PARALLEL INVERSION OF LARGE MATRICES ON GRAPHICS HARDWARE

by

JASON E. TREADWELL

B.S., University of Colorado, 2008

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Computer Science Program

2016

This thesis for the Master of Science degree by

Jason E. Treadwell

has been approved for the

Department of Computer Science

by

Gita Alaghband, Chair

Tom Altman, Advisor

Farnoush Banaei-Kashani

Boris Stilman

December 17, 2016

Treadwell, Jason E. (M.S., Computer Science)

Parallel Inversion of Large Matrices on Graphics Hardware

Thesis directed by Professor Tom Altman

ABSTRACT

Matrix inversion is an important operation in domains ranging from graphics and real-time applications to nonlinear optimization and regression analysis. Traditional methods of matrix inversion have cubic runtimes that scale poorly as the matrix dimension grows.

A method by M. Altman produces a sequence of cubically and monotonically-convergent approximations of the matrix inverse using a few matrix multiplications and additions each iteration. While these operations themselves are hardly trivial, the homogeneous and independent nature of their constituent computations makes them prime candidates for parallel acceleration using recent advances in GPU (*graphics processing unit*) hardware.

Using a consumer GPU, we show that matrices can be quickly inverted with high fidelity using Altman's method. We exploit the error tolerance of the method to gain further speed by using mixed-precision computations and compare the results to those from double-precision and CPU computations.

The form and content of this abstract are approved. I recommend its publication.

Approved: Tom Altman

TABLE OF CONTENTS

CHAPTER

I. INTRODUCTION 1

1.1 Matrices and Their Inverses 1

1.2 Altman's Iterative Matrix Inversion Method 3

1.3 Computational Aspects of Altman's Method 6

II. COMPUTER LINEAR ALGEBRA 7

2.1 BLAS and OpenBLAS 7

2.2 Parallel Computation on the GPU with CUDA 8

2.3 CUDA Linear Algebra with cuBLAS 12

III. IMPLEMENTATION OF ALTMAN'S METHOD 14

3.1 Development Environment and General Architecture 14

3.2 The Initial Inverse Approximation 15

3.3 Computation of the Sequence 16

3.4 Recovery from Divergence 18

IV. EXPERIMENTAL RESULTS 20

4.1 Testing Environment and Overview of Experiments 20

4.2 Diagonally-Dominant Random Matrices 22

4.3 Random Matrices 25

4.4 Real-World Large Matrices 27

4.5 Hilbert Matrices 29

4.6 Comparison of the Orders of Convergence 30

V. CONCLUSION 33

5.1 Implications of Results 33

5.2 Further Research 33

REFERENCES 35

APPENDIX

A. ACMI Inverter Usage 38

B. Sample ACMI Inversion Runs 39

C. Selected Source Code 43

FIGURES

FIGURE

2.1 CPU/GPU Architectural Differences 8

2.2 Heterogeneous Programming Model 10

2.3 CUDA Grid/Block/Thread Model 11

4.1 GeForce GTX 1080 Streaming Multiprocessor 21

4.2 Inversion Runtime for $n \times n$ Diagonally-Dominant Random Matrices . . 23

4.3 Inversion Runtime for $n \times n$ Random Matrices 25

4.4 Inversion Runtime for $n \times n$ Boolean Matrices 26

4.5 Inversion Runtime for Large, Sparse Matrices 28

4.6 Inversion Error for $n \times n$ Hilbert Matrices 30

4.7 Order-p Inversion Runtime for Diagonally-Dominant Random Matrices . 31

4.8 Order-p Inversion Runtime for Random Matrices 32

4.9 Total Order-p Inversion Iterations for Random Matrices 32

TABLES

TABLE

2.1 Differences between a CPU and a GPU 9

3.1 BLAS Matrix/Vector Routines Used 16

3.2 Implemented Custom Matrix Routines 17

4.1 GeForce GTX 1080 Specifications 20

4.2 Sparse Matrix Collection Specimens 27

CHAPTER I

INTRODUCTION

1.1 Matrices and Their Inverses

An $m \times n$ *matrix* is a two-dimensional array of m rows and n columns of real-valued or complex *elements*. Matrices are used to model systems of linear equations, graphs, raster graphics and manifold other computational phenomena. Consider a system of three equations over three scalar variables x_1, x_2 and x_3:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \,,$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \,,$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \,.$$

This linear system can be represented as a matrix of the coefficients times a *vector* (or $n \times 1$ matrix) of the variables equal to the vector of the right-hand side:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} .$$

The same system is represented concisely and algebraically as

$$Ax = b \,,$$

where capital and small symbols represent matrices and vectors, respectively.

Addition, multiplication and other operations may be performed on matrices. The element c_{ij} in the ith row and jth column of the matrix sum $C = A + B$ is simply the sum of the elements from the corresponding positions: $a_{ij} + b_{ij}$. Each element c_{ij} of the matrix product $C = AB$ is the dot product of the ith row of A and

the jth column of B. An $m \times p$ matrix is multiplied on the right by a $p \times n$ matrix to yield an $m \times n$ result; each row from the left must correspond with one column on the right. Multiplication of the matrix (a_{ij}) by a scalar c is commutative and yields (ca_{ij}).

The algebra of matrices admits the *inverse* of a *square* (i.e., $n \times n$) matrix[1]. Analogous to scalar algebra, where the inverse of a scalar x satisfies $x^{-1}x = 1$, a matrix multiplied by its inverse produces the $n \times n$ *identity* matrix I of ones on the main diagonal and zeros elsewhere.

$$A^{-1}A = AA^{-1} = I = (i_{jk}) \qquad i_{jk} = \begin{cases} 1 & \text{if } j = k\,, \\ 0 & \text{otherwise}\,. \end{cases} \quad ,$$

$$AI = IA = A\,.$$

Matrix multiplication is otherwise not generally commutative, yet it is associative and distributes over addition (which itself is both commutative and associative). We can apply the inverse of the matrix A to both sides of the previous system to simultaneously solve all three equations.

$$A^{-1}Ax = A^{-1}b \quad \longrightarrow \quad x = A^{-1}b\,.$$

A *singular* matrix has no inverse. Every *invertible* matrix is square, has a unique inverse, has a nonzero determinant, consists exclusively of linearly-independent columns and satisfies a host of additional conditions per the *invertible matrix theorem*. Every positive-definite matrix is invertible [9]. Given a nonsingular matrix, the process of *inversion* to compute its inverse is hardly trivial.

While inverting a matrix to solve a linear system is neither computationally efficient [6] nor numerically-stable [9] (i.e., prone to amplified round-off error), an approximate inverse can be quickly computed with sufficient accuracy to satisfy certain

[1]The *Moore-Penrose pseudoinverse* generalization of the matrix inverse applies to non-square matrices [20], but was not investigated here.

real-time robotics applications [32]. Matrix inversion is also employed to perform regression analysis [20], nonlinear programming [5], Newton's method for nonlinear systems [6] as well as ray casting in computer graphics [30].

Gauss-Jordan elimination is a slow [29], but introductory method of inversion where we use Gaussian elimination to reduce the left side of the augmented matrix $[A\colon I]$ to I, producing A^{-1} on the right side [6]. The LU and LUP triangular decompositions of the matrix A, which are numerically-stable methods of solving linear systems, can be used to solve $Ax_i = e_i$ for each unit vector e_i to produce the columns of A^{-1} more quickly [9]. Cholesky decomposition of Hermitian, positive-definite matrices is much more efficient than LU decomposition and can similarly be used to invert a matrix [29]. Each of these "traditional" inversion methods has $\Theta(n^3)$ runtime complexity.

Winograd proved that matrix multiplication is no harder than inversion while Aho, Hopcroft, and Ullman proved the converse [9]; the lower and upper runtime bounds for matrix inversion thus have quadratic and cubic order, respectively.

1.2 Altman's Iterative Matrix Inversion Method

M. Altman [3] defined a family of sequences (R_n) of monotonically-convergent approximations to the inverse A^{-1}:

$$R_{n+1} = R_n(I + T_n + T_n^2 + \cdots + T_n^{p-1}) \, ,$$

$$T_n = I - AR_n \, ,$$

$$\lim_{n \to \infty} R_n = A^{-1} \, .$$

Each sequence of this family has *degree* or *order* p, where $p \geq 2$, each iteration of the sequence requires p matrix multiplications, and the sequence converges towards A^{-1} with degree p. The following are the sequences of degree 2, 3 and 4 that converge quadratically, cubically and quartically towards A^{-1}, respectively (noting that we

compute AR_n only once per iteration):

$$\begin{aligned} \textit{quadratic} \quad R_{n+1} &= R_n(2I - AR_n)\,, \\ \textit{cubic} \quad R_{n+1} &= R_n(3I - 3AR_n + (AR_n)^2)\,, \\ \textit{quartic} \quad R_{n+1} &= R_n(4I - AR_n(6I - 4AR_n + (AR_n)^2))\,. \end{aligned}$$

Given an appropriate starting approximation R_0, each sequence forms a method of quickly computing the inverse of A in k iterations within some acceptable *error measure* $\|I - AR_k\|$ (a Frobenius norm) that approaches zero as R_k approaches A^{-1} (since $AA^{-1} = I$). When such an error bound exists and is attainable, the runtime of this method can drastically outperform the traditional inversion methods described in the previous section.

We wish to know which of these sequences provides the fastest method of inversion. Since matrix multiplication dominates the computational runtime of an iteration of any sequence in the family, the closer we get to the true inverse per multiplication, the better. Consider two such sequences: one of degree p (which requires p multiplications per iteration) and one of degree q such that, after m iterations of the first sequence and n iterations of the second, ν matrix multiplications have been performed over each (i.e., $mp = nq = \nu$). The *better* or faster sequence is the one which, after ν multiplications, results in the approximate inverse with the lower error measure. Suppose the sequence of degree p is the better method. Altman showed that, since the error measures of these iterations are a function of the initial inverse approximation's error measure raised to p^m or q^n (respectively), we have

$$\begin{aligned} p^m &> q^n\,, \\ p^{nq/p} &> q^{mp/q}\,, \\ p^{1/p} &> q^{1/q}\,. \end{aligned}$$

The runtime is thus minimized by using the sequence of degree $p = \lceil e \rceil = 3$, which

converges cubically with successive iterations to the inverse of A:

$$R_{n+1} = R_n(3I - 3AR_n + (AR_n)^2)\,.$$

The choice of the initial approximation R_0 used to seed the inversion is crucial. The convergence of the method is guaranteed provided that $\|I - AR_0\| < 1$. For any invertible matrix, we may choose $R_0 = \alpha A^*$, where A^* is the conjugate transpose of A. We shall only consider real-valued matrices, so we effectively have $R_0 = \alpha A^T$. The scalar multiple α must lie within a finite interval and leads to the fastest convergence at the *critical value* $\alpha = 2/(M^2 + m^2)$, where m and M and the minimum and maximum eigenvalues of A, respectively.

Unfortunately, computing eigenvalues[2] is generally about as hard as matrix inversion itself [28]. Fortunately, an acceptable choice of α can be quickly drawn from the interval

$$0 \leq \alpha \leq \frac{1}{\|A\|^2}\,,$$

where the upper bound produces the fastest-converging choice of R_0.

Should A be self-adjoint and positive definite, we can perform the much faster computation $R_0 = \alpha I$, where the fastest convergence occurs with $\alpha = 2/(M + m)$. An easily-computed alternative is

$$\alpha = \frac{1}{\|A\|}\,.$$

The iterative improvement of Altman's method makes it "self-correcting" (in a sense) and grants it some tolerance of floating-point round-off error and less-than-ideal choices of R_0. In contrast to other inversion methods, Altman's can provide useful intermediate results when its runtime is constrained [32].

[2]One typical method of which is the *QR algorithm* [28].

1.3 Computational Aspects of Altman's Method

Since the runtime of an Altman iteration is dominated by its matrix multiplications, the method's complexity depends upon the multiplication algorithm used. The square matrix multiplication problem has $\Omega(n^2)$ and $O(n^3)$ runtime; the theoretical lower bound simply reflects the size of the output.

Specifically, naïve matrix multiplication performed by computing all n^2 dot products runs in $\Theta(n^3)$ time using constant space. Strassen's landmark 1969 divide-and-conquer algorithm runs in $\Theta(n^{\lg 7})$ time and can be run in parallel almost to the same extent as the naïve algorithm, yet is less numerically stable, consumes more space and doesn't exploit sparse matrices well. Furthermore, its runtime has a larger constant factor, outperforming the naïve algorithm only at a highly-variable *crossover point* as high as $n = 2150$ [9]. The Coppersmith-Winograd algorithm of 1987 currently has the best known upper bound of $O(n^{2.376})$, yet has a crossover point so high as to be "wildly impractical for any conceivable applications" [8].

For these reasons, linear algebra libraries tend to implement optimized versions of the naïve multiplication algorithm [19]. Using that algorithm, the computation of a single Altman sequence iteration has the same $\Theta(n^3)$ runtime as the traditional matrix inversion methods. However, the lower constant factor of the former combined with a small number of iterations (due to the fast convergence) can make for a shorter total runtime. Finally, and perhaps most importantly, each Altman iteration requires only matrix multiplication and addition to compute; these operations scale very well in parallel.

CHAPTER II

COMPUTER LINEAR ALGEBRA

2.1 BLAS and OpenBLAS

The *Basic Linear Algebra Subprograms*, or BLAS, are a venerable specification of routines that perform elementary matrix operations. BLAS includes addition, multiplication, rank update, norm and solution routines that are divided among three *levels* [1]:

1. Scalar, vector and vector-vector operations.

2. Matrix-vector operations.

3. Matrix-matrix operations.

BLAS was originally developed in Fortran and traditionally employs *column-major* ordering—a matrix A with m rows of n columns is *strided* across columns and is laid out in sequential memory as $\{a_{11}, a_{21}, \cdots, a_{m1}, a_{12}, a_{22}, \cdots, a_{mn}\}$. BLAS provides routines for working with both single and double-precision floating-point matrix elements; the latter precision requires twice as much storage, but is needed when numerical instability renders the former inadequate. In our implementation of Altman's inversion method, we used the level 3 `GEMM` routines to multiply matrices; we treated matrices as $1 \times n^2$ vectors to norm them and to add them together using level 1 routines.

There are many BLAS implementations. For running computations purely in software, we employed the *OpenBLAS* library, an up-to-date fork of the highly-optimized and sophisticated GotoBLAS library [23]. Specifically, we used OpenBLAS compiled to divide its computations across all available cores using POSIX threads on Linux with bindings for the C programming language. Matrix multiplication in OpenBLAS

is highly-performant on recent Intel processors, outperforming Intel's own *MKL* implementation of BLAS [35].

