TORCH Computational Reference Kernels

A Testbed for Computer Science Research

Alex Kaiser, Samuel Williams, Kamesh Madduri, Khaled Ibrahim,
David H. Bailey, James W. Demmel, Erich Strohmaier

Computational Research Division
Lawrence Berkeley National Laboratory Berkeley, CA 94720, USA

Computer Science Division and Mathematics Department,
University of California, Berkeley, CA 94720, USA

December 2, 2010

Contents

Abstract

For decades, computer scientists have sought guidance on how to evolve architectures, languages, and programming models in order to improve application performance, efficiency, and productivity. Unfortunately, without overarching advice about future directions in these areas, individual guidance is inferred from the existing software/hardware ecosystem, and each discipline often conducts their research independently assuming all other technologies remain fixed. In today's rapidly evolving world of on-chip parallelism, isolated and iterative improvements to performance may miss superior solutions in the same way gradient descent optimization techniques may get stuck in local minima. To combat this, we present **TORCH**: A Testbed for Optimization ResearCH. These computational reference kernels define the core problems of interest in scientific computing without mandating a specific language, algorithm, programming model, or implementation. To compliment the kernel (problem) definitions, we provide a set of algorithmically-expressed verification tests that can be used to verify a hardware/software co-designed solution produces an acceptable answer. Finally, to provide some illumination as to how researchers have implemented solutions to these problems in the past, we provide a set of reference implementations in C and MATLAB.

Chapter 1

Introduction

For decades, computer scientists have sought guidance on how to evolve architectures, languages, and programming models in order to improve application performance, efficiency, and productivity. Unfortunately, without overarching advice about future directions in these areas, individual guidance is inferred from the existing software/hardware ecosystem, and each discipline often conducts their research independently assuming all other technologies remain fixed. Architects attempt to provide micro-architectural solutions to improve performance on fixed binaries. Researchers tweak compilers to improve code generation for existing architectures and implementations, and they may invent new programming models for fixed processor and memory architectures and computational algorithms. In today's rapidly evolving world of on-chip parallelism, these isolated and iterative improvements to performance may miss superior solutions in the same way gradient descent optimization techniques may get stuck in local minima.

To combat this tunnel vision, previous work set forth a broad categorization of numerical methods of interest to the scientific computing community (the seven *Dwarfs*) and subsequently for the larger parallel computing community in general (13 *motifs*), suggesting that these were the problems of interest that researchers should focus on [8, 9, 42]. Unfortunately, such broad brush strokes often miss the nuances seen in individual kernels that may be similarly categorized. For example, the computational requirements of particle methods vary greatly between the naive but more accurate direct calculations and the particle-mesh and particle-tree codes.

In this report, we present an alternate methodology for testbed creation. For simplicity we restricted our domain to scientific computing. The result, **TORCH**: A Testbed for Optimization ResearCH, defines a broad set of computational problems, verification, and example implementations. Superficially, TORCH is reminiscent of the computational kernels in Intel's RMS work [53]. However, we proceed in a more regimented effort. In this introductory chapter, we commence by defining the key components of our testbed, and proceed by enumerating the kernels within our testbed. By no means is the list of problems complete. Rather, it constitutes a sufficiently broad yet tractable set for initial investigation. The remaining chapters of this report provide a prose description of the kernels in the accompanying source distribution as well as an introduction to each motif.

1.1 An Evolved Testbed

For a testbed to have long-term and far-reaching value, it must be free and agnostic of existing software, hardware, and algorithms developed and tuned for them. Today, the underlying semantics of memory and instruction set architecture leech through into benchmarks and limit the ability for researchers to engage in truly novel directions. Languages and programming models should not expose the details of an architectural implementation to programmers, but rather allow the most natural expression of an algorithm. When operating on shared vectors, matrices, or grids, the dogmatic load-store random access memory semantics may be very natural and efficient. However, when operating on shared sets, queues, graphs, and trees, programmers are often forced to create their own representations built on an underlying linear random access memory using loads, stores, and semaphores. This should not be.

To ensure we did not fall prey to the tunnel vision optimization problem, we made several mandates on our evolved testbed. To that end, we strived to stay away from the conventional wisdom that suggests that parallelization and optimization of an existing software implementation is the challenging problem to be solved.

Benchmark Style	Microarchitecture	Compilers	Instruction Set Arch.	Prog. Models	Languages	Memory Architecture
Fixed Binary	✓					
Fixed Source Code	✓	✓	✓			
Fixed Interface, but may optimize code	✓	✓	✓			
Code-based problem definition	✓	✓	✓	✓		
High-level problem definition	✓	✓	✓	✓	✓	✓

Table 1.1: Fields of research enabled by different styles of benchmarks

Rather we believe the starting point is not code, but a problem definition expressed in the most natural language for its field. The Sort benchmark collection [94] initiated by Gray exemplifies the future vision for our reference kernel testbed. The sort benchmark definitions are based on a well-defined problem, include a scalable input generator, multiple metrics for assessing quality (for instance, sort rate for a terabyte-sized dataset, amount of data that can be sorted in a minute or less, records sorted per joule, and so on), and finally a verification scheme. Framing the benchmark objectives as an open challenge, rather than providing an optimized implementation of a particular approach, has led to novel algorithmic research and innovative engineered sort routines.

We believe similar results can be attained in other fields. For example, consider Table 1.1. When testbeds are based on a fixed binary, divorced of the original representation or mathematical problem, very little can be done to improve performance. In fact, similar scenarios drove rapid advances in microarchitecture, but led to a (energy inefficient) local minimum as they alone could never make the jump to multicore architectures. As one proceeds to more and more abstract representations of a problem, more and more researchers can be included in the co-design research effort. As such, we believe any and all of the listed fields of research working on parallel computing could benefit from the initial testbed enumerated in the report.

Although this argument may sound vague, we found the textbook taxonomy to describe problems illustrative. The *"solution"* is the efficient co-design of software and hardware to implement a *"problem"* described in a domain-specific mathematical language (*e.g.* numerical linear algebra, particle physics, spectral analysis, sorting, etc.). The veracity of the solution is determined via an accompanying verification methodology specified in the same domain-specific mathematical language. We may provide *"hints"* to the solution in the form of reference and optimized implementations using existing languages, programming models, or hardware. The quality of the solution is based on the performance, energy, cost (amortized by reuse), and designer productivity.

In the following sections, we will describe and illustrate this process of problem definition, scalable input creation, verification, and implementation of reference codes for the scientific computing domain. Table 1.3 enumerates and describes the level of support we've developed for each kernel. We group these important kernels using the Berkeley Dwarfs/Motifs taxonomy using a red box in the appropriate column. As kernels become progressively complex, they build upon other, simpler computational methods. We note this dependency via orange boxes. We must reiterate that by no means is our list comprehensive. For example, the finite difference methods listed in the structured grid section are easily understood and representative, but are often replaced by more complex methods (*e.g.* the finite volume and lattice Boltzmann methods) and solver acceleration techniques (multigrid, adaptive mesh refinement).

1.1.1 Problem Specification

We create a domain-appropriate high-level definition for each important kernel. To ensure future endeavors are not tainted by existing implementations, we specified the problem definition to be independent of both computer architecture and existing programming languages, models, and data types.

