access, WB = (register) Write Back. The horizontal axis shows physical time (clock cycles), and the vertical axis shows instruction count. Note that the block widths can vary widely from phase to phase. Figure credit: Wikipedia! ¡–How bad is this??

involve around 10 to 20 stages, so Figure C-1 is only a conceptual drawing. Many additional pipeline stages are related to out of order execution; see Section C.4. Exact values are not published by the manufacturers, but can be inferred through experiment; Fog[27] estimates 12 stages for the K8 and K10, 15 for the Core 2, and 17 for the Core i7.

Pipelining does not change the amount of time required for a given instruction, just as a factory assembly line does not decrease the build time for a single car. Rather, pipelining allows more instructions to be in flight simultaneously, potentially increasing the instruction throughput. In the worst case, pipelining allows no benefit and most stages remain empty.

While potentially beneficial, pipelining introduces several new challenges, including: branch misprediction, dependency chains, and data consistency[5] are among the major players. Of these, branch prediction is perhaps the most important. Further details are given in the next section; for now it is sufficient to know that a misprediction forces the entire pipeline to empty, stalling the CPU for a number or cycles equal to the length of the pipeline.

[5]Data consistency will not be discussed in detail. Consistency issues arise because assembly code is written (either by programmers or compilers) under the assumption that instruction I_j and all of its side-effects are complete before I_{j+1} starts. In a pipelined CPU, this is not always the case, since the goal of pipelining is to start executing I_{j+1} the cycle after I_j starts. CPU designers must guarantee that "nothing bad" happens.

Suppose we have a long *dependency chain*: instruction I_n depends on the output of I_{n-1}, which in turn depends on I_{n-2}, etc. Then the execution of I_n will always stall until I_{n-1} is ready. In such scenarios, we gain nothing from pipelining. It is up to the compiler and the programmer to minimize dependency chains. Note that while some dependency chains are logically necessary, others are not and arise solely from assumptions made by the compiler.

In evaluating `(a+b)/c-d`, `a+b` must be computed first; that result is then divided by `c`; and finally `d` can be subtracted from the intermediate `(a+b)/c`. Alternatively, in order to complete the store `A[i+j+k]=(a+b)/c-d`, both the RHS and the memory address must be computed before a store can occur. Here, dependencies are necessary: the CPU cannot divide by c until $(a+b)$ is computed; it cannot store the result until the address is computed; and so forth.

However, some dependency chains arise due to compiler assumptions. Under current compilers, the obvious implementation of a floating-point dot product between vectors of length N leads to a dependency chain of length N. A basic implementation follows:

```
double c = 0.0;
for(i = 0; i < N; i++)
  c += x[i]*y[i];
```

In this dependency chain of length N, iteration $i + 1$ cannot start until iteration i is done, because c must be updated after each iteration. Performance can be increased if the CPU is allowed to execute multiple iterations simultaneously. This is accomplished by unrolling the loop. An implementation of a inner product unrolled four times (assuming $N mod 4 \equiv 0$) follows:

```
double c, c0 = 0.0, c1 = 0.0, c2 = 0.0, c3 = 0.0;
for(i = 0; i < N; i+=4){
  c0 += x[i]*y[i];
  c1 += x[i+1]*y[i+1];
  c2 += x[i+2]*y[i+2];
  c3 += x[i+3]*y[i+3];
}
```

```
c = (c0 + c1) + (c2 + c3);
```

The reason that the compiler will not perform this automatically is that floating-point arithmetic is not associative. In this situation, the compiler is unwilling to make the transformation because it could change the answer.[6] Note that writing `c = (c0 + c1) + (c2 + c3)` is not the same as `c = c0 + c1 + c2 + c3`. `C` arithmetic is evaluated from left to right, so the latter expression is equivalent to `c = (((c0 + c1) + c2) + c3)`, which introduces an unnecessary dependency chain.

In general, loop unrolling can get rid of dependency chains, which is arguably its most important purpose. Regardless of whether dependency chains are present, unrolling can improve parallelism and may allow the compiler to produce better optimizations (e.g., improved instruction scheduling to avoid stalls, although modern CPU out of order engines make this less important). Unrolling also decreases the loop overhead, since fewer iterations are needed.[7] This overhead comes from evaluating the test condition, incrementing loop variable(s), and performing any other iteration-dependent arithmetic.[8] There are also downsides to unrolling. The loop size may be variable; in the previous example, if $N mod 4 > 0$, then extra code is necessary to handle the additional 1, 2 or 3 iterations, which increases the code size. Unrolling also increases the code size since an unrolled loop now has many copies of essentially the same operation(s). A larger code body may decrease the effectiveness of the i-cache, since the working set (in instructions) is larger. Unrolling a loop by an exceedingly large factor is potentially wasteful (in that the code becomes larger but with no performance gain), since there will not be enough registers to handle all operations in parallel. Be aware that explicit loop unrolling is often unnecessary, since most compilers have automatic loop unrolling mechanisms when setting at least `-O2` or specifying `-funroll-loops`. However, the compiler may make poor decisions (e.g., not unrolling or choosing an inappropriate factor) in places where it does not have as much information as the programmer. For example, the

[6]Note that loop unrolling to break the dependency chain may not the best choice with a vectorizing compiler. Current compilers will not compile the unrolled loop with SSE2 instructions (or more directly with SSE4's dot product instruction), but they can vectorize the vanilla code. It is likely that simple, common operations like inner products are recognized as such and replaced with more optimal code.

[7]For very short loops, unrolling can remove the loop construct entirely.

[8]Consider an inner product kernel: `c += (*x++)*(*y++);`. Unrolling decreases the amount of pointer arithmetic (incrementing `x` and `y`) involved.

state rank (N_{sr}) is constant for a given problem, but unless N_{sr} is specified at compile-time, the compiler must assume that it could lie anywhere in the range $[0, 2^{32} - 1]$.

Lastly, some instructions take much less time in execution than others; see Table 1.1. The CPU must guarantee that all results are written out in program order. Naively, the CPU could stall the output of instruction I_n until I_{n-1} is completed (and so on). As it turns out, the machinery necessary for out of order execution (see Section C.4) solves this issue.

C.3 Branch Prediction

A branch arises any time the `JMPQ` (jump) command[9] is issued in assembly, which causes the CPU to jump from the current instruction address to some other location.[10] Any branch is either *conditional* and *unconditional* and either *direct* or *indirect*. The following `C` control statements are examples of jumps: function calls, `if`-statements, `switch` blocks, control loops, and `goto`, `break`, `continue`, and `return` statements. Note that many control structures require at least two `JMPQ` commands. For example, one `JMPQ` is required to enter a branch of an `if`-statement, and another may be required to return to the code following that branch.[11]

Conditional branches include `if` and `switch` statements, where some expression must be evaluated and tested before the state of the jump is known. Unconditional branches include function-calls, where it is always known that the jump will occur.[12] An indirect branch is a jump whose target is not known at compile-time. With a direct jump, examining the jump (or call) assembly command will reveal a literal target address. With an indirect jump, the argument following `jmpq` could be the contents of a register, memory address, etc. In all cases, the target must be computed at run-time; generally, the target could differ each time the jump is executed. In C/C++, indirect jumps arise from `switch` statements, function

[9] `JMPQ` is the 64 bit version; the 32 bit version, `JMP`, may be more familiar.

[10] The `JMPQ` command accepts a known address (i.e., integer value) or the contents of a register as an argument. In x86 assembly, several other commands correspond to jumps; e.g., `JE` (jump if equal), `JNE` (jump if not equal), `CALL` (function-call), `RET` (return), etc.

[11] This is generally the case if an `else` is present; otherwise the exiting jump is not necessary.

[12] `JMPQ` and `CALL` are examples of unconditional jumps, whereas `JE` and `JNE` are conditional.

calls via function pointers,[13] and `virtual` function calls (polymorphism).

When a branch instruction is encountered in execution, the CPU nominally must know whether the branch is taken before proceeding. Additionally, if the branch is indirect, the CPU must also know its target. Given the frequency with which control statements, function calls, etc. appear, having the CPU wait to resolve every branch fully would incur a substantial performance hit. The solution is branch prediction, a phrase which refers to two different concepts: *branch prediction* and *branch target prediction*. The former refers to predicting whether a branch is *taken* or *not taken*. The latter refers to predicting the target (i.e., destination address) of a taken branch.

When a branch is mispredicted, the CPU will pipeline and/or fully execute[14] instructions that were never intended to pass to retirement. The damage must be undone before execution can continue, since the offending code may be trying to overwrite registers, memory, or otherwise change the program state. Recovery from the misprediction involves the CPU flushing the entire pipeline, since all currently in-flight instructions were executed as a result of the misprediction. As a result, branch misprediction latency is equal to the number of stages in the pipeline. Note that correctly predicted branches are not free. These branches are still decoded and their test conditions must be evaluated to determine correctness. They incur a 2 cycle latency, which is small compared to the misprediction penalty, but purely overhead nonetheless.