In addition, we implemented custom routines for handling tasks not easily supported by BLAS, such as adding a multiple of the identity matrix to another matrix, computing the trace of a matrix (i.e., the sum of its diagonal elements) and converting a matrix from single to double precision. These routines were dominated in runtime by the algebraic matrix operations and were thus computed serially for simplicity.

2.2 Parallel Computation on the GPU with CUDA

For parallel computations accelerated on graphics hardware, we used Nvidia's proprietary *CUDA* platform. Introduced in 2007 and the subject of much research and commercial application, CUDA is a *General Purpose Graphics Processing Unit* (or *GPGPU*) toolkit for executing parallel code on a system's graphics hardware for non-graphical tasks. CUDA implements a model of *heterogeneous computing* where a general-purpose *host* CPU coordinates suitable tasks across one or more heavily-specialized GPU *devices*.

CUDA was foreshadowed in the mid-1990s when GPU transistor counts began to exceed those of CPUs. Heterogeneous GPGPU computing exploits the massively-parallel nature of GPU hardware, which was originally developed to rapidly process large numbers of independent pixels for computer games and other graphical applications [34].

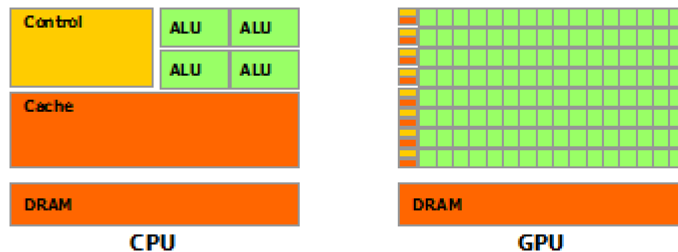

Figure 2.1: *CPU/GPU Architectural Differences* [24]

Table 2.1: *Differences between a CPU and a GPU*

CPU	GPU
Contains several powerful cores.	Contains *many* lightweight cores.
Cache dominates the die.	Logic dominates the die.
High core clock rates.	High memory bandwidth.
Meant for complicated sequential logic.	Designed for *instruction-level parallelism.*
Predicts and caches code paths.	Hides latency through parallelism.
Fully-independent cores.	Cores execute the same instruction.
Expensive thread context switches.	Very cheap thread context switches.

A typical modern CPU contains 2 to 4 cores whereas a recent Nvidia GeForce GTX 1080 card has 2560 cores in total. GPU hardware implements a *single instruction, multiple thread* (*SIMT*) architecture that differs from its *multiple data* (SIMD) counterpart in that an operand vector's elements are processed by independent threads, whose behavior might not be uniform [7]. Computations on the GPU "cover up" driver and bus communication latency as well as lower core clock rates by efficiently executing many threads in parallel. CUDA naturally fits into the current trend towards parallel computing and can accelerate suitable workloads by a factor of 5 to 400; this acceleration comes at the cost of the increased complexity inherent to concurrent computation [34].

A heterogeneous application using the CUDA runtime consists of a standard CPU program binary bundled with GPU *compute kernel* functions written in C++ and compiled with Nvidia's `nvcc` compiler. These kernels are loaded into GPU memory during initialization of the runtime. During execution, the program interacts with the GPU over the PCI Express bus through the device driver by transferring data from host memory to device memory, *launching* kernels over data in device memory and reading results back to host memory. Kernel launches are queued in order of

invocation and dispatched by the driver[1], running asynchronously with respect to the CPU program, which synchronizes with the launched kernels when accessing device memory or upon explicit command.

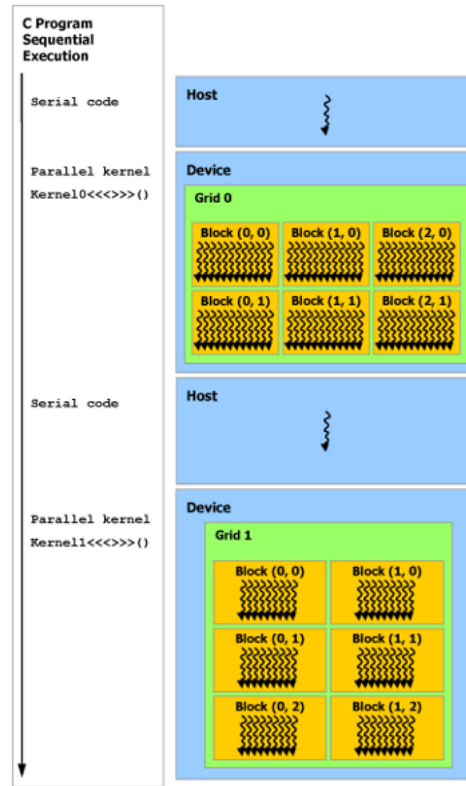

Figure 2.2: *Heterogeneous Programming Model* [24]

The GPU hardware consists of a pool of *global* device memory and some number of *streaming multiprocessors* (or *SMs*). Each SM consists of cores that execute integer and floating-point operations in parallel, smaller numbers of double-precision and special function (e.g., transcendental) units, caches, thousands of registers to be shared among all resident threads, a small pool of *shared* memory for fast inter-thread communication as well as *warp* schedulers.

Kernels are launched with a specified number of *blocks* and threads per block. Each block is scheduled separately by the driver onto the next-available SM where

[1] The programmer may establish one or more *streams* through which kernels can be run concurrently on the GPU.

its threads are divided into warps consisting of 32 threads each. A warp's threads run on distinct execution cores that all simultaneously execute the same instruction dispatched by the warp scheduler [7]. The set of blocks comprising a kernel execution is called a *grid*. Each thread in the grid has a distinct ID with which it can determine its portion of the total parallel computation.

Global device memory access is a major bottleneck for kernels, taking hundreds of clock cycles per request. Kernels can be optimized to internally use shared memory or to *coalesce* memory access; the latter occurs when a block's threads simultaneously operate over contiguous and adjacent runs of global memory, minimizing the number of memory reads and writes performed before each instruction by the GPU, which reads from and writes to memory in 32-byte or wider contiguous chunks [7].

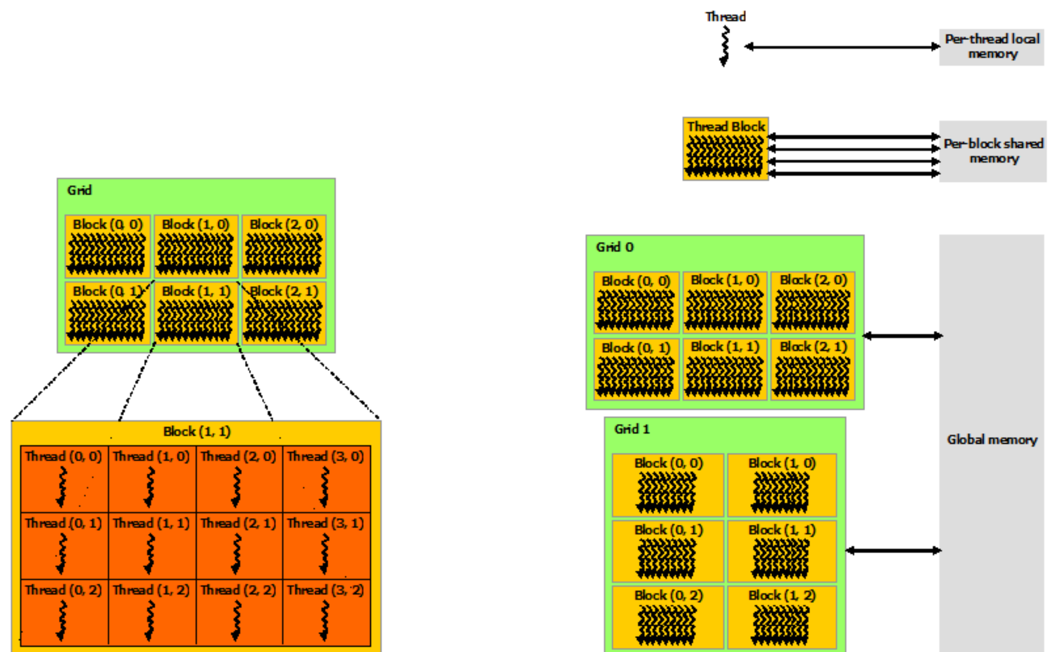

Figure 2.3: *CUDA Grid/Block/Thread Model* [24]

In many ways, GPUs outperform CPUs at individual floating-point operations. Single-precision floating-point operations form the "workhorse" of GPU computations and are thus heavily optimized [34]. Double-precision operations incur a performance penalty as they require twice the memory and registers as well as special units on the

streaming multiprocessor. However, for iterative methods and other computations that are sensitive to numerical instability, double-precision arithmetic proves to be invaluable.

On Nvidia's Tesla GPUs (meant for servers and for scientific computing), these double-precision units are outnumbered 3:1 and 2:1 on the older Kepler and the latest Pascal architectures, respectively; this ratio reflects the relative maximum throughputs of these precisions [7] [26]. Tesla GPUs aren't intended for consumer desktop computing and are extremely expensive, whereas the GeForce line of GPUs, which are intended for computer gaming, are far more affordable and common.

Nvidia's own white papers appear to be mute on the topic, but sources on the Internet variously claim that GeForce GPUs are either fabricated with fewer double-precision units or are outright "crippled" by having these units simply disabled despite being in parity with their Tesla counterparts [33] [15]. Whatever the cause, the result is that the ratio of single-precision to available double-precision units on a high-end Pascal GeForce GPU is 32:1, resulting in a dramatic loss of performance when computing with double precision on the consumer-grade units.

2.3 CUDA Linear Algebra with cuBLAS

Our GPU implementation of Altman's method runs on CUDA primarily through Nvidia's *cuBLAS* library. This BLAS implementation is parallel and heavily optimized for working with dense matrices, outperforming optimized CPU BLAS implementations by a factor of least 15 [7]. CUDA also comes packaged with the cuSPARSE library optimized for computations over sparse matrices, but this isn't used for inverting such matrices as their inverses and intermediate iterations tend to be dense [11].

The cuBLAS library includes "BLAS-like" extensions such as the `GEAM` routines, which are used to more conveniently perform matrix addition and transposition.

Handwritten kernels were employed for the same tasks that weren't well-suited to OpenBLAS. Since these operations are easily divided into independent divide-and-conquer problems and are thus *embarrassingly parallel* [16], their kernels can't exploit inter-thread communication and thus only achieve optimization by coalesced memory access. A handwritten kernel was tested for computing the Frobenius norm of a matrix less some multiple of the identity, but it was still outperformed by the much-simpler level 1 `SNRM2` routine offered by cuBLAS. A custom kernel is also used to convert matrices from single to double precision; while the *copy* function from CUDA's *Thrust* implementation of the C++ STL can accomplish this with identical performance, including Thrust among the linked libraries substantially increased compilation time and inflated the size of the binary.

CHAPTER III

IMPLEMENTATION OF ALTMAN'S METHOD

3.1 Development Environment and General Architecture

Our implementation of Altman's method is called *ACMI*[1]. ACMI was developed on Linux (specifically openSUSE 42.1 and Debian 8.5) using an Nvidia GeForce GTX 1080 graphics card—a high-end, consumer-grade device (see Section 4.1 for additional details regarding computer hardware). ACMI was written primarily in C and compiled with GCC; device kernels and related code were written in C++ and compiled with Nvidia's `nvcc` compiler from the CUDA 8.0 SDK. We have not explored the simultaneous usage of multiple graphics cards over SLI for this application.

For reasons (see Section 2.2) that are as economical as they are practical, ACMI was developed for relatively-inexpensive consumer hardware, hence the GeForce GPU used. Compared to their Tesla counterparts, these units perform double-precision arithmetic very slowly. ACMI was thus written to use single-precision arithmetic by default for as long as it suffices in order to speed up computation, resorting to double precision per heuristics described in this chapter. The size and numerical stability of the source matrix determine how much ACMI can accomplish in single precision. However, ACMI may also be configured to run all computations in double precision or on the CPU from the start.

ACMI can accept a Matrix Market[2] file as input or generate a random source matrix with various configurable characteristics, including size, diagonal dominance and element domains. Since software pseudorandom number generators can be of dubious quality [21] and even respectable ones can fail a statistical matrix rank test [22], ACMI can optionally use a very high-quality [18] hardware random number genera-

[1] ACMI's a Convergent Matrix Inverter.

[2] ACMI uses source code provided by NIST for reading and writing matrices encoded in the Matrix Market format [27].

tor available on Ivy Bridge and newer Intel processors. The inverse computed by the program may be written to disk in the Matrix Market format. ACMI's command-line interface also lets the user configure the order of convergence, initial floating-point precision, maximum inversion error, whether to use the GPU and other inversion parameters (see Appendix A for a complete listing).

During execution, ACMI loads or generates the source matrix, initializes the GPU and runs the centerpiece `altmanInvert()` routine over the matrix. This routine allocates work matrices, computes an initial inverse approximation R_0 and computes iterations $\{R_i\}$ of the Altman sequence until an approximate inverse is generated with an error measure $\|I - R_i\|$ under the given limit (10^{-5} by default); a lower error limit might necessitate additional sequence iterations. If the sequence diverges, the routine either restarts with a different initial approximation, promotes the work matrices to double precision or halts returning the previous inverse approximation, depending on the circumstances.

3.2 The Initial Inverse Approximation

As discussed in Section 1.2, determining the ideal initial inverse approximation of a matrix A requires knowledge of its eigenvalues and is, therefore, a computationally-intensive operation. Acceptable and quickly-computable substitutes are $R_0 = I/\|A\|$ when A is positive definite and $R_0 = A^T/\|A\|^2$ for the general case. However, determining whether A is positive definite is related to the eigenvalue problem and is itself unacceptably expensive [6].

It turns out that the inverse approximation for positive-definite matrices works for many matrices outside that class. For these matrices as well as true positive-definite ones, this approximation yields much faster convergence than when using its general-case counterpart. Moreover, we can usually determine early in the inversion process whether the positive-definite approximation shall lead to divergence and never

produce a suitable inverse. ACMI's default operation is, therefore, to simply apply the positive-definite approximation first and switch to the general approximation in the event of early divergence.

Unexpectedly, both initial approximations usually fail to satisfy $\|I - AR_0\| < 1$ and do not meet Altman's requirement for guaranteed convergence. This frequently happens to be of little consequence. We could artificially satisfy the condition by lowering the α scalar used to produce R_0, but this dramatically slows the convergence of the sequence, in practice.

3.3 Computation of the Sequence

ACMI implements BLAS wrappers that invoke their respective cuBLAS or OpenBLAS routines of the appropriate precision when GPU computation is enabled or disabled, respectively. ACMI additionally implements handwritten CUDA kernels and host functions for performing custom computations.