For example, numerical linear algebra has a well developed lexicon of operands (scalars, vectors, matrices,

etc.) and operators (addition, multiplication, transpose, inverse, etc.). Although programmers are now accustomed to mapping such structures to the array-like data structures arising from the linear random access memories in computer architecture, such an end state is the product of decades of focused optimization of hardware and software. It is not an inherent characteristic or mandate in the problem definition.

Conversely, graph algorithms are often defined as operating on edges and vertices via set and queue operations. Programmers are often forced to map such operands and operators onto architectures optimized for linear algebra. Although such techniques have sufficed in the single-core era, parallelization of set and queue operations on shared random access memories via kludges like atomic operations is unnatural and error prone. By taking a step back to a high-level problem definition, we hope designers may free themselves of their tunnel vision and build truly novel systems adept at such computations.

Whenever possible, we specify the high-level parallel operations (for all, sum, etc.) to be independent of whether or not such constructs will create data dependencies when mapped to existing languages or instruction set architectures. This ensures we neither restrict nor recast parallelism. Moreover, we minimize the expression of user-managed synchronization in the problem specification.

1.1.2 Scalability

The last decade has not only seen an order-of-magnitude increase in inter-node parallelism, but also a more challenging explosion in intra-node parallelism via SIMD, hardware multithreading, and multiple cores. This ever increasing parallelism constrains the fixed problem size benchmarks into a strong scaling regime. Although this might be appropriate for some domains, it is rarely appropriate in the field of scientific computing where weak scaling has been used to solve problems at petascale. Similarly, such a constraint may drive designers to solutions that, although they may be quite appropriate for the benchmark, are utterly unscalable and inappropriate for future problem sizes.

To that end, for each kernel, we have created a scalable problem generator. In some cases this generator may be nothing more than a means of specifying problems using the underlying method's high-level description language. In other cases, code is written to create input datasets. In either case, the problem size is independent of implementation or mapping to architecture.

In the linear algebra world, inspired by the Linear Algebra working note [50], we generate randomized matrices for LU and QR on the fly. We apply similar techniques in the spectral problems by specifying the FFT input size, but randomizing initial values. The n-Body computations can be scaled simply by increasing the number of particles, and the computational challenges seen in the particle-tree codes can be altered by changing the spatial layout of particles and forces used.

Unfortunately, scalable problem generation can be challenging for kernels in which the problem configuration or connectivity is specified in the data structure. Often, sparse linear algebra research has been focused on fixed size matrices that have resided in collections for decades. We believe this tradition must be evolved into a form like [60] so that repositories of scalable matrix generators exist. We acknowledge that this is not universally applicable as some matrices are constructed from discrete real-world phenomena like connectivity of the web. The sparse linear algebra chapter discusses this issue in more detail.

1.1.3 Solution Verification

One could view the high level problem definition as a means to verify the validity of a solution. In effect, one can compare the results from two implementations (reference and hardware/software co-designed), checking for discrepancies on a bit by bit granularity. Unfortunately, such an approach may not always be appropriate. As problem size scales, execution of the reference implementation may not be feasible. Moreover, such a verification regime assumes that implementation and verification code don't contain a common error (matching, but incorrect result). Finally, this approach assumes that there is one true solution. We wish to create a verification methodology that is in some sense orthogonal to the problem definition.

In many cases, we construct problems whose solutions are known a priori or can be calculated with minimal cost. We verify the symmetric eigensolver by constructing randomized matrices with known eigenvalues. To obtain such a matrix, one forms a diagonal matrix D composed of the desired eigenvalues and a randomized orthogonal matrix Q. The test matrix is the product $Q^T D Q$. This 'reverse diagonalization' produces a randomized matrix with pre-determined eigenvalues, the eigenvalues of which can be selected to be as numerically

Complexity of Solving Poisson's equation on an $n \times n \times n$ mesh with $N = n^3$ unknowns					
Method	**Direct or Iterative**	**Serial Complexity**	**Serial Memory**	**Dwarf**	**Section in this report**
Dense Cholesky	Direct	N^3	N^2	Dense LA	2.13
Band Cholesky	Direct	$N^{7/3}$	$N^{5/3}$	Band LA	—
Jacobi	Iterative	$N^{5/3}$	N	Structured Grids	5.8
Gauss-Seidel	Iterative	$N^{5/3}$	N	Structured Grids	—
Conj. Gradients	Iterative	$N^{4/3}$	N	Structured Grids	5.9
Red/Black SOR	Iterative	$N^{4/3}$	N	Structured Grids	—
Sparse Cholesky	Direct	N^2	$N^{4/3}$	Sparse LA	—
FFT	Direct	$N \log N$	N	Spectral Methods	7.5
Multigrid	Iterative	N	N	Structured Grids	5.10
Lower Bound	—	N	N	—	—

Table 1.2: Complexity of various algorithms for solving the 2D and 3D Poisson equation.

challenging or clustered as the user desires. Finite difference calculations are verified by evaluating a function (*e.g.* $sin(xy)$) both symbolically and via the finite difference method. Moreover, we may reuse the same equations for all differential operators and PDEs whether the kernel is categorized within the structured grid, sparse linear algebra, or spectral dwarfs. We may then compare the grid at a subset of the sampled points. Similarly, the result of the Monte Carlo integration kernel can be compared to analytical or numerical results in any dimension.

1.1.4 Solution Quality

The quality of a solution is multifaceted. Thus far, our group has primarily taken the rather narrow focus of optimization of time or energy for a given fixed architecture. Unfortunately, given a set of programmers unrepresentative of the community as a whole (we pride ourselves in our knowledge of architecture and algorithms), we likely minimize the programming and productivity challenges required to attain such performance. In the end, the quality of a solution must take into account not only performance or energy, but must engage the programmer community to determine how productive the solution is. Moreover, the solution must be evaluated on its ability to integrate with existing software and hardware.

At a higher level problem definition, *e.g.* solve Poisson's equation, one can evaluate the quality of a solution relative to the lower bound. For example, let us consider Poisson's equation with zero Dirichlet boundary conditions, discretized by finite differences on a cubic $n \times n \times n$ 3D mesh; see Table 1.2. In effect there are $N = n^3$ unknowns (one per grid point). A number of numerical methods discussed in this report can be used to solve this system of N linear equations with N unknowns. If one selects a dense matrix representation, then he's created N-by-N matrix A has 6's on the diagonal, -1's on 6 off-diagonals, and 0 elsewhere [49]. Such a system has computational complexity of $O(N^3)$ and is thus impractical for all but the simplest problems. Although the sparse and structured grid representations require far less storage and computational complexity, we see there are other algorithms that may reduce the computational complexity to the lower bound of $O(N)$. When designing a system, one may be limited by compute, storage, bandwidth or a relationship between them. As such, selection of the appropriate algorithm is complex.

All these algorithms are of interest on problems sharing at least some of the structure of the Poisson equation; for example if all one knows is that one has a symmetric positive definite band matrix of bandwidth $N^{1/2}$, then one could use band Cholesky and expect a complexity of N^2. Only when exploiting all the structure of the Poisson equation can an optimal (with multigrid) or near-optimal (with FFT) complexity algorithm be achieved.