Branch mispredictions are murder for the performance of pipelined CPUs. The Pentium 4's extremely long (31 stages in some variants) pipeline lead to poor performance in applications with many branches, despite a very strong branch prediction mechanism. The Core i7 also features strengthened predictors (compared to the Core 2) to compensate for its pipeline which is "only" two stages longer than that of the Core 2.

For the pipeline to run efficiently, instructions are fetched in chunks (e.g., currently, 16 bytes) and decoded in the first few pipeline stages. To keep the pipeline full, these instructions should be executed sequentially. In all likelihood the branch target is not part

[13]This definition includes calls to members of dynamically linked libraries. However in this case, the target is always the same within a single run of the program, but it may vary from run to run depending on where the dynamic library is loaded in memory.

[14]Full execution only occurs in out of order processors; see next section.

of the most recently fetched code chunk.[15] A variety of programming practices can help reduce the number of branches along with their impact. For example, so-called "inner-loops" should avoid function-calls and `if`-statements whenever reasonable. Oftentimes, it is preferable to have mostly-redundant copies of inner-loop code. Each loop version lives in a separate branch instead of having branches within a single loop. Additionally, such inner-loops may also be unrolled so that several logical iterations are handled per executed iteration.[16] Additionally, loop unrolling is extremely effective when iterates are handled differently in a known pattern; e.g., all odd iterates take one path and even iterates take another. Here, unrolling can entirely eliminate branching in the loop body. Nested `if`-statements that branch on integers are also inefficient, unless a small number of possibilities are heavily favored and these possibilities appear early in the nesting. Otherwise, `switch` blocks are better.[17] Further, a look-up table is vastly preferable if the `switch` is only mapping integers to other numbers, since this avoids branching entirely. Decreased branching is also an argument for static linking over dynamic linking.

However in many cases, branching is unavoidable. CPU designers have worked hard to decrease the frequency of mispredictions. The most naive branch prediction approach is to always assume the next line of code will be taken, as described above with the incrementing of the PC. Luckily, CPU branch prediction algorithms are much more sophisticated. Efficient prediction is an area of much active research, but one algorithm is currently very popular[27]: the *two-level adaptive predictor*[71].

The two-level adaptive predictor is composed of several 2 bit *saturating counters*. A 2 bit saturating counter has four states: strongly not taken (00), weakly not taken (01), weakly taken (10), and strongly taken (11). Nominally each branch could be tracked by a single saturating counter. The branch starts in some state; each time it is taken, the counter increments, and each time it is not taken, the counter decrements. The counter cannot

[15]This is true of most branches. Only direct, conditional branches could continue execution in the current code chunk, depending on how the taken and untaken code is ordered by the compiler.

[16]The benefit from unrolling the dot product may be two-fold: it shortens the dependency chain and could decrease the loop overhead. Note that the loop overhead is only relevant when the dot-product is not memory-bound.

[17]In the worst case, `switch` statements are implemented with nested `if` statements to binary search over the cases.

decrement past 00 nor increment past 11. The CPU predicts not taken for 00 and 01 and taken for 10 and 11. Saturating counters perform well for branches that are mostly taken or not taken; branches that switch often will be mispredicted. Depending on the initial state, it can take up to two iterations for the counter to transition from not taken to taken or vice versa.

The two-level adaptive predictor solves this problem by storing the history of the last n occurrences of each branch, and applying a 2 bit saturating counter to each of 2^n history possibilities. The idea is best illustrated through an example; in the following, $n = 2$. Consider a branch that alternates between being not taken twice and then taken twice: 0011,0011,0011,.... After seeing the pattern at most 3 times,[18] the saturating counter for 00 will reach weakly/strongly taken, since 00 is always followed by 1. The counter for 01 will read 1, 10 reads 0, and 11 reads 0. After this steady state is reached, no further mispredictions occur. However, consider the pattern 0001,0001,0001,.... 00 may be followed by a 0 or a 1, meaning that the saturating counter for 00 has no steady state value. Generally, the behavior of the two-level method applied to a branch pattern with period p can be summarized as follows[27]:

- For $p \leq n + 1$, prediction is perfect after a transient of no more than $3p$.
- For $n + 1 < p \leq 2^n$, prediction is perfect if all consecutive subsequences of length p are different.
- Otherwise prediction is inaccurate.

In practice, remembering each branch separately is too expensive, since the data storage requirement is exponential in n. Instead, CPU designers use a global history table that is shared across multiple branches. The result is similar to what set associativity does to caching: different branches can be identified as the same, causing extra mispredictions if their patterns are different. Note that branches are assigned entries in the global history table based on their *instruction address*. Thus although all instances of `if(Dim == 2)` in a code segment may be taken, each individual `if` statement gets its own entry in the history

[18]The transient is due to the warm-up time required by the saturating counters.

table. Coalescing identical conditions can only be handled changing the code; e.g., by the programmer or the compiler.

Using the two-level adaptive predictor with global history, the K8 sets $n = 8$, the K10 sets $n = 12$, the Core 2 sets $n = 8$, and the Core i7 sets $n = 18$. The AMD K8 and K10 processors have $16 * 1024$ entries[5], but the size of Intel tables is not known through Intel or by experiment. But it does appear that the Core i7 has a two-level history table; the additional level (presumably) only includes branches that are determined to be more important (e.g., executed more frequently).

The two-level predictor generally performs quite well, obtaining average accuracy rates in the range of 95%[5, 36]. However, this method does not perform well with branches that are taken randomly. Fog[27] reports miss rates for various distributions of not taken/taken; e.g., branches that are randomly taken 40% of the time generate about a 46% miss rate. Randomly taken branches that vary infrequently (e.g., only not taken 1% of the time) are well predicted. The problem is that the two-level method attempts to assign patterns to inputs that have no pattern.

The latest Intel processors (Pentium M, Core 2, Core i7) use the two-level adaptive predictor for all branches except loops. This is somewhat wasteful since unconditional branches require no history to determine whether they are taken. AMD introduced a mechanism in its K8 processor that allows the CPU to discern between: 1) never jump, 2) always jump, 3) use two-level predictor, and 4) use return stack buffer (discussed below). As a result, some branches never enter the history tables. The mechanism is complex and full details are given in [27, 5]. The primary drawback is that only three branches per 16B (assembly) instruction block (aligned to quarters of a 64B cache line) can be predicted accurately. Having more than three branches will cause predictor contention, resulting in at least two mispredictions in that code section[27].

A few additional points remain, concerning special cases that do not fall under the prediction framework previously discussed: function returns, loops, and branch target prediction. Function returns are correctly predicted. During the call sequence, the return address (i.e., the instruction following the call) is pushed onto a stack called the return stack buffer; when the CPU encounters the return command, it simply pops the stack. Modern CPUs use a return stack with around 16 entries[27].

Loops are handled by an alternative mechanism in Intel processors starting with the Pentium M[36, 27]. These processors have separate loop counters for jumps determined to be a part of looping logic. The counters do not share storage with the two-level predictor's global history table. They can accurately predict loop counts of up to 64; for loops with more than 64 iterations, a misprediction is expected up to once every 64 iterations.[19] A meta-predictor mechanism decides when to apply the two-level predictor or the loop counter. Nested loops in particular are handled well by Intel's loop counter method; nesting remains an issue for other processors. Additionally, note that loops containing many branches tend to be mispredicted.

Having covered branch prediction, branch target prediction still remains as part of the CPU's overall prediction mechanism. A history of branch targets are stored in a special cache called the *Branch Target Buffer* (BTB). The BTB is usually set-associative, so aliasing issues (as with the global history tables) are possible. The BTB provides the CPU with information it can use to predict the branch target. A CPU with 100% accurate branch prediction would still have to stall and wait on indirect jumps, since knowing whether the branch is taken provides no information about what instruction to execute next.

The BTB is implemented as a hash table that hashes (jump) instruction addresses to branch targets. The BTB is generally not fully associative,[20] so conflicts can occur if jump instructions lie on certain size boundaries. Without a BTB, branch targets would have to be computed each time a branch arises (regardless of prediction accuracy), incurring extra latency for almost all branch instructions. Modern CPUs have at least two BTBs; one for indirect jumps and one for other jumps. BTB details[27, 5] are listed in Table C.1:

The Core i7 is not listed in Table C.1 because the data is unavailable[27]. As with its history table, the Core i7 also uses a two-level BTB[72].

Older processors generally predicted indirect jumps with non-constant targets inaccurately. Early BTBs only allowed one entry per jump instruction, so new a target for a branch

[19]By contrast, the AMD K8 and K10 processors will perfectly predict loops with 9 and 13 iterations, respectively.