Table 3.1: *BLAS Matrix/Vector Routines Used*

Routine	Function
`GEMM`(α, A, B, β, C)	$C \leftarrow \alpha AB + \beta C$
`GEAM`(α, A, β, B, C)	$C \leftarrow \alpha A + \beta B$
`SCAL`(α, x)	$x \leftarrow \alpha x$
`AXPY`(α, x, y)	$y \leftarrow y + \alpha x$
`NRM2`(A)	$\|A\|$

The `GEMM` routine performs matrix-matrix multiplication and is the main workhorse of the inversion process. `GEAM` is a "BLAS-like" extension to cuBLAS and simplifies matrix-matrix addition; we wrote a CPU implementation of this routine using `SCAL` and `AXPY` in OpenBLAS. `NRM2` computes a Frobenius norm and is used to compute the scalar coefficient of the initial inverse approximation. The handwritten `transpose`

Table 3.2: *Implemented Custom Matrix Routines*

Routine	Function
`transpose`(α, A, T)	$T \leftarrow A^T$
`trace`(A)	$\sum_{i=1}^{n} a_{ii}$
`addId`(α, A)	$A \leftarrow A + \alpha I$
`minusIdNrm2`(A)	$\|I - A\|$

and `trace` routines perform their namesakes and are used to compute the sequence's general-case initial inverse approximation R_0 as well as the optimized, positive-definite initial error measure $\|I - AR_0\|$, respectively. The `addId` routine was written to perform linear-time addition of a scalar multiple of the identity to a matrix; this routine helps us avoid having to allocate a large and slow identity matrix and is also used with `NRM2` to implement `minusIdNrm2` for computing each inversion iteration's error measure. An iteration of the cubically-convergent Altman sequence is thus computed by ACMI to produce the next approximate inverse R of the matrix A as follows:

```
1  gemm(1, mA, mR, 0, mAR);   // mAR <- AR
2  err = minusIdNrm2(mAR);    // compute error of the prev iteration
3  ...                        // based on error, stop or back up
4  gemm(1, mAR, mAR, 0, mX); // mX  <- (AR)^2
5  geam(-3, mAR, 1, mX, mX); // mX  <- -3AR + (AR)^2
6  addId(3, mX);              // mX  <- 3I - 3AR + (AR)^2
7  gemm(1, mR, mX, 0, mAR);   // mAR <- R(3I - 3AR + (AR)^2)
8  swap(&mR, &mAR);           // mR  <- next R, mAR <- prev R
9  swap(&mX, &mAR);           // mX  <- previous R
```

Floating-point operations are not generally associative with respect to round-off error; the order in which a computation is performed is important. In an effort to reduce accumulated error, the above sequence of matrix operations was borrowed from earlier work [32]. The previous iteration is saved in case ACMI decides to "back up" an iteration to reduce error before changing strategy or quitting. While both BLAS engines compute each individual operation in parallel, the sequence of operations is itself serially-interdependent and presents no opportunity for parallel execution of separate matrix operations (i.e., all CUDA operations are run through

a single stream).

In addition to the above, ACMI implements the quadratically and quartically-convergent Altman sequences as well; the latter requires a fifth work matrix (i.e., much more memory). Their relative performances shall be examined in Chapter 4.

3.4 Recovery from Divergence

As we shall see, many matrices, both large and small, cannot be accurately inverted using ACMI with single-precision arithmetic alone. However, the substantial performance penalty of using double precision on the GPU compels us to perform as much of the inversion as possible in single precision. We therefore need a mechanism for detecting when to abandon single precision.

The simplest method of accomplishing this is to compute iterations until one *diverges*. Since Altman's sequences converge monotonically to the inverse, an iteration whose error measure exceeds that of the previous indicates that floating-point round-off error has caught up with ACMI and halted the progress of the inversion. Since Altman's method is iteratively improving and tolerant of errors, ACMI simply promotes its work matrices to double precision after divergence occurs in order to overcome the round-off error and continue the inversion process.

However, our inverse approximation might have accumulated substantial "damage" by the time outright divergence has occurred. This damage can require additional (and expensive) double-precision iterations to correct; it might even derail the remainder of the inversion. We would like a reliable method of measuring numerical instability and detecting divergence ahead of time to elude this damage. To this end, we employed the sequence's *rate of convergence*.

A sequence $\{x_n\}$ that converges with order p towards zero has constant rate μ such that [13]

$$|x_n| \leq \varepsilon_n\,, \quad \lim_{n\to\infty} \frac{\varepsilon_{n+1}}{\varepsilon_n^p} = \mu\,, \quad \mu > 0\,.$$

Substituting our monotonically-decreasing error measure and using the cubically-convergent sequence, we have

$$\lim_{n\to\infty} \frac{\|I - AR_{n+1}\|}{\|I - AR_n\|^3} = \mu\,.$$

We don't know what the value of μ is for a given inversion, but we do know that μ exists, since this sequence does indeed converge cubically to the inverse. We therefore wouldn't expect the sequence of rate measurements $\{\mu_n\}$ to itself diverge unless we fall victim to numerical instability and, based on evidence, hypothesize that such a divergence would occur before that of $\{R_n\}$. ACMI measures this rate after each iteration and promotes its matrices after the rate exceeds some given threshold. The ideal value of this threshold varies; ACMI tolerates a limit of $\mu < 1$ by default, which was found to be reasonable for many matrices during initial testing. We shall explore the effectiveness of this strategy in Chapter 4.

If divergence occurs within the first few iterations, the initial inverse approximation is probably to blame; ACMI then computes the starting approximation for the general case and restarts the inversion. When divergence can't be blamed on the initial approximation and occurs while working in double precision, ACMI terminates the inversion and returns the previously-computed inverse approximation as a consolation prize.

It should be noted that, if the inverter is allowed to indefinitely continue producing inverse approximations after hitting the wall of numerical instability (but not when the matrix is singular or the starting approximation is inadequate, which lead to outright divergence), its iteratively-improving nature shall cause their error measures to hover around the lowest-achievable figure indefinitely.

CHAPTER IV

EXPERIMENTAL RESULTS

4.1 Testing Environment and Overview of Experiments

The experiments were run on a custom-built workstation sporting an Intel Xeon E3-1275 v5 processor from the Skylake generation. The CPU has 4 hyper-threaded cores running at 3.6 GHz with 256KB L2 cache per core and 8MB of shared L3 cache; the system memory is 2133 MHz ECC DDR4. The workstation's graphics card is a Pascal-generation Nvidia GeForce GTX 1080, which was the highest-end and most-recent consumer GPU offered by Nvidia at the time of the system's construction in the summer of 2016; its specifications and SM architecture are shown in Table 4.1 and Figure 4.1, respectively.

Table 4.1: *GeForce GTX 1080 Specifications*

Specification	Rating
Compute capability	6.1
Streaming multiprocessors	20
CUDA cores	2560 total (128 per SM)
Maximum core clock rate	1734 MHz
GFLOPS	8873 [25]
Memory	8 GiB GDDR5X
Memory clock rate	5005 MHz
Memory bus width	256-bit
Memory bandwidth	320 GiB/sec [25]
L2 cache	2 MiB
Max threads per block/SM	1024/2048

Figure 4.1: *GeForce GTX 1080 Streaming Multiprocessor* [25]

The operating system used is openSUSE 42.1 installed with the 367.27 version of Nvidia's graphics/CUDA drivers.

In order to eliminate pseudorandom statistical weaknesses, all random matrices were generated using Intel's hardware random number generator (unless otherwise noted). Each random matrix is populated with integer elements no greater than the matrix dimension in magnitude[1]; allowing much larger or non-integer elements didn't seem to significantly affect any of the results.

Several different classes of matrices were inverted using ACMI and shall be ex-

[1]Diagonals in diagonally-dominant matrices are necessarily not so bounded.

amined in the following sections. Single and double precision on the GPU, single precision on the CPU[2] and GNU Octave were all compared by the speed and quality of their inversions. Every single-precision run was automatically promoted to double precision when necessary to continue the inversion. Each test was performed five times to produce an averaged result.

Octave is a free and open-source numerical computing platform that is largely compatible with MATLAB. Depending upon the characteristics of an input matrix, Octave performs one of several methods of inversion, including LU and Cholesky factorization [12] [2]; Octave thus provides a method of testing ACMI against traditional matrix inversion methods. Octave itself depends upon BLAS and was configured to use the same multithreaded OpenBLAS library used by ACMI.

CUDA kernel launches were tuned to saturate all SMs on the GeForce GPU, allocating up to 40 blocks per grid with 1024 threads per block. The OpenBLAS routines used to perform inversion in software use up to 8 POSIX threads to execute their computations in parallel. The time required to generate or load matrices and to transfer them to GPU memory is not figured into the tested inversion runtimes.

4.2 Diagonally-Dominant Random Matrices

We first tested ACMI against easily-invertible random matrices. A matrix A is *strictly diagonally-dominant* [6] if, for all $1 \leq i \leq n$,

$$|a_{ii}| > \sum_{j=1,\, j\neq i}^{n} |a_{ij}| \,.$$

Such a matrix is easily, rapidly and accurately solved by Gaussian elimination [13]. Forcing the diagonal elements to be positive well conditions[3] the matrix, altogether making the matrix easily invertible.

[2]Double-precision CPU inversions using ACMI aren't studied as they generally form a higher and unnecessary runtime upper bound.

[3]Most such matrices have a condition number of about 2.

We generated diagonally-dominant matrices with dimensions ranging from 128×128 to 16384×16384. Five matrices were generated for each size tested and their inversion runtime averages are graphed in Figure 4.2, where ACMI's double-precision, single-precision (with promotion to double precision if needed) and CPU inverters are compared to each other and to GNU Octave's matrix inverter. The target error

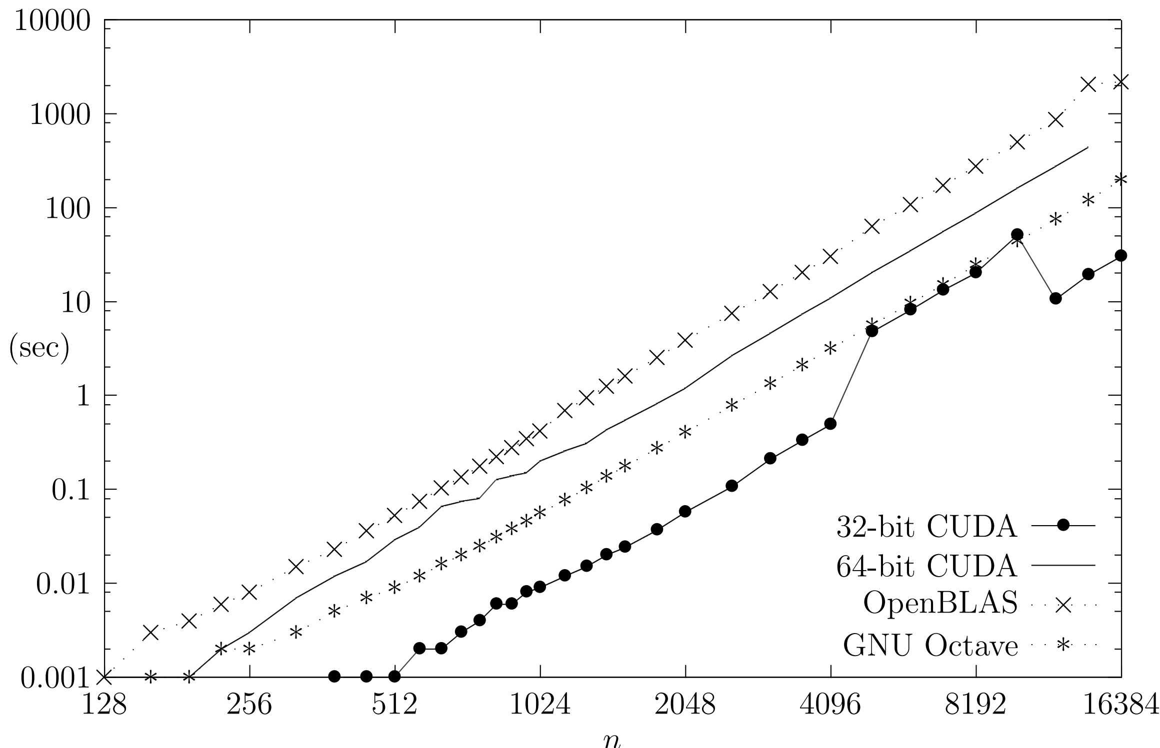

Figure 4.2: *Inversion Runtime for $n \times n$ Diagonally-Dominant Random Matrices*

measure limit for each inversion was $1.15 \cdot 10^{-5}$; this was chosen to be slightly over ACMI's default of 10^{-5} in order to accommodate some large matrices that could not be promoted to double precision due to memory constraints.

The single-precision ACMI inverter generally and greatly outperforms all its competitors, taking 30.4 seconds on average to invert a 16384×16384 matrix while its nearest competitor, GNU Octave, takes 198 seconds—6.5 times as long. Octave outperformed ACMI once during the hump in the latter's curve; this range of matrix dimensions represents a "region" of instability that required ACMI to promote its matrices to double precision in order to meet the inversion error target. As can be

seen in the curve of OpenBLAS inversions performed by ACMI, this region is peculiar to ACMI's CUDA inverter. We do not yet know why our CUDA inverter has trouble with diagonally-dominant matrices of these dimensions. An observation that held across all tests is that, once promoted from single precision, an inversion only rarely requires more than two additional iterations to meet its error target[4]. Despite this and due to the double-precision hardware limitations of GeForce GPUs, these promoted iterations dominate the runtimes of their inversions.

To minimize the extent of this region, we had to raise the convergence rate limit[5] to $\mu < 5 \cdot 10^7$; limits as high as $\mu < 5 \cdot 10^5$ extend the ranges of matrix sizes affected by this single-precision CUDA anomaly. This limit was found by trial-and-error and is poorly-understood; nevertheless, it shows that ACMI's default limit of $\mu < 1$ is hardly as general-purpose as previously hoped. The only way to eliminate the region is to raise the maximum-tolerated error to about $1 \cdot 10^{-3}$, which is unacceptably high.

Even when inverting matrices whose sizes lie within this region, single-precision inversions with promotion dramatically outperform pure double-precision runs by a factor of 23 with respect to runtime. ACMI's double-precision CUDA and single-precision CPU inverters are beaten here by Octave, also by large factors; this might be due to the sophistication of Octave as much as it is to the ease with which diagonally-dominant matrices are solved by Gaussian elimination.

These diagonally-dominant matrices require few ACMI iterations to invert: the smaller matrices tested require 5-6 iterations, 1024×1024 matrices require 7 and the largest require 8-10.

The largest specimen inverted during the entirety of this research was a 22800×22800 diagonally-dominant matrix, taking 91.9 seconds to invert to under $1.2 \cdot 10^{-5}$ error in 9 iterations. The inversion required almost the entirety of the GPU's 8 GiB of RAM and was performed entirely in single precision using ACMI. ACMI took over

[4]This observation no longer holds when the convergence rate limit is set to be very low.
[5]See Section 3.4 for a description of this limit.