1.1.5 Reference Implementation

To provide context as to how such kernels productively map to existing architectures, languages and programming models, we have proceeded by attempting to produce a reference implementation for each kernel. As a

reminder, these should be viewed as "hints" designed to show how other designers have mapped a problem's operands and operators to existing hardware and software. Since we wanted such implementations to be illustrative, we tried to ensure they were the most straightforward implementation in the easiest to understand languages using familiar architectures. To that end, most of the kernels are written in both sequential C and MATLAB using array indexing to process matrices, rather than one-line library calls to compute the same kernel. This ensures that the kernel's computation is explicit and readable in the implementation and not hidden behind a library.

Unfortunately, MATLAB has limitations such as awkward facilities for graphs and tree programming, and does not permit low-level control of computations. For these reasons, our reference implementations of kernels such as the Barnes-Hut n-Body solver, the sorting kernels, and the graph algorithm kernels were written in pure C without any supporting library computations.

1.1.6 Optimization Inspiration

There is a dramatic performance gap between the performance that can be attained via productive programming (the most natural means of implementing the problem using existing languages, programming models and hardware) and the style needed to elicit high performance. The discrepancy in performance should not be viewed as the programmer's failing. Rather, it should be viewed as a lighthouse for future research into architecture, languages, and middleware.

There are decades of optimizations for each of the kernels we have enumerated. As such, it would be wasteful to try and recreate all of them. In some cases like Dense Linear Algebra, Sparse Linear Algebra, and Spectral Transforms, there are existing auto-tuned production libraries (ATLAS, OSKI, SPIRAL, FFTW) that can be downloaded [58, 96, 101, 103]. In other cases, like structured grids, sparse linear algebra, and particle methods, we have investigated performance optimization on multicore architectures [38, 48, 78, 104, 105]. Among these and other fields, there are many, many optimized implementations that can be used to guide future implementations. Where appropriate, we list some of the known top-performing algorithmic strategies and optimizations associated with each kernel.

1.2 Previous Work

A shorter version of this document without any details about specific kernels was previously published in [72]. An extended abstract of this paper was subsequently published in [97] by invitation.

1.3 Related Work

There is abundant prior work on defining micro-benchmarks (*e.g.* LINPACK [87] for peak floating-point performance, pChase [86] for memory latency, STREAM [79] for memory bandwidth), benchmarks for evaluating specific architectural and programming models (*e.g.* HPC Challenge [68] for MPI and distributed-memory systems, Parboil [85] and Rodinia [39] for GPUs and CUDA, PARSEC [29] for cache-based multicore, SPLASH [93] for shared-memory systems, and STAMP [80] for transactional memory implementations), benchmarks that are focused on a particular application-space (*e.g.* ALPBench [76] for multimedia applications, BioPerf [11] for computational biology algorithms, Lonestar [74] for graph-theoretic and unstructured grid computations, NU-MineBench [83] for data mining, PhysicsBench [107] for physics simulations, NAS Parallel benchmarks [17] for scientific computing, and the HPCS SSCA benchmark suite [15] for Informatics applications), and large benchmark consortia (*e.g.* SPEC [95] and EEMBC [55]). From our perspective, we view existing benchmarks as reference implementations of one or more kernels (since the problem size, programming language, and algorithms are typically fixed in the benchmark definition). In fact, the Rodinia and the Parallel Dwarfs project [84] teams adopt the Berkeley 13-motif classification to describe the underlying computation in each of their benchmarks.

While all the aforementioned benchmarks serve the computing research community well, their typical usage is to generate a single performance number corresponding to a benchmark-specific metric. Our intent with creating the kernel reference testbed is to drive hardware-software co-design, leading to innovative solutions that can be potentially applied across application domains. Hence we emphasize that our reference and optimized implementations are only hints on how problems should be solved.

1.4 Remainder of this Report

The remainder of this report is organized as follows: Chapters 2 through 13 define kernels, group them by motif, and discuss their key aspects. Table 1.3 summarizes these characteristics and the relationship between kernel and motif. These sections provide the background for those wishing to fully understand the code in the accompanying distribution of reference implementations. They also discuss how one might modify the reference implementation to run larger problems. By no means should this set be construed as being complete, but rather a core set of interesting problems. Similarly, the reference implementations should not be viewed as efficient implementations, but rather a spring board for understanding how the core problem can succinctly be implemented.

1.5 Future Work

This project attempts to address a very broad set of problems, yet the spectrum of scientific computing is so wide that many more kernels remain. We have not implemented or decided on verification schemes for other important structured grid methods including lattice Boltzmann, finite volume, and AMR. We have yet to enumerate any concise representative kernels for unstructured grid methods. An implementation a Particle in Cell (PIC) N-Body solver is planned; as is an implementation of the Fast Multipole Method (FMM).

Also, our kernel selection predominantly reflects scientific computing applications. There are numerous other application domains within computing whose researchers should enumerate their own representative problems. Some of the problems from other domains may be categorized using the aforementioned motifs, some may be categorized into other Berkeley Motifs not listed above (such as branch-and-bound, dynamic programming), while others may necessitate novel motif creation.

In addition to adding these kernels, we are considering work on parallel versions of the reference implementations. We will also accept user feedback and input on these issues and other expansions and modifications to the testbed.

Kernel	Dense Linear Alg.	Sparse Linear Alg.	Structured Grids	Unstructured Grids	Spectral	Particles	Monte Carlo	Graphs & Trees	Sort	Definition	Reference	Optimized†	Scalable Inputs	Verification
Scalar-Vector Mult.	■									✓	✓		✓	
Elementwise-Vector Mult.	■									✓	✓		✓	
Vector Reductions	■									✓	✓		✓	
Scan	■									✓	✓		✓	
Matrix-Vector Mult.	■									✓	✓		✓	
Matrix Transpose	■									✓	✓		✓	
Triangular Solve	■									✓			✓	
Matrix-Matrix Mult.	■									✓	✓		✓	✓
Solve Linear System (LU)	■									✓	✓	✓	✓	✓
Solve Linear System (Cholesky)	■									✓		✓		
Symmetric Eigensolver (QR)	■									✓	✓		✓	✓
SpMV (y=Ax)		■								✓	✓	✓	✓	✓
SpTS (Lx=b)		■								✓	✓		✓	✓
Matrix Powers (y_k=A^kx, PDE)		■								✓	✓	✓	✓	✓
Conjugate Gradient (Ax=b, PDE)	■	■								✓	✓		✓	✓
GMRES (Ax = b)	■	■								✓	✓		✓	✓
Finite Difference Derivatives			■							✓	✓		✓	✓
FD/Laplacian			■							✓	✓	✓	✓	✓
FD/Divergence			■							✓	✓	✓	✓	✓
FD/Gradient			■							✓	✓	✓	✓	✓
FD/Curl			■							✓	✓		✓	✓
Solve FD/PDE (explicit)			■							✓	✓	✓	✓	✓
Solve FD/PDE (via CG)	■		■							✓	✓		✓	✓
Solve FD/PDE (via Multigrid)	■		■							✓	✓		✓	✓
1D FFT (complex→complex)					■					✓	✓		✓	✓
3D FFT (complex→complex)					■					✓	✓		✓	✓
Convolution	■				■					✓	✓		✓	✓
Solve PDE (via FFT)	■		■		■					✓	✓		✓	✓
2D/3D N^2 Direct						■				✓	✓		✓	
2D/3D N^2 Direct (with cut-off)						■				✓	✓		✓	✓
2D/3D Barnes Hut						■		■		✓	✓		✓	✓
Quasi-Monte Carlo Integration							■			✓	✓		✓	✓
Breadth-First Search								■		✓	✓		✓	✓
Betweenness centrality								■		✓	✓		✓	✓
Integer Sort									■	✓	✓		✓	✓
100 Byte Sort									■	✓	✓		✓	✓
Spatial Sort									■	✓	✓		✓	✓

Table 1.3: Brief Overview of enumerated kernels with their mapping to Dwarfs. Check marks denote progress we've made towards a practical testbed for scientific computing. Note, orange boxes denote the mapping of supporting kernels to dwarfs. †There are optimized implementations spread among dozens of production libraries and hundreds of research papers. We will included a subset of our relevant optimized implementations in a future release of this work.