[20]Associativity is not a published statistic. Fog experimentally estimates that the latest AMD and Intel processors use 4-way BTBs[27].

BTB Type	Entries	Set Associativity
Pentium M and Core 2		
Indirect Jumps	8192	4-way
Loops	128	2-way
Other Jumps	2048	4-way
AMD K8 and K10		
Indirect Jumps (K10 only)	512	?
Other Jumps	2048	4-way

Table C.1: BTB characteristics for various processors. For Pentium M and Core 2, "other" indicates non-looping branches and unconditional branches; for AMD, "other" indicates direct jumps.

was always predicted as its most recent target. Intel processors starting with the Pentium M and AMD processors starting with the K10 are able to predict non-constant indirect targets when there is a regular pattern. These designs employ a separate BTB for indirect jump targets. Indirect branches use the same history table, but the data organization is changed to use some bits to distinguish between different targets.

Lastly, the event counter `RESOURCE_STALLS.BR_MISS_CLEAR` measures the number of cycles stalled due to branch mispredictions. This includes the branch predictor making the wrong choice about a conditional jump as well as the BTB containing the wrong target data. When using unit tests to evaluate or simulate performance, take care to simulate all branches at all function-call levels associated with the inner-most loop. Since history table and BTB space are limited, it is not sufficient to emulate a loop with one function call as a loop with one unconditional branch; i.e., the branching behavior of that function call could adversely affect prediction performance elsewhere. This idea is similar to the fallacy of measuring performance by unit testing code that operates on large data sets with only small data sets.

In summary, branch misprediction is an expensive event which stalls the CPU for a number of cycles equal to its pipeline length. Branch (and target) prediction methods exist and generally perform very well. But these methods require warm-up time to "learn" expected branch behaviors. Additionally, branch predictors have limited space for storing histories and are subject to the same set-associativity issues seen with caches. Performance critical, inner-most loops should not involve branches whenever possible; i.e., avoid `if` statements, function calls, etc.[21] Inlining and macros can help maintain code readability and maintainability while abiding by this rule of thumb. Similarly, "core" parts of the program should also avoid large amounts of branching. For example, executing a large number of different branches infrequently will pollute the two-level adaptive predictor's history tables (the effect is worse since the tables are global) and the BTB. Thus improving branch prediction is not necessarily a local optimization; e.g., consider the following code:

```
for(i=0; i<n; i++){
  Foo(...);


BAR:
  //other branching code
}
```

If the function `Foo` contains (say) many nested `if` statements, especially with loops (or if there are many function calls), pollution of branch prediction resources could cause mispredictions in the code labeled `BAR`, even if `BAR` appears as if it should be well-predicted (i.e., follows the rules for the two-level adaptive predictor). The idea is similar the notion of maintaining small working sets for optimum cache effectiveness discussed in Section 4.1. In such inner-loops or core program components, consider removing branches by making known quantities compiler arguments and using pragmas such as `#if` or `#ifdef` or by lifting branches out of inner loops and having different (potentially largely redundant) code chunks for each branch option.[22]

C.4 Out of Order Execution

In the in-order, pipelined CPU model, if an instruction stalls (e.g., waiting to load data from the memory system), then the entire pipeline has to halt and wait. This limits the

[21] Possible exceptions to this rule of thumb include branches that detect special conditions where substantial computation can be saved.

[22] This practice has the advantage of making some loops static in size, allowing the compiler to optimize better; e.g., with unrolling.

effectiveness of pipelining and superscalar execution, since frequent stalls will result in most compute resources remaining at rest. By allowing *Out of Order* (OoO) execution, the CPU can work on another instruction during some resource stalls. In many cases, even new instructions dependent on the stalled instruction will go through the set up phase, so that they are ready to execute as soon as the stall ends. Additionally, the CPU can execute unrelated code blocks in parallel if enough instruction bandwidth is available. OoO execution improves the potential effectiveness of pipelining by trying to mask stalls that would otherwise shut down the entire pipeline. For example, OoO execution coupled with data prefetching usually masks the cost of L1 cache misses. OoO allows the CPU to perform unrelated operations during the wait period, and prefetching (from the L2 cache) attempts to prevent such misses from ever occurring. However, OoO execution introduces substantial levels of complexity not seen in processors with in-order execution.

The table below lists the major components of the pipeline in OoO CPUs. Recall that CPUs typically have 10-20 pipeline stages, so this list is only conceptual. In longer pipelines, the stages below would be divided into multiple smaller pieces.

1. Fetch the next cache line of instructions.[23] Branch prediction occurs at this early step.

2. Decode the instructions into μops.[24]

3. Assign μop operands to (renamed) registers, storing the assignments in the Register Alias Table (RAT). Stall if the table is full. Modern CPUs have more than 100 registers, even though the x86-64 ABI specifies far fewer; the RAT tracks the allocation of these registers to μops.

4. Create an entry in the Reordering Buffer (ROB) for the μop. Stall if the buffer is full. μops are decoded in program order; the ROB ensures that the results are also committed in program order.

5. Buffer μop in one of the Reservation Stations (RS). Stall if the stations are all full. RS

[23]Some CPUs place these instructions in a buffer in case the decode unit becomes stalled.

[24] only the case where copies of the operands move with the μops. This occurs on all current CPUs, but Intel's upcoming Sandy Bridge architecture diverges.

buffer μops so that if a particular μop cannot execute (applicable execution units are busy, operands unavailable, etc.), that μop can wait without stalling the entire CPU.

6. For each execution unit, execute a μop from the RS that has all of its operands available. If no such μops exist, stall.

7. If the instruction was a load or store (to memory), buffer it in the load or store buffer, respectively. (Recall that these buffers read and write out when bandwidth is available.) Stall the current load/store execution unit if the needed buffer is full.

8. Instruction retirement using the ROB; i.e., outputs are written out to registers in program order.

The new buffers and μop ports designed to support OoO execution are intended to reduce or eliminate the presence of *bubbles* in the pipeline. A bubble arises when pipeline stage s_i stalls. In the next cycle, s_{i+1} receives nothing from s_i, creating a bubble. Bubbles also arise when stages have different latencies, but these occurrences are rare. With the buffers in place, even if part of the pipeline wants to stall, hopefully the relevant buffer can hold the waiting instruction without delaying other work. In the assembly line analogy, buffering μops in the RS is akin to temporarily pulling a car frame off the line because some custom component is unavailable.

Register Renaming and the Register Alias Table (RAT) The x86-64 standard specifies only 16 64 bit integer registers,[25] 16 128 bit SSE registers,[26] and 8 80 bit x87 registers.[27]. There are also a number of special purpose registers (e.g., control, debug, test) that are not a part of typical program execution. Any program can only have direct access to these registers–generally speaking, 16 integer and 16 SSE. However, modern CPUs have upwards of 100 registers available. To promote pipelining and out of order execution, the CPU dynamically renames the registers used by the program to any one of the registers currently free

[25]`RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RBP`, `RSP`, `RFLAGS`, `RIP`, and `R8` through `R15`. Replacing `R` with `E` accesses only the bottom 32 bits.

[26]`XMM0` through `XMM15`. Upcoming AVX registers (256 bit) will be named `YMM0`, `YMM1`, etc.

[27]`ST(0)` through `ST(7)`; note that these registers are special in that they are implemented as a stack. These registers alias with the MMX registers, which are currently rarely used

in a process called *register renaming.* So accessing RAX in assembly could result in manipulating any one of the CPU's registers; this process is entirely transparent to the program. Register renaming confers several advantages. For one, independent code blocks can be executed simultaneously. Nominally these blocks use the same registers, but through renaming, they do not. Even within a single block of code, register renaming can mask instruction latencies[27]:

```
mov eax, [R8]
mov ebx, [R9]
add ebx, eax
imul eax, 6
mov [R10], eax
```

Here, [X] denotes the contents of the memory address contained in X. Suppose that [R8] is in cache but [R9] is not. Then the second mov command will stall. Notice that if multiple copies of eax were available, then that register could be saved off for the add command, while the imul and subsequent mov could proceed. The register renaming scheme automatically handles this operation by copying the contents of eax into a temporary register, thus performing additional work even while the load from [R9] stalls. The RAT tracks the renaming process and ensures that register values (as seen by the program) are changed in program order, despite actually executing out of order. The RAT has enough entries to track all rename-able registers (i.e., registers usable for computation). As mentioned, modern CPUs have more than 100 such registers.

ReOrder Buffer (ROB) The ROB maintains an ordered list of the instructions currently in flight. After instructions are executed, the ROB reorders the instructions so that they are retired in program order, i.e., in the order in which they were dispatched. When an instruction is decoded, it receives an entry in the ROB; that entry is cleared when the instruction is retired. Conceptually, imagine a linked list of instructions: newly decoded instructions are appended to the tail and instructions are retired by popping from the head. The ROB allows many instructions to be in flight simultaneously and out of program order.