110 minutes to invert such a matrix within the same error bound using the CPU alone; Octave needed 513 seconds.

4.3 Random Matrices

We then tested fully-random, non-diagonally-dominant matrices where negative elements are allowed. While such matrices, unlike their diagonally-dominant cousins, may be singular, they prove in practice to almost always be invertible[6]. Figures 4.3 and 4.4 show the runtime results for random integer and Boolean matrices, respectively. The runtime behaviors of these two classes are remarkably similar, though the

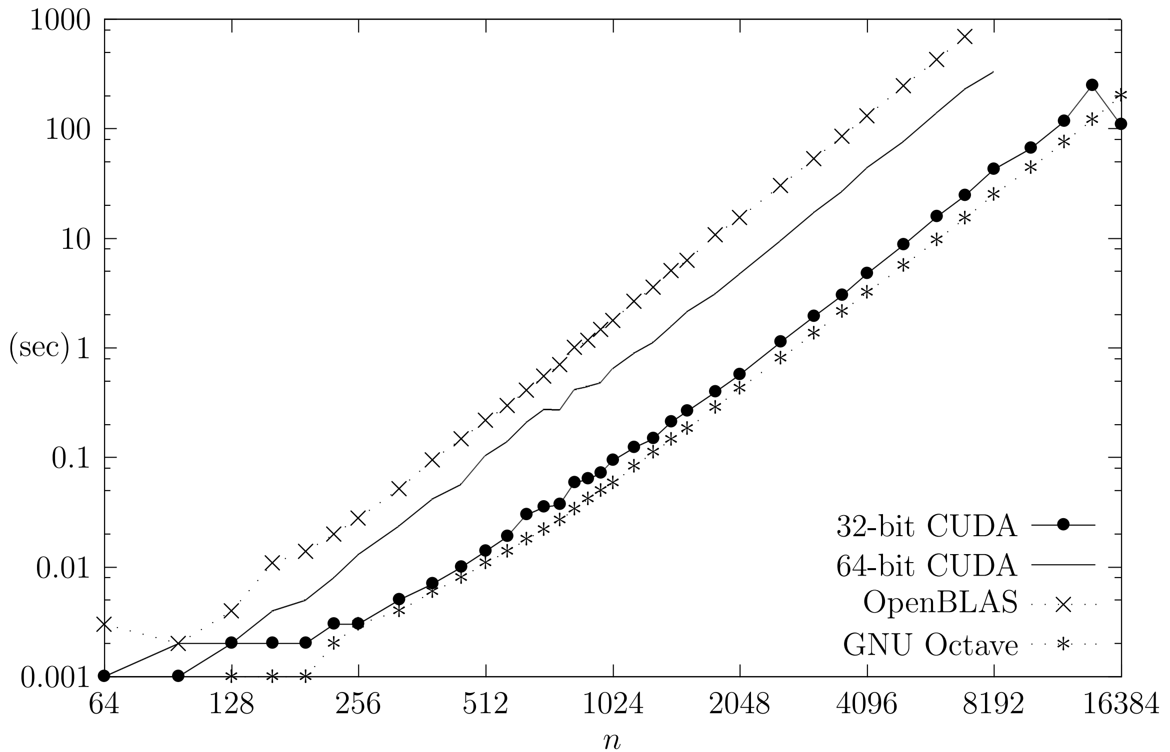

Figure 4.3: *Inversion Runtime for $n \times n$ Random Matrices*

raw figures show that Boolean matrices are slightly harder to invert. A convergence rate limit of $\mu < 5$ was chosen as most of these inversions would diverge if pushed beyond this limit at single precision. This limit was also found by trial-and-error and

[6]During this research, random singularity was only observed with very small matrices—so small that they did not figure into our experiments.

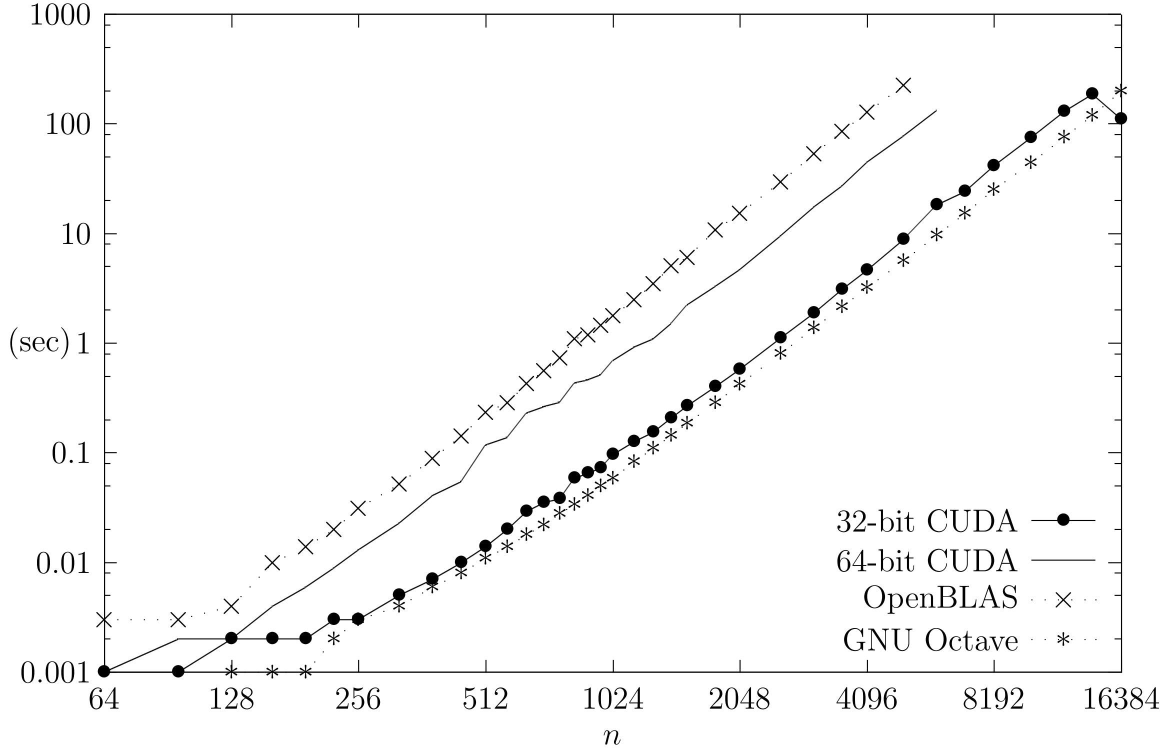

Figure 4.4: *Inversion Runtime for $n \times n$ Boolean Matrices*

differs greatly from that used to optimize the inversions of diagonally-dominant matrices, suggesting that computation of the ideal such limit is non-trivial. The default ACMI inversion error target 10^{-5} was used for all tests.

While ACMI using single precision (with promotion) takes only a ninth of the time needed by double precision to invert a random 8192×8192 integer matrix (36.6 vs. 332 seconds), GNU Octave defeated ACMI for nearly all matrix sizes tested[7], taking only 25 seconds to invert an 8192×8192 matrix.

In contrast to diagonally-dominant matrices, fully-random matrices tend to be poorly-conditioned[8] and possess negative eigenvalues, requiring ACMI to use the general-purpose R_0 initial inverse approximation to successfully invert these matrices. This alternate initial approximation appears to be the culprit for ACMI's poor

[7]While ACMI appears to have won at inverting the largest matrix size tested here, the resulting approximate inverses had poor error measures on the order of 10^{-2} as there was insufficient memory for promotion to double precision.

[8]The condition numbers are on the order of 10^4; ACMI must be told to skip the positive-definite R_0 approximation lest it waste a couple iterations before restarting.

performance, sometimes requiring over 30 iterations to invert within the error target (see the $p = 3$ graph in Figure 4.9).

4.4 Real-World Large Matrices

We, of course, wished to demonstrate practical applications of ACMI by testing it against some large, non-random matrices modeling real-world problems. To this end, we employed the University of Florida Sparse Matrix collection [10], choosing the matrices listed in Table 4.2 as test subjects. Runtime results are shown in Figure 4.5; since single-precision ACMI clearly dominates its other two modes, we compared its results to GNU Octave alone. Here, the default convergence rate limit of $\mu < 1$ was used to no apparent ill effect.

Table 4.2: *Sparse Matrix Collection Specimens* [10]

ID:	Matrix	n	Description
1:	494_bus	494	Bus power system
2:	685_bus	685	Bus power system
3:	1138_bus	1138	Bus power system
4:	CAG_mat1916	1916	Combinatorial problem
5:	ex37	3565	Computational fluid dynamics problem
6:	lshp_265	265	Thermal problem
7:	Kuu	7102	Structural problem
8:	heart3	2339	Quasi-static FE model of a heart
9:	c-40	9941	Non-linear optimization problem
10:	p2p-Gnutella25	22687	Peer-to-peer file sharing network
11:	pli	22695	Magnetohydrodynamic problem

For all but two (small) matrices, ACMI defeated Octave[9], taking 66.5 seconds to

[9] Octave couldn't invert the largest two matrices within a reasonable time limit, hence the truncated results in the histogram.

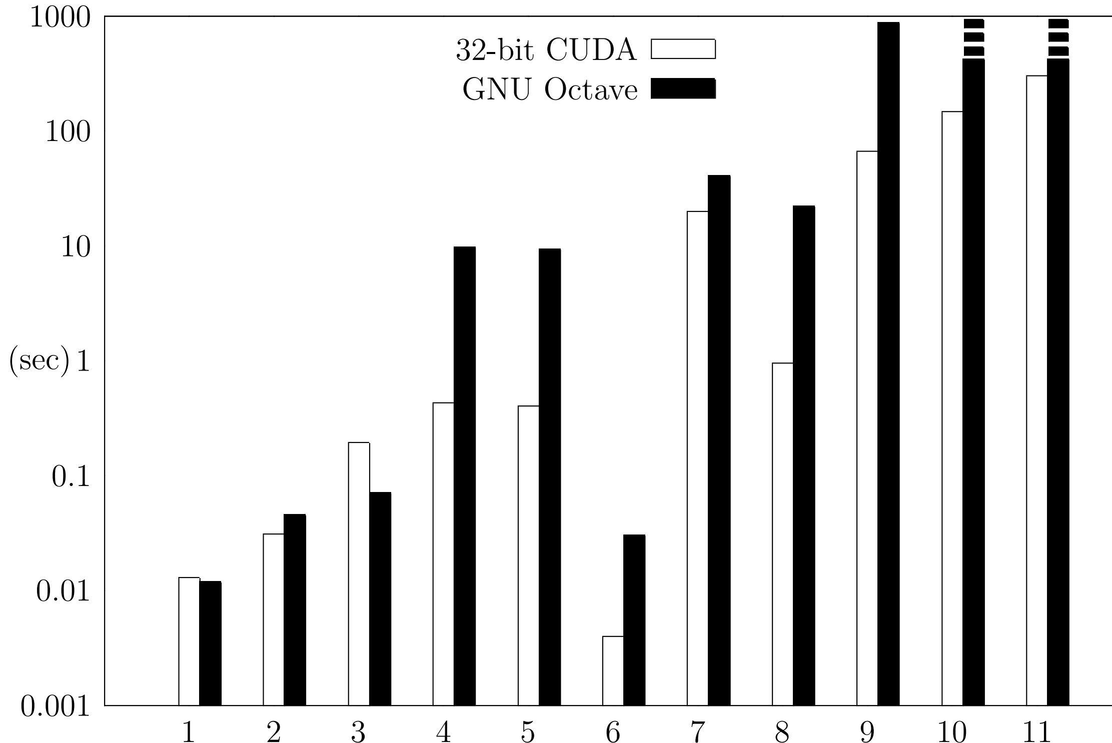

Figure 4.5: *Inversion Runtime for Large, Sparse Matrices*

invert the "c-40" matrix whereas Octave required 879. Unlike with random matrices, ACMI wins even at the non-positive-definite tests (8-11) that require the much more slowly-convergent, general-purpose R_0 seed. These matrices aren't necessarily "friendlier" than the fully-random matrices, since the "heart3" matrix happened to have a much worse condition number of $2.1 \cdot 10^6$. Among these tests, ACMI needed to compute as few as 10 iterations and as many as 32 for the "ex37" and "c-40" matrices, respectively.

The "heart3" matrix happened to "break" ACMI's automatic detection of poor initial inverses: whereas every other tested matrix inversion diverged within the first two iterations if the positive-definite R_0 were wrongly chosen (successfully triggering ACMI to restart the inversion from the general-purpose R_0 in the process), this matrix produced nine iterations from the positive-definite seed before diverging.

As an example of the superior convergence of the positive-definite R_0 approximation, forcing ACMI to use the general-purpose R_0 to invert the "Kuu" matrix results

in a runtime of 24.6 seconds over 27 iterations instead of 20 seconds over 15 iterations.

4.5 Hilbert Matrices

The numerical stability of an invertible matrix A can be predicted by its *condition number*: $\|A\| \cdot \|A^{-1}\|$. A condition number significantly greater than one indicates an *ill-conditioned* matrix where small perturbations of the elements have great influence on operations performed on the matrix and their results [6] [13]. The higher the condition number, the greater the error inflation. Naturally, an ill-conditioned matrix won't tolerate floating-point round-off error well and shall complicate ACMI's computations, possibly even making effective matrix inversion impossible.

Despite being positive definite, the symmetric Hilbert matrices are notoriously ill-conditioned [6]; even the tiny 10×10 Hilbert matrix has a condition number of $1.6 \cdot 10^{13}$ [13]. They thus provided an extreme under which we could test the numerical stability of ACMI. The $n \times n$ Hilbert matrix is

$$H_n = \begin{pmatrix} 1 & \frac{1}{2} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+1} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \cdots & \frac{1}{2n-1} \end{pmatrix}.$$

Figure 4.6 shows the best inversion error measures achieved using ACMI in double precision to invert H_2 through H_{13} and compares them to those obtained by Octave. Inversion runtimes are negligible across all methods at these dimensions and are not shown; these tests serve to show error tolerance, not parallel performance.

ACMI's CPU and GPU inverters perform similarly, both producing inverses of far greater fidelity than Octave. Despite the fact that these matrices all benefit from the positive-definite R_0 approximation, they require many iterations to reach their best error measures: 14 iterations for H_3 and 35 for H_{12}. Altering the convergence rate limit when inverting Hilbert matrices didn't affect the total iterations computed, but did increase the proportion of double precision iterations needed by ACMI.