Chapter 2

Dense Linear Algebra

2.1 Lexicon/Terminology for Motif

Dense linear algebra computations involve a set of mathematical operators performed on scalars, vectors or matrices. The word "dense" refers to the data structure accessed during the computation. These data are usually accessed with a regular (or unit) stride and most of the matrix/vector elements are non-zeros. Indexing of data elements can be calculated using a linear relation given the start of the data structure and the cartesian coordinate of an element.

The computations introduced in this chapter can be broadly categorized based on arithmetic intensity of the computation operated upon the data. We first introduce low arithmetic intensity operators, including scalar-vector, vector-vector, matrix-vector, matrix-matrix, vector reduction, vector scan, and dot product, which carry a constant number of arithmetic operations per data element. They are conventionally the basis of more sophisticated kernels and solvers.

The last three computational kernels, LU, Cholesky and symmetric eigen-value decomposition, can be classified as solver and they exhibit high arithmetic intensity. Data are reused within each kernel, with reuse dependent on the size of the data structure. In practice, the amount of data referenced is usually too large to fit in a typical cache hierarchy, leading to observing smaller floating-point intensity per cached data. Tiling (or blocking) is a common technique that can improve the data reuse for limited cache (or local store) sizes.

Dense linear algebra algorithms usually have a good scaling behavior for multiple reasons: the degree of parallelism grows with the dataset size, and the intensity of floating-point operations per memory access can be well-balanced with cache-based (or local-store based) microprocessor designs.

2.2 Scalar-Vector Operation

A scaler-vector operation on a vector x and a scalar α defines a binary associative operator on the elements of the vector x paired with the scalar value α. The results of these operations are stored in an output vector y. A scalar-vector normally appears in a sequence of operators on the data, used by a high-level solver.

2.2.1 Formal Mathematical Definition

A scalar-vector operation on a vector $x \in (\mathbb{R}|\mathbb{C})^n$ and a scalar $\alpha \in \mathbb{R}|\mathbb{C}$, resulting in a vector $y \in \mathbb{R}|\mathbb{C}$ is defined as:

$$y = op(\alpha, x) \tag{2.1}$$

where *op* can be any binary operation, such as $+, -, \times, /, min, max$, *etc.* For instance, scalar-vector multiplication is defined as $y = \alpha \times x$.

2.2.2 Scalable Problem Definition

It is usually possible to create a source vector of an arbitrary size. The importance of the content of the vector depends on the operator involved. For instance, for addition, zero initialization could be enough to measure performance, *min* operator requires random initialization of the vector. In practice, as these kernels are not used in isolation, the initialization depends on the solver that uses these operations.

2.2.3 Verification Scheme

Verification of these operations is usually done in the context of verifying a high-level solver that calls these operations. Verifying these operations in isolation, though straightforward, is usually not needed.

2.2.4 Functional Reference Implementation

Algorithm 1 presents a generic template for computing vector-scalar operation. The computation involves a simple loop to apply the operator on the elements of the vector x with the scalar α.

1: **for** $i = 1$ to n **do**
2: $\quad y_i \leftarrow op^1(\alpha, x_i)$
3: **end for**

Algorithm 1: A generic reference implementation for a scalar-vector operation.

2.2.5 Literature Survey of Optimiations

The scalar-vector operation performance usually relies on the effectiveness of streaming the vector data. Prefetcher, whether based on software or hardware-support, can improve the performance of this kernel. Traditionally, optimization of this kernel is better done in the context of the solver that utilize this operator. For instance, data layout, which can influence the ability of the compiler to vectorize the code, cannot be optimally decided without studying other instances of accessing the data.

2.3 Elementwise Vector-Vector Operations

A vector-vector operation on the vectors x and y defines an associative operator for each corresponding element pairs of x and y. The results of these operations are stored in the corresponding elements of a vector z. Similar to scalar-vector operations, they normally appear in a sequence of operators on the data.

2.3.1 Formal Mathematical Definition

A vector-vector operation on $x \in (\mathbb{R}|\mathbb{C})^n$ and $y \in (\mathbb{R}|\mathbb{C})^n$, resulting in a vector $z \in (\mathbb{R}|\mathbb{C})^n$ can be defined as follows:

$$z = op^1(x, y) \tag{2.2}$$

For instance, vector-vector addition is defined as $z = x + y$.

2.3.2 Scalable Problem Definition

See Section 2.2.2 for scalable problem definition comment on simple operators.

2.3.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.3.4 Functional Reference Implementation

Algorithm 2 presents a generic template for computing a vector-vector operation. It involves a simple loop over the elements of the vectors x and y to apply the arithmetic operator and to store the results in the corresponding element of z.

```
1: for i = 1 to n do
2:     z_i ← op[1](x_i, y_i)
3: end for
```

Algorithm 2: Vector-vector operation reference implementation.

2.3.5 Literature Survey of Optimiations

Optimization of these operations requires data alignment of the input vectors such that they have good spatial locality. As no reuse of data is involved, most of the optimization effort focuses on efficient streaming of the data, including loop unrolling, data prefetching, and pipelining computation. On most architecture these operations will be bounded by the bandwidth to the memory. Generally, optimizing these operators is better done in conjunction with other operators on the data, by trying to maximize locality in accessing the data. See also Section 2.2.5 for additional comments on optimizations for these simple operations.

2.4 Vector Reductions (sum/product/min/max)

A vector reduction operation on a vector x defines an associative operator used in sequence on the elements of the vector x, resulting in a scalar z.

2.4.1 Formal Mathematical Definition

A reduction on a vector $x \in (\mathbb{R}|\mathbb{C})^n$ to compute a scalar $z \in \mathbb{R}|\mathbb{C}$ can be defined, for instance, for the sum reduction as:

$$z = \sum_{i=1}^{n} x_i \tag{2.3}$$

A vector-multiplication reduction can be defined as

$$z = \prod_{i=1}^{n} x_i \tag{2.4}$$

Similarly, other reduction operators, such as min(), or max(), can be defined as well.

2.4.2 Scalable Problem Definition

See Section 2.2.2 for scalable problem definition comment on simple operators.

2.4.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.4.4 Functional Reference Implementation

Algorithm 3 presents a generic template for computing vector-reduction operation. The computation is done through a simple loop over the vector x, while applying the operator for the scalar z and the elements of x. The initial value of the scalar z is the identity value of the operator.