For example, it is possible that every instruction in the ROB except the one at the head has been completed. Then when the head instruction completes, all current ROB entries will retire very quickly. Note that dependency chains do not directly interfere with the ROB in that every new entry added to the ROB could depend on its (temporal) predecessor. Indirectly, dependency chains decrease the efficacy of the ROB since it will never be true that instructions behind the head finish before the head. Note that the ROB may contain speculative results; these instructions are only retired if the speculation proves correct. There are 96 ROB entries in the Core 2, 128 in the Core i7, and 168 in Sandy Bridge.

Notice that the ROB provides a built-in mechanism for guaranteeing that branch mispredictions can be fixed relatively easily. The ROB guarantees that instructions retire in program order, so speculatively executed code cannot reach the head of the ROB until the jump condition and/or jump target have been computed and checked with the initial guess. In the case of a misprediction, the easiest solution would be to purge the pipeline and all of the buffers after retiring the branch instruction(s). Then restart execution at the correct location.

Reservation Stations (RS) Reservation stations (also called a unified or resource scheduler) buffer μops along with their operands.[28] Reservation stations communicate directly with the execution units. Instructions whose operands are not yet ready (e.g., instructions in a dependency chain) have RS entries as well, but with empty operand(s). These instructions cannot be sent off for execution until their operands are filled; instead, they wait in a station. Note that as long as stations are available, instructions can continued to be decoded and dispatched. One advantage of this system is that only very long dependency chains will stall the processor; shorter chains are not an issue since otherwise stalled instructions are buffered in a RS, allowing other instructions to continue executing. The RS contains 32 entries for the Core 2, 36 entries for the Core i7, and 54 entries for Sandy Bridge.

[28]Buffering operands is expensive in terms of power consumption and circuit-space due to the need to transfer wide operands around inside the CPU; e.g., on Core 2 and Core i7, the widest operands are 128 bits while the upcoming Sandy Bridge employs 256 bit operands. As a result, the Sandy Bridge stations will no longer hold entire operands; instead they will carry a pointer to a common cache of data called the Physical Register File.

Micro-ops and Execution Micro-ops are intended to give additional opportunities for out of order execution, since it is the μops that are reordered, not the instructions. Assembly commands have a one-to-one mapping with binary machine language, the instructions that are fetched by the CPU. These instructions can encode complex tasks. For example, the assembly command to multiply the contents of address X and Y, and then store the result in X is: `IMUL [X] [Y]`. This instruction is broken into four μops: 1) load X into a register, 2) load Y into a register, 3) multiply and store the result in a register, and 4) update X with the result. Consider the code:

```
IMUL [X] [Y]
IMUL [Z] [X]
```

Suppose that Y and Z must be loaded from cache, while X is in a register. Without breaking into μops, the first instruction stalls waiting for Y and there is nothing more to do. With μops, the loads for Y and Z can proceed simultaneously.

Execution units receive μops on different ports. Only one instruction can be issued to a given port per clock cycle; similarly, each port can only complete one μop per clock cycle. The latest Intel CPUs have 5 ports. Ports 2, 3, and 4 handle memory operations almost exclusively. The other ports handle a mix of computation operations. A full listing of assembly instructions with their associated port(s) and latencies is given in [26]; a partial listing is given in Table 1.1. Micro-ops are assigned to ports in a way that groups μops with the same latencies onto the same port. Micro-ops that for which there are multiple execution units (i.e., superscalar) can be received on multiple ports; e.g., adds, bit-wise operations, and moves (between registers) on integers are issuable on ports 0, 1 or 5.

Out of Order Restrictions Modern CPUs are not truly out of order, in particular with respect to reordering memory accesses. CPUs that are fully out of order with respect to loads and stores experience hazards such as Read After Write (RAW).[29] Alternatively, the CPU is free to reorder writes and reads to different addresses. Intel and AMD provide exhaustive

[29]RAW hazards occur when a program specifies a read to address X and then a write to address X, but OoO execution caused the write to retire *first*. The read will load the wrong value, and all instructions depending on that read must be invalidated.

lists describing when the reordering of memory operations is disallowed[39, 5]. As a result of the limit on memory reordering, modern CPUs have load and store buffers (discussed in Section 4.1) to enable the queuing of load and store operations, thereby masking (to some extent) the effect of resource limitations in the memory hierarchy. Since all read and write operations are buffered, these buffers can fill (causing `RESOURCE_STALLS.LD_ST` events) even if all memory accesses hit in cache. This happens when a large number of read and write instructions occur near each other (particularly if accessing entries different cache lines so that coalescing does not occur), preventing the CPU from reordering them with other operations and/or each other. Thus, the `RESOURCE_STALLS.LD_ST` counter is not a sufficient descriptor of memory-limited workloads on its own.

The last OoO pipeline stage requires some additional attention. CPUs are often allowed to execute instructions speculatively. A speculative CPU may execute code later determined to be unnecessary (e.g., prefetching) or incorrect (e.g., branch misprediction). Unnecessary μops simply pollute the various OoO resources with meaningless code. Incorrect code pollutes these resources with μops which must be purged once the misprediction is identified. So there is a distinct difference between executed instructions and retired instructions. Retired instructions are instructions where the output was saved: only retired instructions did any useful work. The ROB guarantees that instructions are retired in program order, despite completing execution in nearly arbitrary order. In general, the number of instructions decoded and executed by the CPU will be larger than the number of instructions retired, due to techniques like speculative execution. The `INST_RETIRED` event counter indicates how many instructions were retired. Again, retired instructions only count "useful" work, not speculatively executed code that proved unnecessary. `MACRO_INSTS` counts the number of instructions that were decoded into μops, including speculative operations.

Lastly, there are event counters for every major OoO pipeline component: `RAT_STALLS`, `RESOURCE_STALLS.ROB_FULL`, `RESOURCE_STALLS.LD_ST`, and `RESOURCE_STALLS.RS_FULL`. In compute-bound workloads (i.e., all memory accesses hit in L1 cache), ROB and RS resources do not usually run out. This may happen if most memory operations work with data already in registers. In memory-bound workloads, these resources are likely to fill up. Consider a situation where every newly decoded instruction requires some data that is currently not in registers, so that data must be retrieved from somewhere in the memory hierarchy. Suppose also that most upcoming instructions depend on the stalled instruction, directly or indirectly. The CPU runs out of reservation stations because the RS fill with instructions whose operands are not yet available, preventing any further instructions from passing to the execution units. The ROB becomes full because each new instruction is added (at its tail), but no instructions can be retired (from the head). The RAT fills because all scratch registers are already assigned to instructions waiting on the stalled instruction. Considering that modern CPUs can decode around 4 μops per cycle and last-level cache misses incur latencies of 100-200 cycles, it is easy to see why such misses are so damaging. Generally, stalls due to resources filling earlier in the pipeline are worse, since larger amounts of the pipeline will sit idle. For example, if the RAT fills, no μops will pass beyond it. However, if the reservation stations are full, no further μops can leave the ROB, but the fetch, decode, etc. machinery remains active unless the ROB is also full. Measuring the severity of memory bounded-ness is not merely a function of the number of cache misses (compared to total instructions retired). The actual performance impact of these misses is better quantified by stalls due to the RAT, ROB, RS, and LD and ST buffers.

Appendix D

GMRES Stopping Criterion

When seeking a steady-state solution to a complex flow problem, it is common practice to start with a small "time-step" to improve the robustness of the nonlinear solve (i.e., Newton's Method). Across most nonlinear iterations where the residual-norm is large, the accuracy of the linear solve has virtually no effect on the performance of the solver. That is, if the current nonlinear residual norm is 10^{-1}, then obtaining a residual of 10^{-16} on the linear problem is wasteful. However, an accurate linear solve is required as the nonlinear solver nears the stationary point; errors here would prevent quadratic convergence. This argument is based on a mathematical heuristic[18].

On the other hand, there is no guarantee that quadratic convergence is always desirable. In particular, the high-accuracy linear solves required in the final stages of the nonlinear solver are the most expensive. If the residual assembly cost is substantially less than the cost of these high-accuracy linear solves, then one possibility is to sacrifice quadratic convergence entirely. By using less accurate solutions, the nonlinear solver will require more overall iterations but the total cost could nonetheless decrease. In the following, a heuristic, adaptive stopping criterion [18] is described, and a simple "fixed" alternative is given.