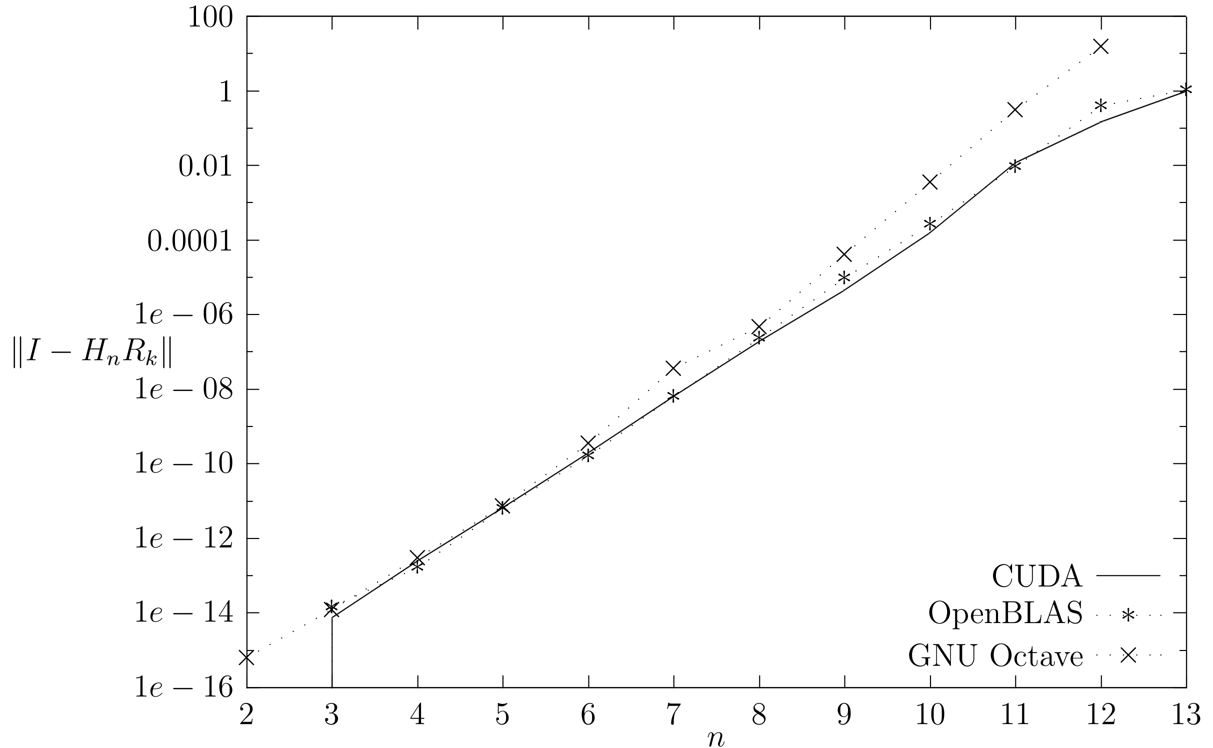

Figure 4.6: *Inversion Error for $n \times n$ Hilbert Matrices*

4.6 Comparison of the Orders of Convergence

Thus far, we had only been computing the cubically-convergent Altman sequence of inverse approximations using ACMI. Since the quadratically and quartically-convergent sequences produce more quickly computable and convergent iterations, respectively, we asked whether parallel computation through CUDA ever grants them an advantage over the sequence of cubic order; the latter is optimal in theory, but not necessarily so with finite-precision software. Figures 4.7 and 4.8 compare the three sequences when applied to diagonally-dominant random and fully-random matrices, respectively. The number of iterations computed by each sequence for the latter case are shown in Figure 4.9. All tests are run in single precision, yet typically promote to double precision for the last two iterations. The same convergence rate limits are used as in the previous sections dealing with these matrix classes.

Surprisingly, the cubic-order sequence is frequently outperformed by the quadratic,

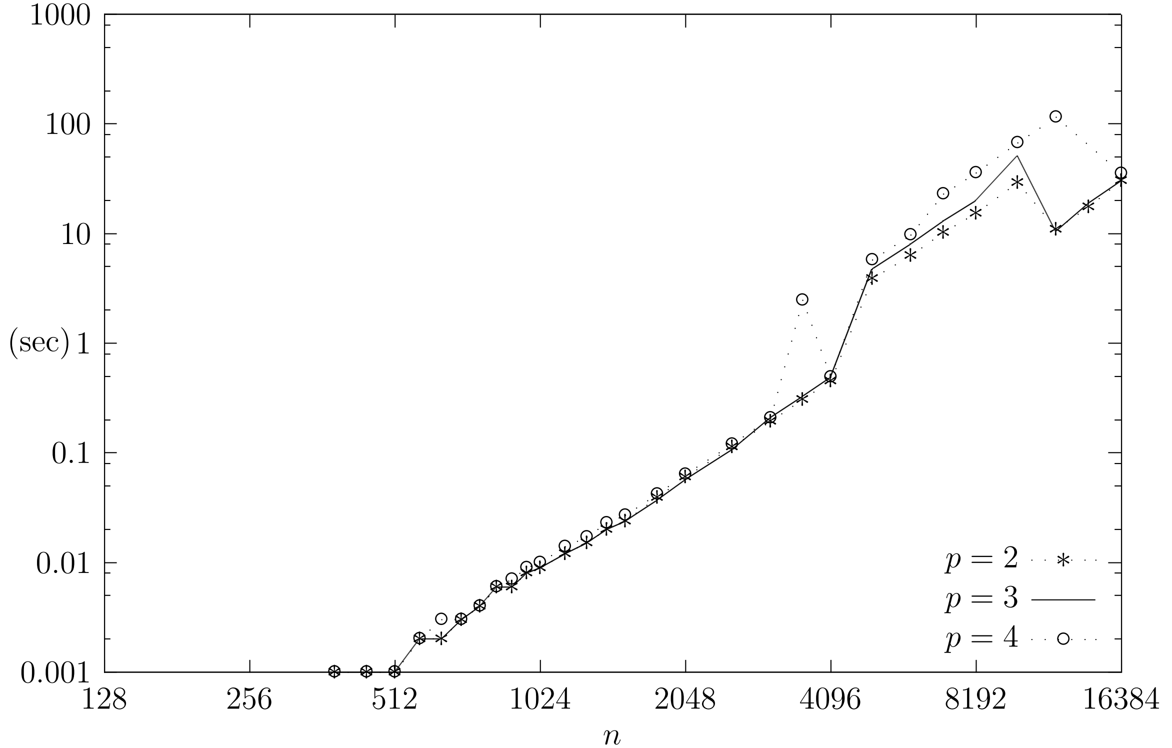

Figure 4.7: *Order-p Inversion Runtime for Diagonally-Dominant Random Matrices*

despite the latter requiring many more iterations to perform each inversion. This result appears to be due to the finite precision and numerical instability of our matrix arithmetic. For each test, all three sequences tend to promote to double precision after similar inversion error measures have been achieved. The runtime dominance of the following few double-precision iterations, which are much more expensive for higher-order sequences, cover up the slower convergence of the lower-order sequence iterations performed in single precision.

It should be noted that, given its slower convergence, the quadratic sequence is more sensitive to lower convergence rate limits and requires them to be raised in order to compete with the other sequences. When this value is low and fixed for all sequences, the cubic-order sequence substantially outperforms the quadratic. Generally, even though a low convergence rate limit can lower the total number of inversion iterations performed, it likely raises the ratio of double-precision to single-precision iterations at the expense of the runtime.

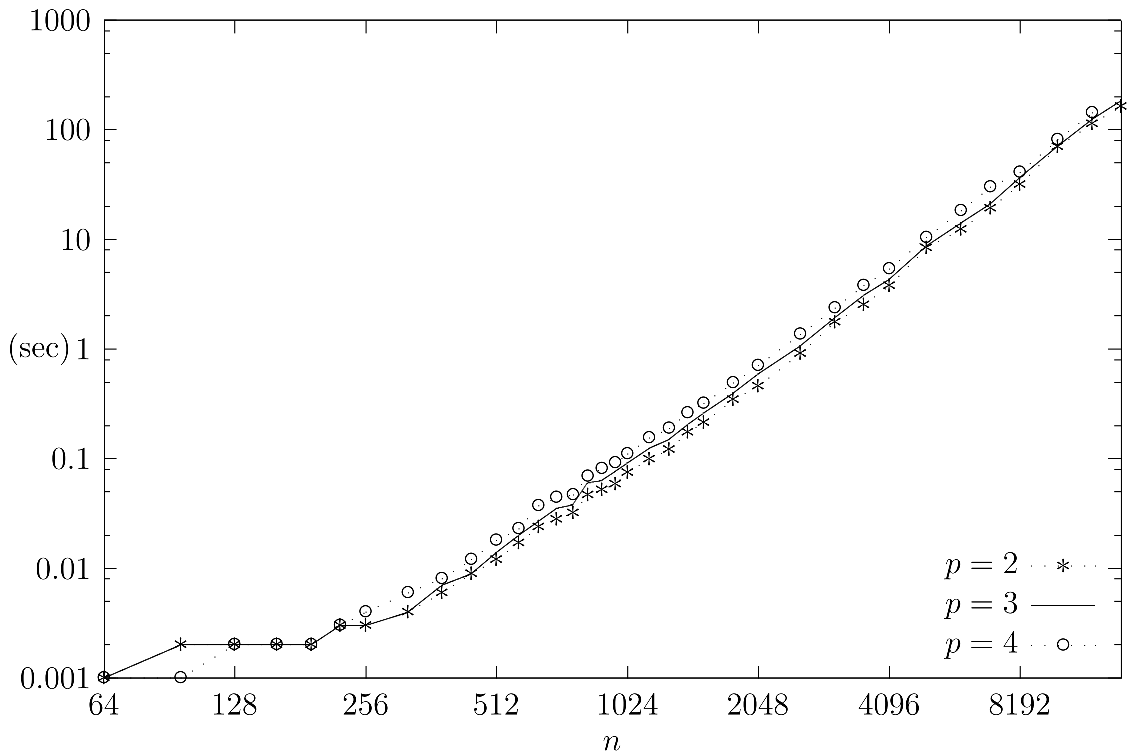

Figure 4.8: *Order-p Inversion Runtime for Random Matrices*

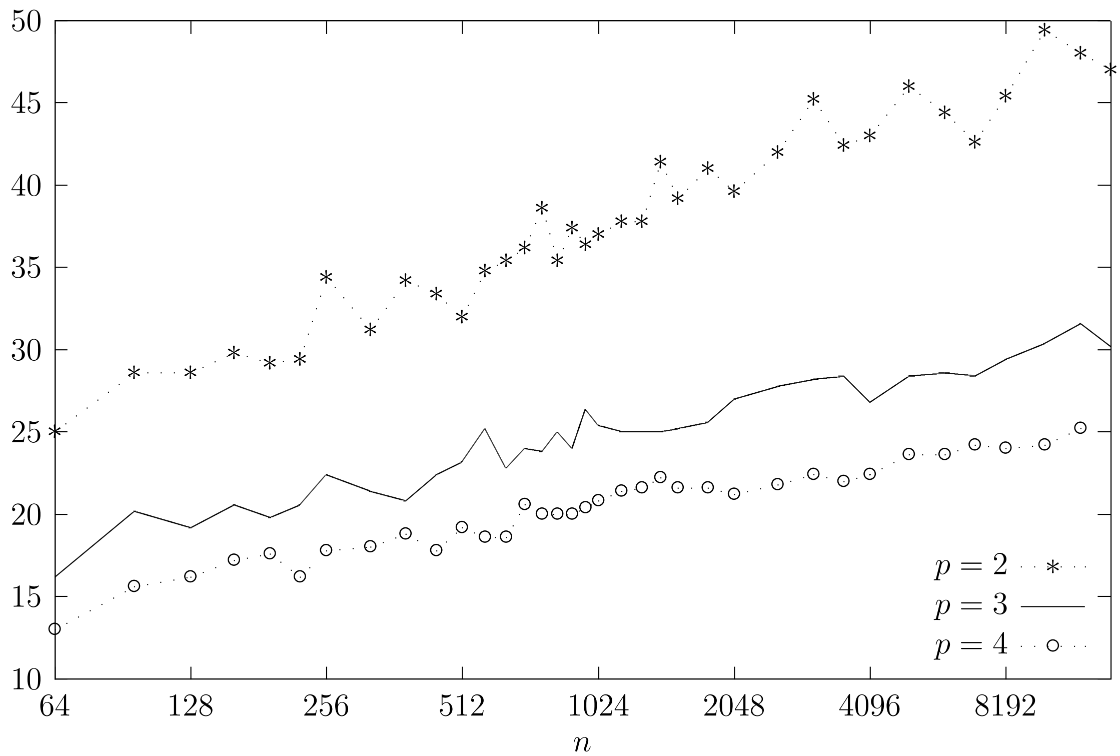

Figure 4.9: *Total Order-p Inversion Iterations for Random Matrices*

CHAPTER V

CONCLUSION

5.1 Implications of Results

A parallel implementation of Altman's matrix inversion method using a consumer-grade GPU and single-precision arithmetic can greatly outperform other matrix inverters, such as GNU Octave. Moreover, the method is error tolerant and allows extended single-precision computation until double precision is absolutely necessary. This runtime advantage is not present when executing the method entirely with double precision or on the CPU. While our implementation defeated GNU Octave at inverting large, easily-invertible matrices as well as matrices representing large scientific and industrial problems, Octave outperformed the implementation at inverting certain ill-conditioned random matrices.

We attempted to exploit the inversion sequence's rate of convergence to predict oncoming divergence and optimize the inversion strategy. The convergence rate limit proved non-trivial to optimize and even increased the inversion computation time when set too low. On the hardware used, a few double-precision iterations dominated the runtime of many single-precision iterations to the extent that attempting to minimize the latter proved to not be worthwhile.

5.2 Further Research

This research studied GPGPU matrix inversion using only consumer Nvidia GPUs through the CUDA framework. Since these consumer units perform relatively poorly during double-precision computations, a study of the performance of ACMI on professional Tesla units would be a logical next step. Adding support for other GPGPU platforms, such as OpenCL and Vulkan, should also be of interest.

Matrix inversion very quickly exhausts single-precision numerical stability and even double precision frequently proves to be inadequate for ACMI. Unfortunately, quadruple-precision support is currently unavailable outside of relatively-exotic IBM mainframes [31] and Power CPUs [14]. A compromise of worthwhile pursuit would be the *double-double* method, which extends floating-point precision by using pairs of double-precision numbers to represent each vector or matrix element [17].

As discovered in Chapter 4, certain ranges of dimensions of diagonally-dominant random matrices are much less numerically stable when using CUDA and cuBLAS and require promotion to much slower double-precision computation to invert well. CPU inversion using OpenBLAS doesn't suffer this instability; an explanation for why this is would be appreciated.