[1]The *op* can be any binary associative operator such as $+$, $-$, $\times$, $/$, min, or max

```
1: z ← identity
2: for i = 1 to n do
3:     z ← op^1(z, x_i)
4: end for
```

Algorithm 3: A generic reference implementation for vector reduction operations.

2.4.5 Literature Survey of Optimiations

See Section 2.2.5 for comments on optimizations for these simple operations.

2.5 Dot Product

A dot product operation on two vectors x and y involves finding the sum over multiplication of corresponding element pairs of x and y. The result of these operations is stored in a scalar z.

2.5.1 Formal Mathematical Definition

The dot product of two vectors $x \in (\mathbb{R}|\mathbb{C})^n$ and $y \in (\mathbb{R}|\mathbb{C})^n$ is defined as:

$$z = x \cdot y \tag{2.5}$$

2.5.2 Scalable Problem Definition

See Section 2.2.2 for scalable problem definition comment on simple operators.

2.5.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.5.4 Functional Reference Implementation

Algorithm 4 presents a simple implementation for the dot product of two vectors. A single loop nest is needed to carry out the computation.

```
1: z ← 0
2: for i = 1 to n do
3:     z ← z + x_i × y_i
4: end for
```

Algorithm 4: Dot product of two vectors x and y.

2.5.5 Literature Survey of Optimiations

See Section 2.2.5 for comments on optimizations for these simple operations.

2.6 Scan

A scan operation can appear in solving long recurrence equations and in particle filter methods in statistics. In a scan operation of an input vector x, resulting in an output vector y, each element of the output vector corresponds to a reduction of the input vector elements up to the index output element. The operation can be inclusive of the target index or exclusive of it. Computationally, both inclusive and exclusive operations are similar, thus we will focus in the following discussion on one of them, for instance inclusive scan. For brevity, we will focus on scan sum operation.

2.6.1 Formal Mathematical Definition

A scan-sum operator on a vector $x \in (\mathbb{R}|\mathbb{C})^n$ resulting in a vector $y \in (\mathbb{R}|\mathbb{C})^n$ is defined as:

$$y = Ax \tag{2.6}$$

where matrix A is a lower triangular matrix, whose non-zeros elements are all equal to one.

2.6.2 Scalable Problem Definition

See Section 2.2.2 for scalable problem definition comment on simple operators.

2.6.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.6.4 Functional Reference Implementation

Algorithm 5 shows a simple reference implementation for the scan operation.

1: $y_1 \leftarrow x_1$
2: **for** $i = 2$ to n **do**
3: $\quad y_i \leftarrow y_{i-1} + x_i$
4: **end for**

Algorithm 5: A reference implementation for scan operation of an input vector x .

2.6.5 Literature Survey of Optimiations

See Section 2.2.5 for comments on optimizations for these simple operations.

2.7 Outer Product

The outer product involves multiplying two vectors x and y resulting in a matrix A. Outer product can be seen as a special case of matrix-matrix multiplication, where the inner dimensions are unity.

2.7.1 Formal Mathematical Definition

The outer product of a vector $x \in (\mathbb{R}|\mathbb{C})^{1\times n}$ multiplied by $y \in (\mathbb{R}|\mathbb{C})^{1\times m}$ resulting in a matrix $A \in (\mathbb{R}|\mathbb{C})^{n\times m}$ is defined as

$$A = x^T \otimes y \tag{2.7}$$

2.7.2 Scalable Problem Definition

See Section 2.2.2 for scalable problem definition comment on simple operators.

2.7.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.7.4 Functional Reference Implementation

Algorithm 6 shows a simple reference implementation for the outer-product operation.

```
1: for i = 1 to n do
2:     for j = 1 to m do
3:         A_{i,j} ← x_i × y_j
4:     end for
5: end for
```

Algorithm 6: An outer product of two vectors x and y reference implementation.

2.7.5 Literature Survey of Optimiations

See Section 2.2.5 for comments on optimizations for these simple operations.

2.8 Matrix-Vector Multiplication

The matrix-vector product involves multiplying a matrix A by a vector x resulting in a vector y. A matrix-vector multiplication is a special case of matrix-matrix multiplication where one dimension of a matrix is unity.

2.8.1 Formal Mathematical Definition

Multiplying a matrix $A \in (\mathbb{R}|\mathbb{C})^{n \times m}$ by a vector $x \in (\mathbb{R}|\mathbb{C})^{1 \times m}$ resulting in a vector $y \in (\mathbb{R}|\mathbb{C})^{n \times 1}$ is defined as follows:

$$y = A \cdot x^T \tag{2.8}$$

2.8.2 Scalable Problem Definition

See Section 2.2.2 for scalable problem definition comment on simple operators.

2.8.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.8.4 Functional Reference Implementation

Algorithm 7 shows a simple reference implementation for the matrix-vector multiplication. A two-nested loop is needed to multiply each row of the matrix by the input vector resulting in an element of the output vector.

```
1: for i = 1 to n do
2:     y_i ← 0
3:     for j = 1 to m do
4:         y_i ← y_i + A_{i,j} × x_j
5:     end for
6: end for
```

Algorithm 7: A matrix-vector reference implementation.

2.8.5 Literature Survey of Optimiations

See Section 2.2.5 for comments on optimizations for these simple operations.

2.9 Matrix Transpose

Matrix transpose is a frequently used computational kernel, which does not use the computational capability of the system, but it stresses the latency and bandwidth of the memory subsystem.

2.9.1 Formal Mathematical Definition

Transposing a matrix $A \in (\mathbb{R}|\mathbb{C})^{n \times m}$ into a matrix $B \in (\mathbb{R}|\mathbb{C})^{m \times n}$ is denoted by

$$B = A^T \tag{2.9}$$

2.9.2 Scalable Problem Definition

The kernel processing does not depend on the content of the data making it easy to generate a problem of arbitrary size. Choosing m, n parameters control the amount of processing, and the smaller the difference between m and n the more the involved data movement. See Section 2.2.2 for additional comments on scalable problem definition comment on simple operators.

2.9.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.9.4 Functional Reference Implementation

The simplest implementation of matrix transpose involves double nested loop, as shown in Algorithm 8. Copying from the source matrix to the destination matrix can be characterized by unit-stride reads from the source matrix and regularly distanced writes to the destination. This simple implementation does not exploit any caching and the performance is usually bounded by the streaming capabilities of the computing elements.

```
1: for i = 1 to n do
2:     for j = 1 to m do
3:         B_{i,j} ← A_{j,i}
4:     end for
5: end for
```

Algorithm 8: Matrix transpose reference implementation.

2.9.5 Literature Survey of Optimiations

A naïvely implemented matrix transpose on a general-purpose processor may suffer performance degradation because of the latency to access the memory system. Using prefetching can push the bottleneck to the bandwidth of the memory system. Blocking or tiling can substantially improve the performance of this kernel. It divides a matrix into smaller blocks that can reside in the processor cache hierarchy. The transpose process is done in two phases, the first involves transposing each blocks. The second phase involves transposing these blocks. The nested process can normally be extended depending on the number of levels of the cache hierarchy and the level of sophistication of the data structure and coding.