GMRES should only solve the linear system accurately enough for the nonlinear solver to obtain Newton convergence. Let $e_h^m = U_h - U_h^m$ be the solution error at nonlinear iteration m. Newton methods exhibit quadratic convergence for sufficiently small error; i.e., $\|e_h^{m+1}\| =$

$C_1\|e_h^m\|^2$. Assuming the same behavior in the nonlinear residual,[1] R_h, the residual could decrease by at most:

$$\frac{\|R_h(U_h^{m+1})\|}{\|R_h(U_h^m)\|} \sim \left(\frac{\|R_h(U_h^m)\|}{\|R_h(U_h^{m-1})\|}\right)^2 = (d^m)^2$$

So if $(d^m)^2$ is large, the linear solve need not be very accurate. After the linear residual (on the k-th iteration), $r_h^k = b_h - A_h x_h^k$, has decreased by:

$$\frac{\|r_h^n\|}{\|r_h^0\|} \leq K_A \min(1, (d^m)^2)$$

GMRES should halt. Far from convergence, there is no guarantee that the nonlinear residual norm decreases monotonically; hence the presence of min. K_A is a safety factor; we find that a value of 10^{-3} is appropriate. When $\min(1, (d^m)^2) \approx 1$, setting K_A too large (e.g., $K_A = 1$) results in insufficient accuracy, providing garbage output to the nonlinear solver and preventing convergence. Finally, if $(d^m)^2$ is extremely small, the following modification ensures that GMRES does not attempt an impossible task:

$$\frac{\|r_h^n\|}{\|r_h^0\|} \leq K_A \max\left(\min(1, (d^m)^2), K_2 \frac{\|R_h^{desired}\|}{\|R_h^m\|}\right),$$

where $\|R_h^{desired}\|$ is the desired nonlinear residual and K_2 is another safety factor set at 10^{-1}. We have found that this method requires the same number of nonlinear iterations as always solving the linear system to machine precision. But the cost of all but the last few linear solves (near convergence) is decreased greatly.

In practice, we have found that using the non-adaptive (fixed) stopping criterion,

$$\frac{\|r_h^n\|}{\|r_h^0\|} \leq K_F,$$

is faster. We set $K_F = 10^{-3}$. The F subscript differentiates the fixed scheme from the adaptive scheme, which has parameter K_A. This criterion incurs additional nonlinear iterations

[1]The assumption that the residual predicts the behavior of the error is heuristic, based on the fact that the two differ by at most a factor of $cond(A)$ when the problem is linear.

near convergence, since the inaccurate solves prevent quadratic convergence. However, each iteration is cheaper. The K_A, K_F factors are often called the *forcing term*; see [63] for some further discussion and references.

Finally, regardless of the GMRES termination criterion, the importance of right preconditioning is worth emphasizing. Testing convergence with the (linear) residual norm under left-preconditioning leads to specious conclusions; only right preconditioning yields meaningful residual norm values.[2] Left-preconditioned GMRES naturally produces the value $\|M^{-1}\left(Ax^m - b\right)\|_2$ after the m-th iteration. This quantity includes a factor of M^{-1}, whose properties are often unclear. right preconditioned GMRES naturally produces $\|Ay^m - b\|_2$, where $y^m = M^{-1}x^m$. Tracking changes in $\|Ay^m - b\|_2$ is a more accurate representation of the progress made by GMRES.

[2]Calculating $\|Ax - b\|_2$ explicitly in every GMRES iteration also yields meaningful results, but the cost is prohibitive.

Appendix E

Code Development Practices for Performance

The following section lists out some programming tips for performance enhancement. Some techniques (e.g., cache-blocking, loop unrolling, branch elimination) were already discussed elsewhere in this thesis and will not be repeated here. The list is loosely organized into categories of related techniques. Some techniques are specific to `C` and `C++`, but most are general.

- Compiler and Language:
  - `C` modifiers: Two of the most useful modifiers for scientific codes are described below. The description for the first modifier also includes a review of how to interpret `C` modifiers.
    - Use of the `const` keyword: This keyword is part of the original `C` standard. `const` is short for *constant*; its purpose is to declare the modified variable *immutable*. `const` is useful for adding code clarity, particularly to function argument lists with pointers, where it is often unclear which referenced data will be modified. `const` variables must have their values assigned at declaration.

      As a warning, using the `const` keyword forms a contract with the compiler: the programmer promises not to modify the constant variable. If the compiler detects a violation, warnings or errors will be produced. However, through pointer manipulation, it is not hard to modify `const` variables without the compiler's knowledge. Thus, most compilers will not optimize based on the `const` keyword.

Recall that `C` modifiers are read from right to left. For example, consider:

```
double myArray[10] = {0}; //initialize myArray to 0
int const x = 10; //x, a constant integer with value 10.
double * const A; //A, a constant pointer to double.
                  //The value of A (e.g., A=NULL)
                  //can never be set.
double const * B; //B, a pointer to constant double
                  //Thus B = myArray; is valid, but
                  //B[3] = 5.3; is not!
double * const * C; //C, a pointer to const pointer to double
                    //The value of C and C[i][j] can be
                    //changed, but C[i] (for any i)
                    //refers to constant pointers
double const * const * D;
//D, a pointer to const pointer to const double
//D may be modified, but D[i] and D[i][j] cannot be modified.
```

Note that the assignment `C=E` is only valid if `E` is has the same declaration as `C`; this is similarly true of assignments to `D`.

Note that adding describing pass-by-value arguments as `const` is redundant and unnecessary. For example, in:

```
int foo(int const n, double const * A, double * const x);
```

only the second argument should have the `const` modifier, if the array values are indeed inputs only.

∗ Use of the `restrict` keyword: This keyword was added in the `C99` standard. In `gcc`, it can be enabled with the options `-std=c99` or `-std=gnu99`; in `icc`, it is enabled with `-restrict`. The restrict keyword applies only to pointers:

```
double * restrict x;
x = malloc(...); //say x has 100 entries
//code to initialize x
```

The `restrict` keyword promises the compiler that no other pointers will write-alias x and x will write-alias no other pointers. That is, x will not write to and read from regions of memory that other pointers will also write to and read from. For example, consider:

```
memset(x,0,100*sizeof(double));
double * restrict y = x+50;
for(i=0; i<50; i++){
  x[i] = 1.0;
  y[i] = 2.0;
}
```

as compared to:

```
memset(x,0,100*sizeof(double));
double * restrict y = x;
for(i=0; i<50; i++){
  x[i] = 1.0;
  y[i] = 2.0;
}
```

In the first loop, x and y do not write-alias each other. x only works with the first 50 numbers; y only works with the last 50. In the second example, x and y do write-alias each other, and the initialization of y breaks the `restrict` contract.

As with `const`, `restrict` is only a promise to compilers that the restricted pointer will not write-alias any other pointers and that it will not be write-aliased by any other pointers. Breaking the contract purposefully or inadvertently will lead to undefined behavior. In the second code block above, the first 50 values of x will be an unpredictable mix of 1.0 and 2.0, since the contract was broken. Note that the compiler has almost no chance of detecting broken `restrict` contracts, so warnings will be given rarely.

Consider another example:

```
void someprod(double * x, double * y, int n){
  int i;
  for(i=0; i<n; i++)
    x[i] = y[i]*y[i];
  return temp;
}
```

On the surface, it seems like the n iterations of this loop could occur in parallel. However, this is not the case. Consider:

```
double y[10] = {...};
double * x = y + 1;
someprod(x,y,9);
```

Here, the code effectively being executed is `y[1] = y[0]*y[0]` followed by `y[2] = y[1]*y[1]`, etc. since `x[0] == y[1]` and so forth. Thus each iteration is dependent on the previous iterations. Most programmers would never intend for `someprod` to be used like this, but the compiler cannot make that assumption on its own. The n iterations of the loop in `someprod` are only independent if `x,y` are restricted. A better declaration follows:

```
void someprod(double * restrict x,
              double const * restrict y, int n);
```

Functions like matrix-matrix and matrix-vector products also benefit from the use of the `restrict` keyword. Without it, a product like $y+=Ax$ could have x, y sitting somewhere inside of A. This introduces many dependency chains that are, in most situations, unintended.

Finally, [54] noted speedups of 10% to 20% overall in their finite element code after adding the `restrict` keyword. Comparable gains in ProjectX components have also been observed.

  - Use builtin functions: For the most part, this is automatic. For example, when calling many standard math functions such as `sin` or `fabs`, the compiler automatically replaces calls to reference `libc` implementations with optimized versions. For example, modern CPUs have a specific assembly instruction for `fabs`. In some cases, this may not be automatic. The three main memory manipulation functions in `C`, `memset`, `memmove`, and `memcpy`, are one such example. In these situations, the optimized implementations will use SSE instructions and switch to non-temporal moves as needed (for large data). In fact, current compilers will recognize

    ```
    for(i=0; i<n; i++) A[i] = 0;
    ```

    and replace it with a (possibly inlined) call to `memset`. This mechanism fails with more complex loops, such as:

    ```
    for(i=0; i<n; i++){
      A[i] = 0;
      B[i] = 0;
    }
    ```

    since the loop technically specifies that the i-th value of `B` be set immediately after the i-th value of `A`. Finally, note that calls to these three memory functions are undesirable for very small amounts of data (less than 100 bytes).