Finally, Strassen's matrix multiplication algorithm has acceptable numerical stability for some applications and scales very well with parallel execution [9]. A GPU implementation of this algorithm might appreciably accelerate Altman's method.

REFERENCES

[1] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/. Accessed: 2016-10-15.

[2] ?getrf. https://software.intel.com/en-us/node/520877. Accessed: 2016-10-25.

[3] M. Altman. *Approximation Methods in Functional Analysis*. California Institute of Technology, 1959.

[4] M. Altman. An optimum cubically convergent iterative method of inverting a linear bounded operator in Hilbert space. *Pacific Journal of Mathematics*, 10(4):1107–1113, December 1960.

[5] Bradley, Hax, and Magnanti. *Applied Mathematical Programming*. Addison-Wesley, 1977.

[6] R. Burden and J. Faires. *Numerical Analysis*. Cengage Learning, 9th edition, August 2010.

[7] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 1st edition, September 2014.

[8] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, March 1990.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[10] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011. Accessed: 2016-10-27.

[11] J. L. Dongarra and V. Eijkhout. Numerical linear algebra algorithms and software. *Journal of Computational and Applied Mathematics*, 123:489–514, November 2000.

[12] J. W. Eaton. Techniques used for linear algebra. https://www.gnu.org/software/octave/doc/interpreter/Techniques-Used-for-Linear-Algebra.html, 2016. Accessed: 2016-10-25.

[13] W. Gautschi. *Numerical Analysis*. Birkhäuser, 2nd edition, 2012.

[14] GCC Team. GCC 6 release series: Changes, new features, and fixes. https://gcc.gnu.org/gcc-6/changes.html, October 2016. Accessed: 2016-10-28.

[15] H. Hagedoorn. Nvidia GeForce GTX 1080 review - Pascal GPU architecture. http://www.guru3d.com/articles-pages/nvidia-geforce-gtx-1080-review,3.html, May 2016. Accessed: 2016-10-20.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 1st edition, June 2012.

[17] Y. Hida, X. S. Li, and D. H. Bailey. Library for double-double and quad-double arithmetic. December 2007.

[18] Intel. *Intel Digital Random Generator (DRNG) Software Implementation Guide*, 1.1 edition, August 2012.

[19] J. Li, S. Ranka, and S. Sahni. Strassen's matrix multiplication on GPUs. *Parallel and Distributed Systems (ICPADS)*, December 2011.

[20] J. Liesen and V. Mehrmann. *Linear Algebra.* Springer, 1st edition, December 2015.

[21] G. Marsaglia. Random numbers fall mainly in the planes. *Proc. N. A. S.*, 61(1):25–28, September 1968.

[22] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), July 2003.

[23] K. Milfeld. GotoBLAS2. https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2. Accessed: 2016-10-14.

[24] NVIDIA Corporation. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide, September 2016. Accessed: 2016-10-16.

[25] NVIDIA Corporation. NVIDIA GeForce GTX 1080. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016. Accessed: 2016-10-20.

[26] NVIDIA Corporation. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. Accessed: 2016-10-20.

[27] National Institute of Standards and Technology. ANSI C library for Matrix Market I/O. http://math.nist.gov/MatrixMarket/mmio-c.html, May 2000. Accessed: 2016-10-21.

[28] V. Pan, Z. Chen, and A. Zheng. The complexity of the matrix eigenproblem. *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 507–516, May 1999.

[29] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1992.

[30] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18:109–144, February 1982.

[31] E. Schwarz. The IBM z13 SIMD accelerators for integer, string, and floating-point. http://arith22.gforge.inria.fr/slides/s1-schwarz.pdf, June 2015. Accessed: 2016-10-28.

[32] R. Senser. *GPU Declarative Framework*. PhD thesis, University of Colorado Denver, November 2014.

[33] T. Valich. Pascal secrets: what makes Nvidia GeForce GTX 1080 so fast? http://vrworld.com/2016/05/10/pascal-secrets-nvidia-geforce-gtx-1080, May 2016. Accessed: 2016-10-20.

[34] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 1st edition, June 2013.

[35] Z. Xianyi. OpenBLAS wiki. https://github.com/xianyi/OpenBLAS/wiki. Accessed: 2016-10-14.

APPENDIX A

ACMI INVERTER USAGE

Listing A.1: ACMI Inverter Usage

```
1  ACMI Convergent Matrix Inverter
2  J. Treadwell, 2016
3
4  Usage:
5    acmi [options] <input file>
6
7    Only Matrix Market I/O is supported. To generate and invert a
8    random matrix, enter the matrix dimension prepended by '@' in lieu
9    of the input file; prepend with '%' for symmetry.
10
11 Options:
12   -f           Print matrix file info and exit
13   -c           Perform all computations in software without the GPU
14   -l           Use low-speed, safe initial R0 approximation
15   -o <path>    Output computed matrix inverse to path
16   -q <order>   Set the order of convergence (2-4, default: cubic)
17   -p <#bits>   Set initial matrix element floating-point precision
18                (32 or 64, default: 32)
19   -e <+real>   Set inversion error limit (default: 1e-05)
20   -t <ms>      Set inversion time limit in ms (default: none)
21   -m <+real>   Set max-allowed single-precision convergence rate;
22                prepend with 'x' and set to >= 1 to use a multiple of
23                the starting rate (default: 1)
24   -b <+int>    Set max blocks run by each GPU kernel
25                (default: 40, max: 65535)
26 Random matrix options:
27   -H           Enable hardware random number generator (unseedable)
28   -R           Enable real elements
29   -N           Enable negative elements
30   -D           Generate dominant diagonal elements
31   -V <+real>   Set max element magnitude
32                (default: matrix dimension, max: 32767)
33   -U <path>    Output generated, uninverted matrix to path
34   -S <hex>     Set PRNG seed (not yet portable)
```

APPENDIX B

SAMPLE ACMI INVERSION RUNS

Listing B.1: GPU single-precision, random diagonally-dominant ($n = 2048$).

```
 1  > ./acmi @2048 -DS deadbeef
 2  7998.438 MiB device memory available
 3  seeding PRNG with deadbeef
 4  generating 32-bit random 2048x2048 integer nonnegative
 5    diagonally-dominant matrix...
 6  16.000 MiB/matrix; allocating 64.000 MiB total
 7  setting up cuBLAS
 8  setting up kernels on GeForce GTX 1080
 9  blocks/matrix: 40, threads/matrix: 40960
10  blocks/vector: 2, threads/vector: 2048
11  uploading matrix to device
12  inverting random matrix with cubic convergence...
13  initializing work matrices...
14  computed alpha = 1.05396e-08
15   R0: err = 4.4255e+01, μ = 0.0000e+00  (0.000s)
16   R1: err = 4.2322e+01, μ = 4.8828e-04  (0.000s)
17   R2: err = 3.7015e+01, μ = 4.8829e-04  (0.010s)
18   R3: err = 2.4766e+01, μ = 4.8835e-04  (0.008s)
19   R4: err = 7.4232e+00, μ = 4.8870e-04  (0.007s)
20   R5: err = 2.0094e-01, μ = 4.9125e-04  (0.008s)
21   R6: err = 5.9508e-05, μ = 7.3342e-03  (0.008s)
22   R7: err = 5.9787e-06, μ = 2.8372e+07  (0.007s)
23  inversion halted in 7 iterations after 0m0.056s
24  shutting down cuBLAS
25  shutting down kernels
```

Listing B.2: GPU double-precision, random diagonally-dominant ($n = 2048$).

```
 1  > ./acmi @2048 -DS deadbeef -p 64
 2  7998.438 MiB device memory available
 3  seeding PRNG with deadbeef
 4  generating 64-bit random 2048x2048 integer nonnegative
 5    diagonally-dominant matrix...
 6  32.000 MiB/matrix; allocating 128.000 MiB total
 7  setting up cuBLAS
 8  setting up kernels on GeForce GTX 1080
 9  blocks/matrix: 40, threads/matrix: 40960
10  blocks/vector: 2, threads/vector: 2048
11  uploading matrix to device
12  inverting random matrix with cubic convergence...
13  initializing work matrices...
14  computed alpha = 1.05396e-08
15   R0: err = 4.4255e+01, μ = 0.0000e+00  (0.000s)
16   R1: err = 4.2322e+01, μ = 4.8828e-04  (0.000s)
17   R2: err = 3.7015e+01, μ = 4.8829e-04  (0.272s)
18   R3: err = 2.4766e+01, μ = 4.8835e-04  (0.185s)
19   R4: err = 7.4229e+00, μ = 4.8868e-04  (0.182s)
20   R5: err = 2.0076e-01, μ = 4.9086e-04  (0.182s)
21   R6: err = 4.1234e-06, μ = 5.0957e-04  (0.183s)
22  inversion halted in 6 iterations after 0m1.186s
23  shutting down cuBLAS
24  shutting down kernels
```

Listing B.3: CPU single-precision, random diagonally-dominant ($n = 2048$).

```
 1  > ./acmi @2048 -DS deadbeef -c
```

```
 2  seeding PRNG with deadbeef
 3  generating 32-bit random 2048x2048 integer nonnegative
 4    diagonally-dominant matrix...
 5  16.000 MiB/matrix; allocating 64.000 MiB total
 6  GPU acceleration disabled!
 7  inverting random matrix with cubic convergence...
 8  initializing work matrices...
 9  computed alpha = 1.05396e-08
10   R0: err = 4.4255e+01, μ = 0.0000e+00  (0.003s)
11   R1: err = 4.2322e+01, μ = 4.8828e-04  (0.530s)
12   R2: err = 3.7015e+01, μ = 4.8829e-04  (0.523s)
13   R3: err = 2.4766e+01, μ = 4.8835e-04  (0.523s)
14   R4: err = 7.4229e+00, μ = 4.8868e-04  (0.523s)
15   R5: err = 2.0077e-01, μ = 4.9089e-04  (0.523s)
16   R6: err = 1.7253e-05, μ = 2.1318e-03  (0.523s)
17   R7: err = 3.8735e-06, μ = 7.5426e+08  (0.523s)
18  inversion halted in 7 iterations after 0m3.846s
```

Listing B.4: Laptop GPU single-precision, random diagonally-dominant ($n = 2048$).

```
 1  > ./acmi @2048 -DS deadbeef
 2  850.562 MiB device memory available
 3  seeding PRNG with deadbeef
 4  generating 32-bit random 2048x2048 integer nonnegative
 5    diagonally-dominant matrix...
 6  16.000 MiB/matrix; allocating 64.000 MiB total
 7  setting up cuBLAS
 8  setting up kernels on NVS 4200M
 9  blocks/matrix: 40, threads/matrix: 40960
10  blocks/vector: 2, threads/vector: 2048
11  uploading matrix to device
12  inverting random matrix with cubic convergence...
13  initializing work matrices...
14  computed alpha = 1.05396e-08
15   R0: err = 4.4255e+01, μ = 0.0000e+00  (0.004s)
16   R1: err = 4.2322e+01, μ = 4.8828e-04  (0.000s)
17   R2: err = 3.7014e+01, μ = 4.8829e-04  (0.948s)
18   R3: err = 2.4766e+01, μ = 4.8836e-04  (0.714s)
19   R4: err = 7.4232e+00, μ = 4.8870e-04  (0.713s)
20   R5: err = 2.0094e-01, μ = 4.9125e-04  (0.714s)
21   R6: err = 5.9371e-05, μ = 7.3176e-03  (0.713s)
22   R7: err = 5.9608e-06, μ = 2.8482e+07  (0.714s)
23  inversion halted in 7 iterations after 0m5.234s
24  shutting down cuBLAS
25  shutting down kernels
```

Listing B.5: Single-precision, random ($n = 4096$) with promotion to double precision.

```
 1  > ./acmi -HD @4096
 2  7998.438 MiB device memory available
 3  using RDRAND RNG
 4  generating 32-bit random 4096x4096 integer nonnegative
 5    diagonally-dominant matrix...
 6  64.000 MiB/matrix; allocating 256.000 MiB total
 7  setting up cuBLAS
 8  setting up kernels on GeForce GTX 1080
 9  blocks/matrix: 40, threads/matrix: 40960
10  blocks/vector: 4, threads/vector: 4096
11  uploading matrix to device
12  inverting random matrix with cubic convergence...
13  initializing work matrices...
14  computed alpha = 1.86387e-09
15   R0: err = 6.3000e+01, μ = 0.0000e+00  (0.001s)
16   R1: err = 6.1047e+01, μ = 2.4414e-04  (0.000s)
17   R2: err = 5.5544e+01, μ = 2.4414e-04  (0.079s)
```

```
18   R3: err = 4.1838e+01, μ = 2.4415e-04  (0.059s)
19   R4: err = 1.7885e+01, μ = 2.4421e-04  (0.058s)
20   R5: err = 1.3999e+00, μ = 2.4472e-04  (0.059s)
21   R6: err = 5.3994e-04, μ = 1.9681e-04  (0.059s)
22   R7: err = 5.3156e-05, μ = 3.3769e+05  (0.059s)
23  diverging, extending to double precision...
24   R8: err = 4.5140e-04, μ = 0.0000e+00  (0.064s)
25   R9: err = 2.9680e-13, μ = 3.2268e-03  (1.479s)
26  inversion halted in 9 iterations after 0m3.382s
27  shutting down cuBLAS
28  shutting down kernels
```

Listing B.6: Excerpt of an attempt to invert within an unachievable error limit with divergence detection disabled, showing self-corrective bouncing where double-precision numerical stability ends ($n = 1024$).

```
 1  R18: err = 1.1310e+00, μ = 3.2129e-01  (0.026s)
 2  R19: err = 9.2789e-01, μ = 6.4140e-01  (0.027s)
 3  R20: err = 7.5785e-01, μ = 9.4861e-01  (0.026s)
 4  R21: err = 4.3523e-01, μ = 9.9993e-01  (0.029s)
 5  R22: err = 8.2442e-02, μ = 1.0000e+00  (0.027s)
 6  R23: err = 5.6034e-04, μ = 1.0000e+00  (0.027s)
 7  R24: err = 1.7601e-10, μ = 1.0004e+00  (0.027s)
 8  R25: err = 4.8498e-12, μ = 8.8943e+17  (0.029s)
 9  R26: err = 5.0036e-12, μ = 4.3863e+22  (0.026s)
10  R27: err = 4.9887e-12, μ = 3.9825e+22  (0.025s)
11  R28: err = 5.0116e-12, μ = 4.0365e+22  (0.028s)
12  R29: err = 5.1043e-12, μ = 4.0550e+22  (0.028s)
13  R30: err = 4.8396e-12, μ = 3.6393e+22  (0.027s)
14  R31: err = 4.8361e-12, μ = 4.2664e+22  (0.028s)
15  R32: err = 5.0029e-12, μ = 4.4231e+22  (0.028s)
16  R33: err = 4.9357e-12, μ = 3.9418e+22  (0.027s)
```