2.10 Triangular Solve

Triangular solve is common technique in solving a system of linear equations, when the matrix representing the equations has either upper or lower triangular form. A lower triangular matrix has all elements above the diagonal equal to zero, while upper triangular has all elements below diagonal as zeros. A square matrix can be converted into a triangular matrix using several decomposition techniques such as Schur decomposition. In the following discussion we will focus on solving a system of linear equation where the matrix is already in a triangular form.

2.10.1 Formal Mathematical Definition

Triangular solve for a system of linear equations for a vector $x \in (\mathbb{R}|\mathbb{C})^n$ in the form

$$Ax = b \tag{2.10}$$

where $A \in (\mathbb{R}|\mathbb{C})^{n \times n}$ is a lower triangular matrix. Solving the system in Eqn. 2.10 usually involves iterative substitution.

2.10.2 Scalable Problem Definition

Generating a scalable dataset for this problem can be done using random initialization of the array, while setting the upper triangular elements to zeros for lower triangular arrays. Generally this kernel is used by other high-level routine that does the initialization according to the problem.

2.10.3 Verification Scheme

The verification of this problem can be done through multiplying the matrix A by the vector x and comparing the results with the vector b, certainly with tolerance to the system precision.

2.10.4 Functional Reference Implementation

Algorithm 9 shows a simple reference implementation for the triangular solve. Two nested loops are needed; the inner loop is used to back-substitute the computed system unknowns; and the outer is used to compute one of system unknown.

```
1: x_1 ← b_1/A_{1,1}
2: for i = 1 to n do
3:     x_i ← b_i
4:     for j = 1 to i do
5:         x_i ← x_i − A_{i,j} × x_j
6:     end for
7:     x_i ← x_i/a_{ii}
8: end for
```

Algorithm 9: Triangular solve reference implementation.

2.10.5 Literature Survey of Optimiations

See Section 2.2.5 for comments on optimizations for these simple operations.

2.11 Matrix-Matrix Multiplication

Matrix-Matrix multiplication is frequently used in iterative methods, where a direct solve is difficult to compute. It involves multiplying two matrices A and B resulting in a matrix C.

2.11.1 Formal Mathematical Definition

The multiplication of two matrices $A \in (\mathbb{R}|\mathbb{C})^{n \times r}$, and $B \in (\mathbb{R}|\mathbb{C})^{r \times m}$ resulting in a matrix $C \in (\mathbb{R}|\mathbb{C})^{n \times m}$ is defined as

$$C = AB \tag{2.11}$$

The multiplication is defined *iff* the second dimension the matrix A is equal to the first dimension of the first matrix. The computation complexity is $O(n \times r \times m)$ and the memory access complexity is $O(n \times r + r \times m)$. We will restrict the discussion in this section to the case when $n = r = m$, where the highest computational intensity (floating point per memory words) is observed as $O(n)$.

2.11.2 Scalable Problem Definition

The computations of matrix-matrix multiplication do not depend on the values of the matrix elements. Generating scalable problem is straightforward; the dimensions of the input arrays can be varied arbitrarily, and the contents of the arrays can be randomly initialized. See Section 2.2.2 for additional comments on scalable problem definition comment on simple operators.

2.11.3 Verification Scheme

See Section 2.2.3 for problem verification comment on simple operators.

2.11.4 Functional Reference Implementation

The simplest implementation of matrix-matrix multiplication involves a triple nested loop, as shown in Algorithm 10. Even though each element of the input matrices A and B is visited multiple times during the course of computation, the reference implementation may not be able exploit caches to capture this locality, especially when the matrices sizes exceed the size of the cache.

```
1: for i = 1 to n do
2:   for j = 1 to m do
3:     C_{i,j} ← 0
4:     for k = 1 to r do
5:       C_{i,j} ← C_{i,j} + A_{i,k} × B_{k,j}
6:     end for
7:   end for
8: end for
```

Algorithm 10: Matrix-matrix multiplication reference implementation.

2.11.5 Literature Survey of Optimiations

For realistic datasets, input matrices cannot fit in the caching system on most microprocessors. Code optimizations for matrix-matrix multiplication try to expose temporal locality of references for the input matrices. Conventionally, this is accomplished by blocking the input matrices into $b \times b$ blocks and computing resultant matrix by multiplying the smaller blocked matrices. The block size is chosen to guarantee that one block of matrix A and another from B will fit in the highest level of cache, $L1$ cache for instance [75]. Blocking for other level of caches can improve performance as well.

This computation is usually amenable to vectorization; depending on the architecture memory, alignment and padding may be needed [75]. Given the streaming behavior of this computation prefetching can be leveraged using software or hardware techniques.

2.12 Solver of Linear System (LU Factorization)

LU factorization aims at factorizing a matrix to achieve a triangular form that can be used to solve a system of linear equations easily.

A matrix $A \in \mathbb{R}^{n \times n}$ has an LU factorization *iff* all its leading principle minors are non-zeros, *i.e.*, $\det(A\,[1:k, 1:k]) \neq 0\ for\ k = 1:n-1$. A non-singular matrix can have an LU factroization if it satisfy this condition. For detailed description on the conditions for existance and uniqueness readers can refer to [64]. The LU factorization can be used in solving a system of linear equations.

2.12.1 Formal Mathematical Definition

The LU factorization of a matrix A takes the form:

$$A = LU \tag{2.12}$$

where L and U are lower and upper triangular matrices, respectively. Their dimensions are similar to that of the matrix A.

2.12.2 Scalable Problem Definition

Scalable problem definition can start by generating a randomly initialized lower and upper matrices L, and U, then multiplying them to generate the input matrix A.

2.12.3 Verification Scheme

The verification of the LU factorization is usually straight forward by multiplying the upper and lower triangular part of the factorization and then computing the root mean square error of the resultant matrix compared with the original one. Verification success is usually achieved when the residual is less than a predetermined tolerance value.

The input dataset initialization should ensure the possibility of the factorization and ease the verification by reducing the rounding effect.

2.12.4 Functional Reference Implementation

A common technique to compute the upper triangular part of the factorization, U, is to use gausian elimination, see for instance [52]. The main computation is a triple-nesting loop procedure that is shown in Algorithm 11. The floating-point operations involved in this computation is $O\left(n^3\right)$, while the data accessed are $O\left(n^2\right)$. This shows that the arithmatic intensity of the alogorihm, defined as floating-point operations per byte, increases with the problem size. For practical problem sizes the dataset cannot fit in the cache, leading to a much less reuse of the cached data than what the computational kernel has. In the next section, we will discuss some of the techiques used to improve the data reuse.

```
1:  for i = 1 to n do
2:      Find maximum absolute element in column i below the diagonal
3:      Swap the row of the maximum element with row i
4:      for j = i + 1 to n do
5:          L_{j,i} ← A_{j,i} / A_{i,i}
6:      end for
7:      for j = i to n do
8:          U_{i,j} ← A_{i,j}
9:      end for
10:     for j = i + 1 to n do
11:         for k = i + 1 to n do
12:             A_{j,k} ← A_{j,k} - L_{j,i} × U_{i,k}
13:         end for
14:     end for
15: end for
```

Algorithm 11: Computing LU factorization using Gaussian elimination method.