  - Compiler options: Table E.1 contains a list of compiler options useful with both `gcc` and `icc`. Table E.2 contains a list of options unique to `icc`, and Table E.3 contains options unique to `gcc`. Outside of the commonplace -O2 or -O3 options, these compilers have a number of additional, performance-boosting options that may not be as well-known. Finding an appropriate set of compiler options can substantially improve performance, whereas setting the wrong options can worsen performance.

- Write and execute code in a 64 bit environment: Unless legacy compatibility is of

Option Name	Compiler	Comments
-On	both	$n = 0, 1, 2, 3$, higher for more optimizations. Enables automatic optimizations for both compilers. -O2 is common.
-falign-functions=16	both	Align functions on 16-byte boundaries. Improves i-cache performance.
-falign-jumps=16	both	Align jumps on 16-byte boundaries. Improve i-cache performance.
-finline	both	Inlines functions declared with the `inline` keyword.
-finline-functions	both	Inlines functions as the compiler sees fit.
-fomit-frame-pointer	both	Free up frame pointer register for general use. Adds a scratch register.
-fpack-struct	both	Pack `struct`s to occupy as little space as possible; see below.
-fshort-enums	both	Uses minimal amount of space for `enum` types, thus reducing memory use and improving cache performance.
-freg-struct-return	both	Return `struct`s in registers when possible.
-march=?	both	Specify target architecture; e.g., replace ? by core2. Implies -mtune=?

Table E.1: Useful compiler options for `icc` and `gcc`. Consult the compiler manuals or references for further details on these options.

Option Name	Compiler	Comments
-alias-const	`icc`	Compiler assumes pointers to `const` cannot alias pointers to non-`const`.
-align	`icc`	Compiler modifies alignment for variables and arrays to improve performance.
-falign-stack=maintain-16-byte	`icc`	Maintain 16-byte stack boundaries.
-fast-transcendentals	`icc`	Use fast transcendental functions that at worst lose precision in the last decimal place.
-fma	`icc`	Fused Multiply Adds: floating point multiplies followed by addition (or subtraction) are combined into one operation.
-ftz	`icc`	Flush denormalized numbers to 0. Avoids penalty of working with denormalized numbers.
-no-prec-div	`icc`	Allows expensive divisions can be replaced by cheaper multiplication by reciprocal operations.
-no-prec-sqrt	`icc`	Use a fast sqrt function that at worst loses precision in the last decimal place.
-opt-multi-version-aggressive	`icc`	Compiler checks for pointer aliasing and scalar replacement.
-opt-subscript-in-range	`icc`	Assume array indexes never overflow.
-vec	`icc`	Compiler vectorizes loops and code blocks when it sees fit. Needs "-fp-model fast=1" for maximum efficacy.
-vec-guard-write	`icc`	Performs conditionals inside vectorized loops, avoiding additional stores.

Table E.2: Useful compiler options for `icc`. Consult the compiler manuals or references for further details on these options.

Option Name	Compiler	Comments
-fassociative-math	`gcc`	`gcc` assumes that floating point math is associative. Requires -fno-signed-zeros and -fno-trapping-math
-ffinite-math-only	`gcc`	Assumes `nan` and `inf` will never arise.
-fno-math-errno	`gcc`	Floating point exceptions do not set the `errno` flag.
-fno-signaling-nans	`gcc`	Signaling `nan`s disallowed.
-fno-signed-zeros	`gcc`	Positive and negative 0 are the same.
-fno-trapping-math	`gcc`	Disallows overflow, underflow, division by 0, and other floating point exceptions. Increases speed if these never occur but generates invalid results otherwise. Requires -fno-signaling-nans.
-freciprocal-math	`gcc`	Allows expensive divisions can be replaced by cheaper multiplication by reciprocal operations.
-ftree-vectorize	`gcc`	Enables automatic vectorization. Requires -fassociative-math to vectorize floating point arithmetic.
-mfpmath=sse	`gcc`	Use SSE floating point instructions (instead of x87).
-mpreferred-stack-boundary=4	`gcc`	Sets $2^4 = 16$-byte stack boundaries.

Table E.3: Useful compiler options for `gcc`. Consult the compiler manuals or references for further details on these options. Note that the last several floating point math optimizations (names starting with -f) for `gcc` are contained within the options -ffast-math and -funsafe-math-optimizations, the former of which includes the latter and some options not listed in this table.

utmost importance, 32 bit operating systems and CPUs offer no advantages over their 64 bit brethren. Oft-cited examples include: 1) 32 bit environments can only address 4GB of RAM in a single process; 2) 32 bit environments allow only half as many registers (8 compared to 16 for 64 bit); 3) the 32 bit (function) calling convention has far fewer registers available for passing arguments, resulting in relatively slow stack references; and 4) SSE instructions (and in particular, the important floating point SSE2 ones) are not available in 32 bit mode.

- Vectorization/Use of SSE instructions: As mentioned above, current compilers can automatically vectorize code if the proper options are specified (-vec for `icc` and -ftree-vectorize for `gcc`). Vectorization via SSE instructions is extremely important for performance, particularly with floating point operations. Vectorized code can conduct multiple floating point operations with a single instruction, effectively doubling (for `double` data) or quadrupling (for `float` data) throughput. With the AVX extensions available on Intel's newest (Sandy Bridge) processors, these performance factors are again doubled.

  However, some loops or code blocks are sufficiently complex so as to defeat the compiler's vectorization algorithms. Alternatively, some loops may not be vectorizable without specific assumptions that the compiler does not know. The latter issue can be handled with `#pragma` compiler hints. In highly performance-critical code regions, the former issue is only addressable by the programmer manually vectorizing code. Fortunately, this does not require writing assembly. Libraries such as `mmintrin.h` provide hooks into SSE assembly instructions. These libraries allow programmers to access SSE instructions directly in the familiar `C/C++` environment, with no overhead.

- Improving branch prediction:

  - `if` vs. `switch` statements: Generally, `switch` statements are preferable when selecting over more than 2 integer options. First, a description of how the compiler implements `switch` statements is in order. `switch` statements have a set of target cases ranging from a to b. Not every integer between a and b needs to be specified. The assembly code for `switch` statements takes one of two forms. First, if more than about 10% of the integers between a and b have specified `case`s, the compiler forms a *jump table*. This is an array of memory addresses (containing instructions); depending on what argument `switch` receives, execution jumps to the appropriate memory address by effectively doing an array lookup. The unspecified integers may be automatically inserted by the compiler into the jump table (with jump instructions pointing to the default condition or to skip past the `switch` statement) or handled with one or more `if` statements. When too few targets between a and b are specified, the compiler builds a set of nested `if` statements that perform a binary search for the desired target.

Nested `if`-statements must evaluate every conditional to reach the deepest level. As described above, `switch` statements evaluate at worst a number of conditionals logarithmic in the size of the range of options (i.e., $b - a$). At best, the `switch` statement involves simply an array-lookup. Thus, `switch` statements usually outperform `if` statements when their use is possible.

Note that neither `if` nor `switch` statements are appropriate for mapping an integer to another number. In this case, the programmer should create a (`static`) lookup table. This avoids the branching altogether.

If a small number of `case`s are much more common than the others, it is advantageous to "lift" these cases out of the `switch` statement and test them with one or more `if`-statements. These common targets should be ordered from most to least common in set of `if`s. `__builtin_expect` (discussed below) can handle this automatically if only one `switch` target is commonplace. `switch` statements incur more overhead than `if` statements if the `if` statement is guaranteed to only need one or two comparisons.

– The `__builtin_expect` hint: In the following code example, the resulting compiled code will have `Code A` followed by `Code B` followed by `Code C`, all adjacent to each other. In execution, if the branch is taken, the conditional jump results in continued execution of already-fetched code. However, if the conditional is very unlikely, then a jump into block C is always necessary, wasting bandwidth in loading B. Thus placing `Code B` outside the main code body achieves better performance through improved locality.

```
...Code A...
if(conditional){ //unlikely conditional
  ...Code B...
}
... Code C...
```

Many modern compilers accept hints to indicate that a conditional is likely or unlikely. Given the hint, the compiler will move code around accordingly to improve instruction cache performance. For `icc` and `gcc`, the "function call" `__builtin_expect` is used to provide these hints[38, 66]. These compilers also accept `__builtin_expect` in the `switch` clause, where it indicates which case is the most likely.

- Data Type Limitations and Data/Code Alignment:
  - Avoid floating point special cases: Avoid floating point (either x87 or SSE) underflow and overflow exceptions; e.g., denormalized numbers, `inf`, and `nan`.[1] The execution units responsible for floating point arithmetic cannot directly handle these special cases; the resulting execution interruption carries a latency of around 100 cycles. Thus, using `inf` or `nan` as flag values is unwise. Additionally, a floating point system supporting denormalized numbers is not necessary for proving commonly considered numerical properties like backward-stability. In most cases, algorithms that are strongly dependent on denormalized numbers should be resdesigned.
  - Typecasting: [28] provides a complete listing of the various kinds of type casts. A brief summary is given here. In short, not all type casts are free and some are quite expensive, so care is needed when casting. First, casting from any type

[1]Integer overflow and underflow do not cause slowdowns by themselves.