Listing B.7: 10×10 Hilbert matrix.

```
 1  > ./acmi mats/hilb10.mtx -p 64
 2  7998.438 MiB device memory available
 3  loading mats/hilb10.mtx
 4  matrix is 10x10 and 0.001 MiB in size
 5  0.001 MiB/matrix; allocating 0.003 MiB total
 6  setting up cuBLAS
 7  setting up kernels on GeForce GTX 1080
 8  blocks/matrix: 1, threads/matrix: 100
 9  blocks/vector: 1, threads/vector: 10
10  uploading matrix to device
11  inverting mats/hilb10.mtx with cubic convergence...
12  initializing work matrices...
13  computed alpha = 0.560059
14   R0: err = 2.9344e+00, μ = 0.0000e+00  (0.000s)
15   R1: err = 2.8557e+00, μ = 1.1302e-01  (0.000s)
16   R2: err = 2.7731e+00, μ = 1.1908e-01  (0.000s)
17   R3: err = 2.6940e+00, μ = 1.2634e-01  (0.000s)
18   R4: err = 2.6116e+00, μ = 1.3357e-01  (0.000s)
19   R5: err = 2.5429e+00, μ = 1.4276e-01  (0.000s)
20   R6: err = 2.4541e+00, μ = 1.4925e-01  (0.000s)
21   R7: err = 2.3916e+00, μ = 1.6181e-01  (0.000s)
22   R8: err = 2.3137e+00, μ = 1.6914e-01  (0.000s)
23   R9: err = 2.2264e+00, μ = 1.7976e-01  (0.000s)
24  R10: err = 2.1733e+00, μ = 1.9694e-01  (0.000s)
25  R11: err = 2.0897e+00, μ = 2.0357e-01  (0.001s)
26  R12: err = 1.9970e+00, μ = 2.1885e-01  (0.000s)
27  R13: err = 1.9492e+00, μ = 2.4474e-01  (0.000s)
28  R14: err = 1.8725e+00, μ = 2.5285e-01  (0.000s)
29  R15: err = 1.7619e+00, μ = 2.6838e-01  (0.000s)
30  R16: err = 1.7039e+00, μ = 3.1151e-01  (0.000s)
```

```
31  R17: err = 1.6522e+00, μ = 3.3399e-01  (0.000s)
32  R18: err = 1.5440e+00, μ = 3.4236e-01  (0.000s)
33  R19: err = 1.4254e+00, μ = 3.8725e-01  (0.000s)
34  R20: err = 1.3839e+00, μ = 4.7781e-01  (0.000s)
35  R21: err = 1.3287e+00, μ = 5.0136e-01  (0.000s)
36  R22: err = 1.2029e+00, μ = 5.1278e-01  (0.000s)
37  R23: err = 1.0393e+00, μ = 5.9712e-01  (0.000s)
38  R24: err = 9.8325e-01, μ = 8.7588e-01  (0.001s)
39  R25: err = 9.4944e-01, μ = 9.9881e-01  (0.000s)
40  R26: err = 8.5587e-01, μ = 1.0000e+00  (0.000s)
41  R27: err = 6.2693e-01, μ = 9.9998e-01  (0.000s)
42  R28: err = 2.4643e-01, μ = 1.0001e+00  (0.000s)
43  R29: err = 1.4969e-02, μ = 1.0002e+00  (0.000s)
44  R30: err = 1.5726e-04, μ = 4.6887e+01  (0.000s)
45  R31: err = 2.2343e-04, μ = 5.7455e+07  (0.000s)
46  WARNING: diverged, R30 is the best we can do
47  inversion halted in 31 iterations after 0m0.002s
48  WARNING: failed to converge to error < 1e-05 within 2147483647 ms
49  shutting down cuBLAS
50  shutting down kernels
```

Listing B.8: Doomed attempt to invert a 2×2 singular matrix of all ones.

```
 1  > ./acmi mats/1111.mtx
 2  7998.438 MiB device memory available
 3  loading mats/1111.mtx
 4  matrix is 2x2 and 0.000 MiB in size
 5  0.000 MiB/matrix; allocating 0.000 MiB total
 6  setting up cuBLAS
 7  setting up kernels on GeForce GTX 1080
 8  blocks/matrix: 1, threads/matrix: 4
 9  blocks/vector: 1, threads/vector: 2
10  uploading matrix to device
11  inverting mats/1111.mtx with cubic convergence...
12  initializing work matrices...
13  computed alpha = 0.5
14   R0: err = 1.0000e+00, μ = 0.0000e+00  (0.000s)
15   R1: err = 1.0000e+00, μ = 1.0000e+00  (0.000s)
16  diverged, retrying with alternate R0...
17   R0: err = 1.0000e+00, μ = 0.0000e+00  (0.000s)
18   R1: err = 1.0000e+00, μ = 1.0000e+00  (0.000s)
19  diverging, extending to double precision...
20  backing up to reduce error
21   R0: err = 1.0000e+00, μ = 0.0000e+00  (0.000s)
22   R1: err = 1.0000e+00, μ = 1.0000e+00  (0.000s)
23   R2: err = 1.0000e+00, μ = 1.0000e+00  (0.000s)
24  WARNING: diverged, R1 is the best we can do
25  inversion halted in 4 iterations after 0m0.000s
26  WARNING: failed to converge to error < 1e-05 within 2147483647 ms
27  shutting down cuBLAS
28  shutting down kernels
```

APPENDIX C

SELECTED SOURCE CODE

For the complete source code, please refer to: https://github.com/zauberkraut/acmi

Listing C.1: Altman Inversion Implementation

```
1  /* invert.c
2
3     ACMI convergent inversion algorithm implementation. */
4
5  #include <time.h>
6  #include "acmi.h"
7
8  enum { MAX_RESTART_ITER = 2 };
9
10 /* Quickly computes ||I - AR0|| for R0 = alpha*I. */
11 static double traceErr(double alpha, Mat mA) {
12   return sqrt(MatN(mA) + 1 - 2*alpha*trace(mA));
13 }
14
15 /* Returns the number of milliseconds elapsed since start. */
16 static int msSince(struct timespec* start) {
17   struct timespec time;
18   clock_gettime(CLOCK_MONOTONIC, &time);
19   return (time.tv_sec - start->tv_sec)*1000 +
20     (time.tv_nsec - start->tv_nsec)/1.e6;
21 }
22
23 /* Swaps matrix pointers. */
24 static void swap(Mat* mp, Mat* np) {
25   Mat t = *mp;
26   *mp = *np;
27   *np = t;
28 }
29
30 /* The inversion algorithm.
31
32    If convRateLimit < 0, |convRateLimit| specifies a multiple of the
33    first measured rate to use as the limit. */
34 double altmanInvert(const Mat mA, Mat* const mRp,
35                     const int convOrder, const double errLimit,
36                     const int msLimit, double convRateLimit,
37                     bool safeR0) {
38   static struct timespec start;
39
40   debug("initializing work matrices...");
41   clock_gettime(CLOCK_MONOTONIC, &start); // start clock
42   const int matCount = convOrder < 4 ? 4 : 5;
43   Mat mR = MatBuild(mA), // allocate matrices with A's dimensions
44       mAR = MatBuild(mA),
45       mX = MatBuild(mA),
46       mY = matCount < 5 ? 0 : MatBuild(mA);
47
48   const double alpha = 1/nrm2(mA);
49   debug("computed alpha = %g", alpha);
50   double err = NAN;
51
52   if (safeR0) {
53     debug("starting with safe R0");
54     transpose(alpha*alpha, mA, mR);
```

```
55    } else {
56      MatClear(mR);
57      addId(alpha, mR);
58      err = traceErr(alpha, mA);
59    }
60
61    int iter = 0, totalIters = 0, ms; // while time remains
62    for (; (ms = msSince(&start)) < msLimit; iter++, totalIters++) {
63      static double prevErr = INFINITY;
64      static int prevMS = 0;
65
66      gemm(1, mA, mR, 0, mAR); // mAR <- AR
67      if (iter || safeR0) {     // don't overwrite above optimization
68        err = minusIdNrm2(mAR);
69      }
70      // rate of convergence
71      const double convRate = fabs(err)/pow(fabs(prevErr), convOrder);
72
73      debug("%*sR%d: err = %.4e, μ = %.4e  (%.3fs)", iter < 10, "",
74            iter, err, convRate, (ms - prevMS)/1000.);
75      prevMS = ms;
76
77      if (err <= errLimit) { // we've achieved the desired accuracy
78        break;
79      }
80
81      // handle divergence
82      if (!safeR0 && err >= prevErr && iter <= MAX_RESTART_ITER) {
83        /* Our attempt to exploit the self-adjoint, positive-definite
84           R0 = alpha*I failed. Start over using the slow, safe
85           R0 = alpha^2 * A^T instead. */
86        debug("diverged, retrying with alternate R0...");
87
88        safeR0 = true;
89        transpose(alpha*alpha, mA, mR);
90        prevErr = INFINITY;
91        iter = -1; totalIters--;
92
93        continue;
94      } else if (MatElemSize(mA) < MAX_ELEM_SIZE && convRateLimit > 0
95                 && (convRate >= convRateLimit || err >= prevErr)) {
96        debug("diverging, extending to double precision...");
97
98        if (MatDev(mA) && // quit if we lack GPU memory for promotion
99            !checkDevMemEnough(MatN(mA), MatElemSize(mA), matCount)) {
100         debug("not enough device RAM; halting");
101         if (err >= prevErr) { // back up if last iter were better
102           swap(&mX, &mR);
103         }
104         break;
105       }
106
107       MatPromote(mA); MatPromote(mR); MatPromote(mAR);
108       MatPromote(mX);
109       if (mY) {
110         MatPromote(mY);
111       }
112
113       double tmp = prevErr;
114       prevErr = INFINITY; // in case error shall reinflate
115       if (err >= tmp) {   // if we've fully diverged...
116         debug("backing up to reduce error");
117         swap(&mX, &mR);
118         iter -= 2; totalIters--;
119         continue; // recompute AR
120       }
```

```
121       } else if (err >= prevErr || !isfinite(err)) {
122         warn("diverged, R%d is the best we can do", iter - 1);
123 
124         swap(&mX, &mR); // back up R to its previous value
125         err = prevErr;
126         break;          // quit
127       } else {
128         if (iter == 1 && convRateLimit < 0) {
129           // set rate limit to initial value
130           convRateLimit *= -convRate;
131         }
132 
133         prevErr = err;
134       }
135 
136       switch (convOrder) { // compute the next iteration
137         case 2: // quadratic convergence
138           gemm(1, mR, mAR, 0, mX);  // mX  <- RAR
139           geam(-1, mX, 2, mR, mAR); // mAR <- 2R - RAR = next R
140           swap(&mR, &mAR);          // mR  <- next R, mAR <- prev R
141           swap(&mX, &mAR);          // mX  <- previous R, mAR <- junk
142           break;
143         case 3: // cubic convergence
144           gemm(1, mAR, mAR, 0, mX); // mX  <- (AR)^2
145           geam(-3, mAR, 1, mX, mX); // mX  <- -3AR + (AR)^2
146           addId(3, mX);             // mX  <- 3I - 3AR + (AR)^2
147           gemm(1, mR, mX, 0, mAR);  // mAR <- R(3I - 3AR + (AR)^2)
148           swap(&mR, &mAR);          // mR  <- next R, mAR <- prev R
149           swap(&mX, &mAR);          // mX  <- previous R
150           break;
151         case 4: // quartic convergence
152           gemm(1, mAR, mAR, 0, mX); // mX <- (AR)^2
153           geam(-4, mAR, 1, mX, mX); // mX <- -4AR + (AR)^2
154           addId(6, mX);             // mX <- 6I - 4AR + (AR)^2
155           gemm(-1, mAR, mX, 0, mY); // mY <- -AR(6I - 4AR + (AR)^2)
156           addId(4, mY);             // mY <- 4I - AR(6I-4AR+(AR)^2)
157           swap(&mR, &mX);           // mX <- previous R
158           gemm(1, mX, mY, 0, mR);   // mR <- R(4I - AR(6I-4AR+(AR)^2))
159           break;
160         default:
161           fatal("unsupported convergence order: %d", convOrder);
162       }
163     } // end while
164 
165     ms = msSince(&start);
166     int minutes = ms/60000;
167     double seconds = (ms - minutes*60000)/1000.;
168     debug("inversion halted in %d iterations after %dm%.3fs",
169           totalIters, minutes, seconds);
170     if (err > errLimit) {
171       warn("failed to converge to error < %g within %d ms", errLimit,
172            msLimit);
173     }
174 
175     // cleanup
176     MatFree(mAR);
177     MatFree(mX);
178     if (mY) {
179       MatFree(mY);
180     }
181     *mRp = mR; // return inverted matrix
182     return err;
183   }
```