2.12.5 Literature Survey of Optimiations

Multiple variants of the same algorithm exist to handle rectangular matrix [52], to reduce the rounding errors, and to improve the computation efficiency. Most optimization tries to rearrange the data and computation such that more temporal locality is exploited while factorizing the array. A notable optimization is tiling the main array into smaller blocked arrays that can fit on a cache size, depending on the targeted hardware [40, 69]. The computation can also be easily vectorized by a compiler when the data layout carries spatial contiguity.

2.13 Solver of Linear System (Cholesky Factorization)

A well-known technique for solving a system of linear equation is Cholesky decomposition, which transform a matrix, A, into a product of an upper triangular matrix U and its conjugate transpose. Cholesky decomposition can be much faster than alternative methods for solving a system of linear equation for the special class of matrices that are symmetric (or Hermitian for complex matrices) and positive definite.

2.13.1 Formal Mathematical Definition

Given a positive definite matrix $A \in (\mathbb{R}|\mathbb{C})^{n \times n}$, Cholesky decomposition seeks to find an upper triangular matrix U such that:

$$A = U^T U \tag{2.13}$$

The condition for a matrix A to be positive definite can be stated as for all non-zeros complex vectors $\mathbf{x} \in \mathbb{C}^n$ then

$$\mathbf{x}^* A \mathbf{x} > 0$$

where $\mathbf{x}^*$ is the conjugate transpose of $\mathbf{x}$.

2.13.2 Scalable Problem Definition

The only condition that needs to be satisfied for the input matrix A is that it has to be positive definite to guarantee convergence to solution. The actual content of the matrix does not affect the amount of computation needed. A scalable definition can involve creating an upper diagonal matrix and then multiply it by its conjugate to compute the initial matrix.

2.13.3 Verification Scheme

Verification of this kind of problem is usually straight forward based on the problem definition. Given the upper triangle matrix U, one can multiple it by its conjugate and compare the results to the original matrix A, certainly with some tolerance due to precision and rounding errors.

2.13.4 Functional Reference Implementation

Cholesky Method have some commonality with LU factorization in terms of the computational kernel. Algorithm 12 outlines the procedure involved in computing the elements of the Matrix U.

Cholesky factorization generally requires $1/3n^3$ flops, approximately half that is needed for LU factorization. It also does not need pivoting, but it applies to restricted class of matrices.

The reference implementation of this kernel will be included in a future release of this testbed.

2.13.5 Literature Survey of Optimiations

The computational characteristics of Cholesky carry a lot of similarity with LU factorization. Doing computation element-wise is not the most efficient way, instead the matrix is divided into square tiles (or blocks) to operate upon.

Load balancing the workload is usually tricky because the amount of work increases with the outer loop, unlike LU where the workload decreases. In both cases, the conventional approach is to have a cookies cutter assignment, for more information reader can refer to [64].

```
1: for i = 1 to n do
2:   for j = i to n do
3:     sum ← A_{i,j}
4:     for k = 1 to i do
5:       sum ← sum - L_{i,k} × L*_{j,k}
6:     end for
7:     if i = j then
8:       if sum < tol then
9:         Cholesky failed (A not positive definite)
10:      end if
11:      U_{i,i} ← √sum
12:    else
13:      U_{j,i} ← sum / U_{i,i}
14:    end if
15:  end for
16: end for
```

Algorithm 12: Computing Cholesky U Matrix for a positive definite matrix A.

2.14 Symmetric Eigenvalue Decomposition

Eigenvalue computation is of great importance to many problems in physics. The computation of eigenvalue is of great importance in control system theory, vibration analysis, quantum mechanics theory, etc. Depending on the problem domain, the largest eigenvalues in magnitude (called the *dominant eigenvalues*) can be of more importance than others.

2.14.1 Formal Mathematical Definition

The eigenvalues of a matrix $A \in \mathbb{R}^{n\times n}$ are the n roots $\{\lambda_1, \cdots, \lambda_n\}$ of its characteristic polynomial $p(z) = \det(A = zI)$. The determinant of the matrix A can be expressed as $\prod_{i=1}^{n} \lambda_i$. A non-zero vector $x \in \mathbb{C}^n$ is referred to as eigenvector if it satisfy

$$Ax = \lambda_i x \tag{2.14}$$

The computation of eigenvalues must be iterative if $n > 4$. One way of computing eigenvalues is based on QR factorization of the matrix A that will be discussed in detail in Section 2.14.4. Symmetry guarantees that the eigenvalues of A are real and that there is an orthogonal basis for eigenvectors. In the following discussion we will consider only the case where the matrix A is symmetric, and will only compute eigenvalues of the given matrix.

2.14.2 Scalable Problem Definition

The generation of scalable random matrix starts with choosing an eigenvalue distribution. For a given eigenvalues λ, a diagonal matrix, D, is created as $D = diag(\lambda)$. An orthogonal matrix Q, generated from a random entries, can be used to generate the input matrix $A = Q^T DQ$. The matrix A can be reduced to a tridiagonal form. The size of the eigenvalues and its distribution control the difficulty associated with the computation. We followed the path set by Demmel et. al [50] in generating multiple input test cases that allow to uncover not only the performance of a particular algorithm, but also its idiosyncrasies.

2.14.3 Verification Scheme

To verify that eigenvalues are correctly computed, we simply compare the computed eigenvalues of the product $A = Q^T DQ$ with the original generated eigenvalues. The maximum difference between the eigenvalues must satisfy the user specified tolerance to pass. Default tolerance is 10^{-10}.

Also, the computation of the eigenvalues is frequently associated with the computation of eigenvectors. The numerical accuracy of the computation [50] usually involves verification of the orthogonality of the eigenvectors and the accuracy of the eigen-pairs. For certain applications, only the dominant eigenvalues are of importance. Assuming m dominant eigenvalues are computed for the tridiagonal matrix $A \in \mathbb{R}^{n \times n}$, where the eigenvalues are $W = diag(w_1 w_2 \cdots w_m)$ and the associated eigenvectors are $Z = [z_1 z_2 \cdots z_m]$, then the orthogonality of the eigenvectors and the accuracy of the eigenvalues can be computed as follows:

$$orthogonality = \begin{cases} \frac{\|I - ZZ^T\|}{n \times ulp} & if\ m = n \\ \frac{\|I - Z^T Z\|}{n \times ulp} & if\ m < n \end{cases}$$

$$accuracy = \begin{cases} \frac{\|A - ZWZ^T\|}{\|A\| \times n \times ulp} & if\ m = n \\ \frac{\|Z^T AZ - W\|}{\|A\| \times n \times ulp} & if\ m < n \end{cases}$$

where *ulp* is the *unit of least precision* of a particular system, representing the numerical precision. The current reference implementation only computes eigenvalues and not eigenvectors. Thus, it does not include this more sophisticated and expensive verification scheme.

2.14.4 Functional Reference Implementation

To compute the eigenvalues of a symmetric matrix A for a given tolerance greater than *ulp*, the symmetric QR algorithm, see Algorithm 14, can be used. The algorithm uses an implicit QR step with Wilkinson shift [64]. At the end of the computation, the Diagonal elements of the matrix D will carry the computed eigenvalues. The reference implementation computes only the eigenvalues and not the eigenvectors as this provides significant savings.