(e.g., `signed, unsigned`) or size (e.g., `short`, `long`) of `int` to any other type of `int` is free. The cost of casting from integer to float (or vice versa) depends on whether x87 or SSE2 floating point instructions are being used. Lastly, `float` to `double` (or vice versa) is not free, since a 32-bit `float` is not simply the lower half of a 64-bit `double`, as with 32 and 64-bit integers.

– Use the smallest type for the job: Choose `char` and `short` (available `signed` and `unsigned`) when the data range is sufficiently small.

– Data alignment: The Table (E.4) below lists the alignment boundaries for common data types. These boundaries are important when laying out `struct`s. Note that the given boundaries are OS and hardware dependent; the following table specifies alignments used in 64 bit Linux.

Type	Size (bytes)	Alignment (byte-boundary)
`char`	1	1
`short`	2	2
`int`	4	4
`float`	4	4
`double`	8	8
`long int`	8	8
Any pointer	8	8
`long double`	16	16

Table E.4: Alignment of Primitives in Linux64

Finally, as mentioned in Table E.1, stack alignment can be specified through compiler options. For best performance of SSE instructions, 16-byte boundaries should be used. Current implementations of `malloc` usually default to 16-byte boundaries for heap-allocated data. Specialized `malloc` implementations (e.g., `mm_malloc`) that allow programmers to specify data alignment also exist.

– Pack `struct`s: The compiler orders data in `struct`s in the order defined in code, unless `-fpack-struct` is given. With -fpack-struct, the compiler automatically orders `struct` members to occupy as little space as possible. To see how this works, consider: `struct`:

```
struct mystruct{
  char charA;
  double doubleB;
};
```

Looking at Table E.4, there are only 9 bytes of data here. But `double` must be 8-byte aligned, meaning that in memory, 7 bytes of padding will be inserted after the `char`. Thus `sizeof(mystruct)` is 16 bytes. If the positions were flipped:

```
struct mystruct2{
  double doubleB;
  char charA;
};
```

then `sizeof(mystruct)` is 9 bytes. This problem is compounded if the programmer were to form an array of `mystruct` types.

However, it is sometimes preferable to waste some space in very large `struct`s. If some `struct` members are accessed more commonly than others, the most commonly accessed members should be grouped together and placed at the beginning of the `struct`. This reordering improves cache locality. Similarly, if a programmer were using an array of `mystruct2`, but the `char` is accessed more often than the `double`, cache performance would be improved by breaking `mystruct2` apart.

– Code alignment: It is often preferable for jump targets (e.g., function bodies, loops, etc.) to be aligned in memory. This is related to how the CPU fetches instructions; fetches of unaligned instructions will grab irrelevant code, wasting bandwidth. Compiler options, in the form of `-falign-*`, exist to accomplish this automatically; see Table E.1. Compilers align code by inserting instructions that do nothing (no-ops). The potential disadvantages of code alignment include growth in code size and wasted CPU cycles from executing no-ops. Generally, -falign-functions and -falign-loops are suggested. There are two other options, -falign-loops and -falign-blocks; these usually do more harm than good.

- Keep function argument lists short in oft-called functions: In 32 bit mode, all function arguments are pushed onto the stack (pre-call) and popped off again before the first line of the function executes. In 64-bit mode,[2] 8 `float`, `double`[3], or SSE data types (e.g., `__m128`) in addition to 6 `int` or pointer arguments are passed via registers. Any additional arguments are passed on the stack. Stack manipulations are expensive relative to register manipulations, since they involve at least the L1 cache if not more. So frequently used (and especially short) functions should avoid this overhead. Structures are "broken" into their component types, and each component is treated as a separate argument; e.g., a `struct` with 4 `int` and 10 `double` would see all 4 integers passed in registers, the first 8 members passed in registers, and the last two on the stack. However, it is generally preferable to pass a pointer to a `struct` instead of passing the entire `struct` by value. The full specification of the calling convention is given in[50]; [25, 38] give additional discussion.

- Avoid excessive `malloc`,`free` calls: When possible, memory should be preallocated. Dynamic allocation can be slow, particularly when multiple small blocks are allocated and then released in a different order. This creates many small holes in the memory, fragmenting it just as a hard drive can become fragmented. The small holes reduce `malloc` performance. If memory is allocated in some order and then released in the reverse order (i.e., the heap operates like the stack), then the overhead is minimal. Nonetheless, the overhead unnecessary if preallocation is possible.

- Polymorphism: `C++` `virtual` functions (polymorphism) are among the most expensive parts of object-oriented programming. These calls incur extra run-time costs due extra branching used to look up which member function to call, based on the data types involved. When the type is obvious, the compiler may bypass the extra overhead, but this mechanism is unreliable[28]. Avoiding making frequently called functions `virtual` allows the benefits of object-oriented programming without most of the expense. Outside of polymorphism, most `C++` code will run as quickly as equivalent `C` code. The

[2]The following discussion follows the convention defined by the x86-64 ABI[50], which is used in Linux. 64 bit Windows has its own, different rules.

[3]These are passed in `XMM0` through `XMM7`; in particular, the `long double` type is passed on the stack.

object-oriented framework is readily applicable to finite element projects; e.g., consider classes for element types, governing equations (fluxes), basis types, etc.

- Optimal array indexing: Consider a block of code that does some processing on a $m \times n$ matrix A; a sample implementation follows:

```
double * restrict A = malloc(...);
//initialize A
for(i=0; i<n; i++){
  for(j=0; j<m; j++){
    temp = A[i*m + j]; //or temp = *(A + i*m + j);
    //do computations
  }
}
```

  Note that when setting temp, the compiler will generate code that looks more like: `temp = *(A + i*m + j);`. In assembly, this corresponds to moving the data at address `A+(i*m+j)*sizeof(double)` to the register holding `temp`. More specifically, the CPU will have to allocate registers and spend time performing the arithmetic operations to resolve the address first. Then the register holding the resulting address and the register holding `temp` are used with `MOV` to physically copy the data. So if the loop variable is j, accessing `x=A[j+p]` involves (integer) arithmetic operations followed by a `MOV`. The integer arithmetic operations have to calculate $j + p$ and multiply it by `sizeof(*A)` to compute the offset. Simple integer operations break into one μop (See Section C.4) and `MOV` breaks into one μop.

  There is also another assembly command, `LEA` (short for Load Effective Address), that can resolve the destination address using only one instruction (and one μop). In trade for its efficiency, `LEA` has some limitations. In particular, the most complex expression it can compute is: $A+i*size+const$, where A is a pointer (a memory address), i is an integer, $size$ is 2,4, or 8 (i.e., `sizeof(*A)`) and known at compile-time, and $const$ is an integer known at compile-time. So `x=A[j]` only requires `LEA` to calculate the address, and then MOV to copy data, as before. However, A[j+p] cannot be resolved with LEA. The previous code snippet can be rewritten to utilize LEA:

```
double * A = malloc(...);
//initialize A
double * restrict Aptr = A; //valid as long as A is not used
for(i=0; i<n; i++){
  for(j=0; j<m; j++){
    temp = Aptr[j];
    //do computations
  }
  Aptr += m;
}
```

As a result, the amount of overhead from address calculation has decreased significantly. Rewriting the matrix-vector product example in Section 3.5 to use LEA improves performance by as much as 20%.

The rewritten loop above has several other advantages. The counter i is not used and j is only used for simple addressing, so these can be register-allocated. Additionally, modern CPUs execute very short loops (less than 64bits of instructions) quickly, so removing the instruction overhead from address calculations helps the size low. Lastly, loops involving simple statements like `temp = Aptr[j];` are easier for the compiler to "understand," leading to better chances for automatic vectorization, unrolling, reordering, etc.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial, 1995.

[2] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] Hassan F. Al-sukhni, Brian C. Grayson, Smittle Holt, James C., Snyder Matt B., and Michael D. A method and apparatus for branch prediction using a second level branch prediction table, 2006.

[4] Saman P. Amarasinghe and Charles E. Leiserson. 6.172 performance engineering of software systems. Lecture Notes, 2009.

[5] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2010.

[6] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA Journal of Numreical Analysis*, 14:563–581, 1994.

[7] A. H. Baker, J. M. Dennis, and E. R. Jessup. An efficient block variant of gmres. *SIAM Journal on Scientific Computing*, 27, 2006.

[8] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted gmres. *SIAM J. Matrix Anal. Appl.*, 26:962–984, April 2005.