Listing C.2: Linear Algebraic Function Wrappers

```
1  /* linalg.c
2
3     ACMI linear algebraic operations implemented using (cu)BLAS and
4     CUDA kernels. */
5
6  #include <cublas_v2.h>
7  #include <openblas/cblas.h>
8  #include "acmi.h"
9
10 static cublasHandle_t g_cublasHandle;
11
12 void gpuSetUp(const int maxBlocksPerKernel, const int n) {
13   debug("setting up cuBLAS");
14   if (cublasCreate(&g_cublasHandle) != CUBLAS_STATUS_SUCCESS) {
15     fatal("couldn't open cuBLAS handle");
16   }
17   cuSetUp(maxBlocksPerKernel, n);
18 }
19
20 void gpuShutDown() {
21   debug("shutting down cuBLAS");
22   if (g_cublasHandle) cublasDestroy(g_cublasHandle);
23   cuShutDown();
24 }
25
26 /* mT <- alpha*mA^T */
27 void transpose(double alpha, Mat mA, Mat mT) {
28   const int n = MatN(mA);
29   const void* const a = MatElems(mA);
30   void* const t = MatElems(mT);
31   const bool dev = MatDev(mA);
32   const double beta = 0;
33
34   switch (MatElemSize(mA)) {
35   case 4:
36     if (dev) {
37       float alpha32 = alpha;
38       cublasSgeam(g_cublasHandle, CUBLAS_OP_T, CUBLAS_OP_N, n, n,
39                   &alpha32, a, n, (float*)&beta, t, n, t, n);
40     } else {
41       cblas_somatcopy(CblasColMajor, CblasTrans, n, n, alpha, a, n,
42                       t, n);
43     }
44     break;
45
46   case 8:
47     if (dev) {
48       cublasDgeam(g_cublasHandle, CUBLAS_OP_T, CUBLAS_OP_N, n, n,
49                   &alpha, a, n, &beta, t, n, t, n);
50     } else {
51       cblas_domatcopy(CblasColMajor, CblasTrans, n, n, alpha, a, n,
52                       t, n);
53     }
54     break;
55   }
56 }
57
58 /* C <- alpha*A*B + beta*C */
59 void gemm(double alpha, Mat mA, Mat mB, double beta, Mat mC) {
60   const int n = MatN(mA);
61   const void* const a = MatElems(mA);
62   const void* const b = MatElems(mB);
63   void* const c = MatElems(mC);
64   const bool dev = MatDev(mA);
65
```

```
66    switch (MatElemSize(mA)) {
67    case 4:
68      if (dev) {
69        float alpha32 = alpha, beta32 = beta;
70        cublasSgemm(g_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N, n, n, n,
71                    &alpha32, a, n, b, n, &beta32, c, n);
72      } else {
73        cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, n, n,
74                    n, alpha, a, n, b, n, beta, c, n);
75      }
76      break;
77
78    case 8:
79      if (dev) {
80        cublasDgemm(g_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N, n, n, n,
81                    &alpha, a, n, b, n, &beta, c, n);
82      } else {
83        cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, n, n,
84                    n, alpha, a, n, b, n, beta, c, n);
85      }
86      break;
87    }
88  }
89
90  /* mB = mC    => C <- alpha*A + beta*C
91     otherwise     C <- alpha*A + beta*B */
92  void geam(double alpha, Mat mA, double beta, Mat mB, Mat mC) {
93    const int n = MatN(mA);
94    const int n2 = MatN2(mA);
95    const void* const a = MatElems(mA);
96    const void* const b = MatElems(mB);
97    void* const c = MatElems(mC);
98    const bool dev = MatDev(mA);
99
100   switch (MatElemSize(mA)) {
101   case 4:
102     if (dev) {
103       float alpha32 = alpha, beta32 = beta;
104       cublasSgeam(g_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N, n, n,
105                   &alpha32, a, n, &beta32, b, n, c, n);
106     } else {
107       if (b == c) {
108         cblas_sscal(n2, beta, c, 1);
109       } else {
110         memset(c, 0, MatSize(mC));
111         cblas_saxpy(n2, beta, b, 1, c, 1);
112       }
113       cblas_saxpy(n2, alpha, a, 1, c, 1);
114     }
115     break;
116
117   case 8:
118     if (dev) {
119       cublasDgeam(g_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N, n, n,
120                   &alpha, a, n, &beta, b, n, c, n);
121     } else {
122       if (b == c) {
123         cblas_dscal(n2, beta, c, 1);
124       } else {
125         memset(c, 0, MatSize(mC));
126         cblas_daxpy(n2, beta, b, 1, c, 1);
127       }
128       cblas_daxpy(n2, alpha, a, 1, c, 1);
129     }
130     break;
131   }
```

```
132 }
133
134 /* mA <- mA + alpha*I */
135 void addId(double alpha, Mat mA) {
136   if (MatDev(mA)) {
137     cuAddId(alpha, MatElems(mA), MatN(mA), MatElemSize(mA));
138   } else for (int diag = 0; diag < MatN(mA); diag++) {
139     /* This could be marginally sped up using *axpy with a 1xn
140        vector of ones and a stride of n + 1 over the matrix, but
141        it's not worth the trouble. */
142     MatPut(mA, diag, diag, alpha + MatGet(mA, diag, diag));
143   }
144 }
145
146 /* Computes the Frobenius norm of a matrix. */
147 double nrm2(Mat mA) {
148   const int n2 = MatN2(mA);
149   const void* a = MatElems(mA);
150   const bool dev = MatDev(mA);
151   double norm;
152
153   switch (MatElemSize(mA)) {
154   case 4:
155     if (dev) {
156       float norm32;
157       cublasSnrm2(g_cublasHandle, n2, a, 1, (float*)&norm32);
158       norm = norm32;
159     } else {
160       norm = cblas_snrm2(n2, a, 1);
161     }
162     break;
163
164   case 8:
165     if (dev) {
166       cublasDnrm2(g_cublasHandle, n2, a, 1, (double*)&norm);
167     } else {
168       norm = cblas_dnrm2(n2, a, 1);
169     }
170     break;
171   }
172
173   return norm;
174 }
175
176 /* Computes the sum of the entries on the main diagonal. */
177 double trace(Mat mA) {
178   double trace;
179
180   if (MatDev(mA)) {
181     trace = cuTrace(MatElems(mA), MatN(mA), MatElemSize(mA));
182   } else {
183     trace = 0.;
184     for (int i = 0; i < MatN(mA); i++) {
185       trace += MatGet(mA, i, i);
186     }
187   }
188
189   return trace;
190 }
191
192 /* Computes the norm of (I - A). */
193 double minusIdNrm2(Mat mA) {
194   addId(-1, mA); // sort of a hack, but it works very well
195   double norm = nrm2(mA);
196   addId(1, mA);
197   return norm;
```

```
198 }
```

Listing C.3: Handwritten CUDA Kernels

```
1  /* kernels.cu
2
3     Custom ACMI CUDA kernels. */
4
5  #include "acmi.h"
6
7  namespace {
8
9  enum { SWEEP_FACTOR = 16 };
10
11 /* Kernel parameters. */
12 int g_blocksPerVector, g_threadsPerVectorBlock,
13     g_blocksPerSweep,  g_threadsPerSweepBlock,
14     g_blocksPerMatrix, g_threadsPerMatrixBlock;
15 double* g_buckets;
16 int g_bucketsLen;
17
18 /* Copies elements from one nxn matrix to another, converting them
19    to the precision of the destination matrix. */
20 template<typename T, typename U> __global__ void
21 kernCopy(T* dst, const U* src, const int64_t n2) {
22   const int offset = blockIdx.x*blockDim.x + threadIdx.x;
23   const int stride = gridDim.x*blockDim.x;
24   const T* const end = dst + n2;
25   src += offset;
26   dst += offset;
27
28   for (; dst < end; dst += stride, src += stride) {
29     *dst = *src;
30   }
31 }
32
33 template<typename T> __global__ void
34 kernAddId(const T alpha, T* a, const int n) {
35   const T* const end = a + n*n;
36   a += (blockIdx.x*blockDim.x + threadIdx.x)*(n + 1);
37   const int stride = gridDim.x*blockDim.x*(n + 1);
38
39   for (; a < end; a += stride) {
40     *a += alpha;
41   }
42 }
43
44 /* Sweeps sums of matrix elements into buckets. */
45 template<typename T> __global__ void
46 kernSweepSums(const T* a, const int64_t len, const int pitch,
47               T* const buckets, const int bucketsLen) {
48   const int offset = blockIdx.x*blockDim.x + threadIdx.x;
49   const T* const end = a + len;
50   a += offset*pitch;
51   const int stride = gridDim.x*blockDim.x*pitch;
52
53   if (offset < bucketsLen) {
54     T partialSum = 0.;
55     for (; a < end; a += stride) {
56       partialSum += *a;
57     }
58     buckets[offset] = partialSum;
59   }
60 }
61
62 } // end anonymous namespace
```

```
63
64  extern "C" {
65
66  /* Sets up kernel parameters. */
67  void cuSetUp(const int maxBlocksPerKernel, const int n) {
68    cudaDeviceProp prop;
69    cudaGetDeviceProperties(&prop, 0); // assume using first device
70    debug("setting up kernels on %s", prop.name);
71    if (maxBlocksPerKernel > prop.maxGridSize[0]) {
72      fatal("max blocks supported: %d", prop.maxGridSize[0]);
73    }
74    const int maxThreadsPerBlock = prop.maxThreadsPerBlock;
75
76    const int64_t n2 = n*n;
77    g_bucketsLen = (n + SWEEP_FACTOR - 1)/SWEEP_FACTOR;
78    g_buckets = (double*)cuMalloc(sizeof(double)*g_bucketsLen);
79
80    g_blocksPerVector = (n  + maxThreadsPerBlock - 1) /
81                        maxThreadsPerBlock;
82    g_blocksPerVector = iMin(maxBlocksPerKernel, g_blocksPerVector);
83    g_blocksPerSweep  = (g_bucketsLen  + maxThreadsPerBlock - 1) /
84                        maxThreadsPerBlock;
85    g_blocksPerSweep  = iMin(maxBlocksPerKernel, g_blocksPerSweep);
86    g_blocksPerMatrix = (n2 + maxThreadsPerBlock - 1) /
87                        maxThreadsPerBlock;
88    g_blocksPerMatrix = iMin(maxBlocksPerKernel, g_blocksPerMatrix);
89    g_threadsPerVectorBlock = iMin(maxThreadsPerBlock, n);
90    g_threadsPerSweepBlock  = iMin(maxThreadsPerBlock, g_bucketsLen);
91    g_threadsPerMatrixBlock = iMin(maxThreadsPerBlock, n2);
92
93    auto threadsPerVector = g_blocksPerVector*g_threadsPerVectorBlock,
94         threadsPerMatrix = g_blocksPerMatrix*g_threadsPerMatrixBlock;
95    debug("blocks/matrix: %d, threads/matrix: %d\n"
96          "blocks/vector: %d, threads/vector: %d", g_blocksPerMatrix,
97          threadsPerMatrix, g_blocksPerVector, threadsPerVector);
98  }
99
100 /* Cleans up kernel environment. */
101 void cuShutDown() {
102   debug("shutting down kernels");
103   cuFree(g_buckets);
104 }
105
106 /* Doubles matrix precision. */
107 void cuPromote(void* dst, void* src, int srcElemSize, int64_t n2) {
108   kernCopy<<<g_blocksPerMatrix, g_threadsPerMatrixBlock>>>
109     ((double*)dst, (const float*)src, n2);
110 }
111
112 /* Adds alpha*I to the matrix backed by the device array elems. */
113 void cuAddId(double alpha, void* elems, int n, int elemSize) {
114   switch (elemSize) {
115   case 4:
116     kernAddId<<<g_blocksPerVector, g_threadsPerVectorBlock>>>
117       ((float)alpha, (float*)elems, n);
118     break;
119   case 8:
120     kernAddId<<<g_blocksPerVector, g_threadsPerVectorBlock>>>
121       (alpha, (double*)elems, n);
122     break;
123   }
124 }
125
126 /* Computes a matrix trace by sweeping sums. */
127 double cuTrace(void* elems, int n, int elemSize) {
128   const int64_t n2 = n*n;
```

```
129    double trace = NAN;
130
131    switch (elemSize) {
132      float trace32;
133
134    case 4:
135      kernSweepSums<<<g_blocksPerSweep, g_threadsPerSweepBlock>>>
136        ((float*)elems, n2, n + 1, (float*)g_buckets, g_bucketsLen);
137      kernSweepSums<<<1, 1>>>
138        ((float*)g_buckets, g_bucketsLen, 1, (float*)g_buckets, 1);
139      cuDownload(&trace32, g_buckets, sizeof(float));
140      trace = trace32;
141      break;
142    case 8:
143      kernSweepSums<<<g_blocksPerSweep, g_threadsPerSweepBlock>>>
144        ((double*)elems, n2, n + 1, (double*)g_buckets, g_bucketsLen);
145      kernSweepSums<<<1, 1>>>
146        ((double*)g_buckets, g_bucketsLen, 1, (double*)g_buckets, 1);
147      cuDownload(&trace, g_buckets, sizeof(double));
148      break;
149    }
150
151    return trace;
152  }
153
154  } // end extern "C"
```

Listing C.4: Random Matrix Generator

```
 1  /* Tests for CPU support of the RDRAND instruction. */
 2  static bool rdRandSupported() {
 3    unsigned eax, ebx, ecx, edx;
 4    return __get_cpuid(1, &eax, &ebx, &ecx, &edx) && ecx & bit_RDRND;
 5  }
 6
 7  static int cstdRand16() {
 8    return (int)((unsigned)rand() >> 15); // discard correlated bits
 9  }
10
11  /* Uses RDRAND instruction to generate high-quality random integers.
12     Requires an Ivy Bridge or newer x86 CPU. Requires no seeding. */
13  static int rdRand16() {
14    static int n = 0;
15    static unsigned long long r = 0;
16
17    if (!n) {
18      /* RDRAND always pulls 64 bits, so splitting each term into four
19         16-bit numbers is far faster. */
20      _rdrand64_step(&r); // assume entropy suffices
21      n = 4;
22    }
23
24    int s = (int)(r & 0xffff);
25    r >>= 16;
26    n--;
27
28    return s;
29  }
30
31  /* Draws off a bit from a random number and returns it as a sign. */
32  static int drawSign(int* n) {
33    int tmp = *n;
34    *n >>= 1;
35    return (tmp & 1) ? 1 : -1;
36  }
37
```

```c
38 /* Generates a random, probably-invertible matrix of integers.
39    (Certainly-invertible if diagonally-dominant). Allowing negative
40    entries shall probably cause ill-conditioning.
41    TODO: replace bool cascade with binary flags
42    TODO: large dominant diagonals might swallow bias of +1
43    TODO: populating elements in column-major order _might_ be faster
44    TODO: 16-bit granularity of reals can be course */
45 Mat MatNewRand(int n, int elemSize, double maxElem, bool symm,
46                bool real, bool neg, bool diagDom,
47                bool useHardwareRNG) {
48   if (useHardwareRNG && !rdRandSupported()) {
49     fatal("your CPU doesn't support the RDRAND instruction");
50   }
51   int (*rand16)() = useHardwareRNG ? rdRand16 : cstdRand16;
52   int rMax = neg ? SHRT_MAX : USHRT_MAX;
53
54   Mat m = MatNew(n, elemSize, false);
55   int maxElemEx = floor(maxElem) + 1;
56
57   for (int row = 0; row < n; row++) {
58     int col = symm ? row : 0;
59     double rowSum = 0;
60     // if symmetric, sum the elements before this diagonal
61     for (int i = 0; i < col; i++) {
62       rowSum += MatGet(m, row, i);
63     }
64
65     for (; col < n; col++) {
66       if (!diagDom || row != col) { // diagonals are summed later
67         int r = rand16();
68         double sign = neg ? drawSign(&r) : 1;
69         double absElem = real ? (double)r/rMax * maxElem :
70                                 (double)(r % maxElemEx);
71         double elem = sign*absElem;
72         MatPut(m, row, col, elem);
73         if (symm && row != col) { // mirror symmetric element
74           MatPut(m, col, row, elem);
75         }
76         rowSum += absElem;
77       }
78     }
79
80     if (diagDom) {
81       double sign = neg ? (rand16() & 1 ? 1 : -1) : 1;
82       /* Make diagonal element strictly greater than the sum of the
83          other row entries. */
84       double diag = sign * (real ? nextafter(rowSum, INFINITY) :
85                                    rowSum + 1);
86       MatPut(m, row, row, diag);
87     }
88   }
89
90   return m;
91 }
```