[9] A.H. Baker, E.R. Jessup, and Tz.V. Kolev. A simple strategy for varying the restart parameter in gmres(m). *Journal of Computational and Applied Mathematics*, 230(2):751 – 761, 2009.

[10] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. Technical Report UCB/EECS-2009-62, UC Berkeley, May 2010.

[11] Garrett E. Barter and Dave L. Darmofal. Shock capturing with pde-based artificial viscosity for dgfem: Part i, formulation. *Journal of Computational Physics*, 229(5):1810–1827, 2010.

[12] F. Bassi, A. Crivellini, S. Rebay, and M. Savini. Discontinuous Galerkin solution of the Reynolds averaged Navier-Stokes and k-ω turbulence model equations. *CF*, 34:507–540, May-June 2005.

[13] F. Bassi and S. Rebay. GMRES discontinuous Galerkin solution of the compressible Navier-Stokes equations. In Karniadakis Cockburn and Shu, editors, *Discontinuous Galerkin Methods: Theory, Computation and Applications*, pages 197–208. Springer, Berlin, 2000.

[14] F. Bassi and S. Rebay. Numerical evaluation of two discontinuous Galerkin methods for the compressible Navier-Stokes equations. *Int. J. Numer. Meth. Fluids*, 40:197–207, 2002.

[15] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[16] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[17] Laslo T. Diosady. A linear multigrid preconditioner for the solution of the Navier-Stokes equations using a discontinuous Galerkin discretization. Masters thesis, Massachusetts Institute of Technology, Computation, Design, and Optimization Program, May 2007.

[18] L.T. Diosady and D.L. Darmofal. Discontinuous Galerkin solutions of the Navier-Stokes equations using linear multigrid preconditioning. AIAA Paper 2007-3942, 2007.

[19] C.C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rde, and C. Wei. Fixed and adaptive cache aware algorithms for multigrid methods. In E. Dick, K. Riemslagh, and J. Vierendeels, editors, *Multigrids Methods VI*, volume 14, pages 87–93, Berlin, Germany, June 2000. Springerlink.

[20] Ulrich Drepper. What Every Programmer Should Know About Memory, 2007.

[21] Howard C. Elman and Dennis K.-Y. Lee. Use of linear algebra kernels to build an efficient finite element solver. *Parallel Comput.*, 21:161–173, January 1995.

[22] Jay Fenlason. GNU gprof Software, 2009.

[23] Krzysztof J. Fidkowski. *A Simplex Cut-Cell Adaptive Method for High-Order Discretizations of the Compressible Navier-Stokes Equations*. PhD thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2007.

[24] Krzysztof J. Fidkowski, Todd A. Oliver, James Lu, and David L. Darmofal. p-multigrid solution of high-order discontiunous Galerkin discretizations of the compressible Navier-Stokes equations. *Journal of Computational Physics*, 207(1):92–113, 2005.

[25] Agner Fog. Calling conventions for different C++ compilers and operating systems, 2010.

[26] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2010.

[27] Agner Fog. The Microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers, 2010.

[28] Agner Fog. Optimizing Software in C++: An optimization guide for windows, linux and mac platforms, 2010.

[29] Agner Fog. Optimizing Subroutines in Assembly Language: An optimization guide for x86 platforms, 2010.

[30] GNU. *GNU gprof*, 2010.

[31] Marcia A. Gomes-Ruggiero, Vera L. Rocha Lopes, and Julia V. Toledo-Benavides. A safeguard approach to detect stagnation of GMRES(m) with applications in Newton-Krylov methods. *Computational & Applied Mathematics*, 27:175 – 199, 00 2008.

[32] Mel Gorman. Huge pages, 2010.

[33] Jim Handy. *The Cache Memory Book*. Academic Press Professional, Inc., San Diego, CA, USA, 1998.

[34] M. Hoemmen. *Communication-Avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, April 2010. UCB EECS Technical Report EECS-2010-37.

[35] Eun-Jin Im. Optimizing the performance of sparse matrix-vector multiplication. Technical Report UCB/CSD-00-1104, University of California, Berkeley, 2000.

[36] Intel. Intel®64 and IA-32 Architectures Optimization Reference Manual, 2009.

[37] Intel. Intel®VTune Performance Analyzer 9.1, 2009.

[38] Intel. *Intel®C++ Compiler 11.1 User and Reference Guides*, 2010.

[39] Intel. Intel®IA-64 Architecture Software Developer's Manual - Volume 3: System Programming Guide, 2010.

[40] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64:1017–1026, September 2004.

[41] Hong Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[42] Maynard Johnson, Suravee Suthikulpanit, Richard Purdie, Daniel Hansel, and Robert et al. Richter. OProfile Software, 2009.

[43] Robert C. Kirby. Optimizing fiat with level 3 blas. *ACM Trans. Math. Softw.*, 32:223–235, June 2006.

[44] A. Klckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863 – 7882, 2009.

[45] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[46] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. Technical report, University of Minnesota, 2010.

[47] Adam G. Litke. "Turning the Page" on Hugetlb Interfaces. In *Proceedings of the Linux Symposium*, pages 277–284. Ottowa Linux Symposium, 2007.

[48] Adam G. Litke, Eric Munson, and Nishanth Aravamudan. libhugetlbfs, 2010.

[49] James Lu. *An a Posteriori Error Control Framework for Adaptive Precision Optimization Using Discontinuous Galerkin Finite Element Method*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2005.

[50] Michael Matz, Jan Hubicka, Andrea Jaeger, and Mark Mitchell. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, 2010.

[51] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *ISPASS '08: Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 95–104, Washington, DC, USA, 2008. IEEE Computer Society.

[52] Kent Milfeld, Kazushige Goto, Avi Purkayastha, Chona Guiang, and Karl Schulz. Effective Use of Multi-Core Commodity Systems in HPC. In *The 11th LCI International Conference on High-Performance Clustered Computing*. Pittsburgh Supercomputing Center, 2007.

[53] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12, New York, NY, USA, 2009. ACM.

[54] Malte Neumann, Sunil R. Tiyyagura, Wolfgang A. Wall, and Ekkehard. Ramm. Efficiency aspects for advanced finite element formulations. In E. Ramm, W. A. Wall, K. U. Bletzinger, and M. Bischoff, editors, *Proceedings of the 5th International Conference on Computation of Shell and Spatial Structures*, June 2005.

[55] Rajesh Nishtala, Richard Vuduc, James Demmel, and Katherine Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18:297–311, 2007. 10.1007/s00200-007-0038-9.

[56] Todd A. Oliver. *A Higher-Order, Adaptive, Discontinuous Galerkin Finite Element Method for the Reynolds-averaged Navier-Stokes Equations*. PhD thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2008.

[57] Todd A. Oliver and David L. Darmofal. Analysis of dual consistency for discontinuous galerkin discretizations of source terms. *SIAM Journal on Numerical Analysis*, 47(5):3507–3525, 2009.

[58] Thomas C Oppe. *Tuning Application Performance on the Cray XT Without Modifying the Code*, pages 37–38. Department of Defense Super-computing Resource Centers, 2010.

[59] Roger P. Pawlowski, John N. Shadid, Joseph P. Simonis, and Homer F. Walker. Globalization Techniques for Newton-Krylov Methods and Applications to the Fully Coupled Solution of the Navier-Stokes Equations. *SIAM Rev.*, 48:700–721, November 2006.

[60] Per-Olof Persson and Jaime Peraire. Newton-GMRES preconditioning for discontinuous Galerkin discretizations of the Navier-Stokes equations. *SIAM Journal on Scientific Computing*, 30(6):2709–2722, 2008.

[61] Youcef Saad and Martin H Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, July 1986.

[62] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2 edition, April 2003.

[63] Joseph P. Simonis. A Numerical Study of Globalizations of Newton-GMRES Methods. Masters thesis, Worcester Polytechnic Institute, Department of Applied Mathematics, May 2003.

[64] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[65] Pavel Solin, Karel Segeth, and Ivo Dolezel. *Higher-Order Finite Element Methods*. Chapman and Hall/CRC, New York, NY, 2003.

[66] Richard M. et al. Stallman. *GCC 4.5.1 Manual*, 2010.

[67] Gilbert Strang and George J. Fix. *An Analysis of the Finite Element Method*. Wellesley-Cambridge Press, Wellesly, MA, 1988.

[68] Brian Waldecker. Quad Core Processor Overview, 2007.

[69] Li Wang and Dimitri J. Mavriplis. Adjoing-based *h-p* adaptive discontinuous Galerkin methods for the compressible Euler equations. AIAA Paper 2009-952, 2009.

[70] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving sparse linear systems on nvidia tesla gpus. In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*, pages 864–873. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-01970-8_87.

[71] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.

[72] Tse-Yu Yeh, P. Sharangpani, Harshvardhan, and Intel®. Prefetching across a page boundary, 2000.