1: $d \leftarrow (T_{n-1,n-1} - Tn, n)/2$
2: $\mu \leftarrow T_{n,n} - T^2_{n,n-1}/(d + sign(d)\sqrt{d^2 + t^2_{n,n-1}})$
3: $x \leftarrow T_{1,1} - \mu$
4: $z \leftarrow T_{2,1}$
5: $Z \leftarrow I$
6: **for** $k \leftarrow 1ton - 1$ **do**
7: $\quad G_k \leftarrow$ compute Givens rotation [64] of x and z
8: $\quad T \leftarrow G_k^T \cdot T \cdot G_k$
9: $\quad$ **if** $k < n - 1$ **then**
10: $\quad\quad x \leftarrow T_{k+1,k}$
11: $\quad\quad z \leftarrow T_{k+2,k}$
12: $\quad$ **end if**
13: $\quad Z \leftarrow Z \cdot G_k$
14: **end for**

Algorithm 13: Implicit QR step.

Algorithm 14 shows the iterative algorithm to compute the eigenvalues. All A_i have the same eigenvalues. The algorithm converges when the A_i becomes a triangular matrix and the eigenvalues becomes simply the diagonal elements of the matrix.

In practice, it is impractical to compute the exact eigenvalues instead a certain error tolerance can be set for convergence test. The eigenvectors can be computed as a column of the product of the orthogonal transformation $Q = Q_1 Q_2 \cdots Q_i$.

2.14.5 Literature Survey of Optimiations

The optimization of the above formulation for computing eigenvalue is tightly related to improving the QR factorization part. Tiling the QR computations can significantly improve the cache performance [37, 40, 41].

$T \leftarrow Q^T \cdot A \cdot Q$ {Compute tridiagonalization of the input matrix by finding an orthogonal matrix Q}
$D \leftarrow T$
$q \leftarrow 0$
while $q \neq n$ **do**
  for $i = 0$ to $n - 1$ **do**
    if $min\left(\|d_{i+1,i}\|, \|d_{i,i+1}\|\right) \leq tol \cdot \left(\|d_{i,i}\| + \|d_{i+1,i+1}\|\right)$ **then**
      $D_{i+1,i} \leftarrow 0$
      $D_{i,i+1} \leftarrow 0$
    end if
  end for
  Format the matrix D such that q is the largest and p is the smallest

$$D = \begin{bmatrix} D_{1,1} & 0 & 0 \\ 0 & D_{2,2} & 0 \\ 0 & 0 & D_{3,3} \end{bmatrix} \begin{matrix} p \\ n-p-q \\ q \end{matrix}$$
$$\begin{matrix} n & n-p-q & q \end{matrix}$$

  Where $D_{3,3}$ is diagonal and D2,2 is unreduced.
  if $q < n$ **then**
    apply ImplicitQR, §Algorithm 13, on $D_{2,2}$
    $X \leftarrow diag\left(I_p, \bar{Z}, I_q\right)$
    $D \leftarrow X^T \cdot D \cdot X$
    $Q \leftarrow Q \cdot X$
  end if
end while

Algorithm 14: Computing eigenvalues based on QR algorithm.

Some domain specific optimization can reduce the amount of computation by computing the only dominant eigenvalues.

Chapter 3

Band Linear Algebra

Where the dense linear algebra chapter enumerated computational kernels that operated on dense or triangular matrices, this chapter deals with matrices whose nonzero elements are clustered within a band along the diagonal. As operations on zeros may be statically eliminated, we may define a complementary set of computational kernels that exploit this property.

These kernels will be enumerated in a future release of this testbed.

Chapter 4

Sparse Linear Algebra

In this chapter we examine generalized sparse linear algebra and discuss its key aspects, challenges, and computational kernels. We will examine several basic kernels: SpMV, SpTS, matrix powers, as well as integrate them into more complex PDE solvers including explicit, implicit/iterative, and implicit/direct methods.

4.1 Lexicon/Terminology for Motif

Like dense linear algebra, sparse linear algebra operates principally on three objects: scalars, vectors, and matrices. However, what distinguishes sparse linear algebra from dense is the fact that the vectors and matrices can be "sparse". That is, a predominance of the elements are zero. Thus, like band linear algebra, operating only on the nonzeros reduces both the bulk storage and computational requirements. We differentiate sparse linear algebra from band in that there is no mandated rhyme or reason as to the "sparsity" pattern, or distribution, of nonzeros. As a result, substantial storage and (non floating-point) computational overhead is required to represent and operate directly on a sparse vector or matrix. In practice, there is relatively little difference in sparse vectors and sparse matrices as the former can easily be represented as a 1×N special case of the latter. As such, in this chapter we will focus on operations on sparse matrices and dense vectors.

Where dense linear algebra is dominated by one of two representations (column-major or row-major), in sparse linear algebra there is a myriad of formats that have been tailored for matrix, kernel, and processor architecture. However, by far, the most common format (and the one used for all included reference implementations) is compressed sparse row (CSR). In CSR, each matrix is represented by 3 arrays. First, we have an array containing all the nonzero values (`V[number_of_nonzeros]`). It is key that the elements of this array be properly sorted. All nonzeros within a row of the matrix are contiguous in the array. Additionally all such blocks of nonzeros are sorted by their row. Thus, if we were to examine the array, we would first see a block of nonzeros for row 0, immediately followed by a block of nonzeros for row 1, and so forth. Mirroring the nonzero value array is an array that hold the nonzero column indices (`V[number_of_nonzeros]`). Finally, to express how these array of nonzeros are partitioned into rows, we include a row pointer array (`P[number_of_rows]`). By convention we interpret its contents as the array index of the first element of each row either starting with row 0 or at row 1. Thus, for each nonzero, we explicitly store its column address while implicitly storing its row address (calculated by position in the array). In the dense linear algebra world, both indices would be implicitly stored (calculated based on position within the array).

4.2 Sparse Matrix-Vector Multiplication (SpMV)

Perhaps the canonical sparse kernels is sparse matrix-vector multiplication (SpMV). Unlike the dense matrix-vector multiplication, SpMV is challenged by random memory access and locality that can not be predicted without first inspecting the matrix. We see SpMV at the core of many routines across many domains including scientific computing, financial modeling, and information retrieval. As such, understanding its performance characteristics and optimizing for them is often key to improving application performance.

4.2.1 Formal Mathematical Definition

Simply put, we evaluate $\vec{y} = \mathbf{A} \cdot \vec{x}$, where $\mathbf{A}$ is a sparse matrix and $\vec{x}$ and $\vec{y}$ are dense vectors. In the dense world, we would express this as $y_i = \sum_{i=j}^{n} A_{i,j} \times x_j$. However, as most matrix entries ($A_{i,j}$) are zero, it is not uncommon to rework the problem definition to only operate on the nonzeros:

$\vec{y} = 0$
$\forall$ nonzero $A_{i,j}$:
  $y_i = y_i + A_{i,j} \times x_j$

More complex operators, including $\vec{y} = \mathbf{A}\vec{x} + \vec{z}$ and $z = \vec{x}^T \mathbf{A}\vec{x}$, and solvers like Conjugate Gradient are often built from SpMV. As such, the challenges and insights garnered from optimizing SpMV are broadly applicable.

4.2.2 Scalable Problem Definition

For simplicity, consider the finite difference method discussed in Chapter 5. We may leverage the grid, its discretization, and its initial and boundary conditions as a means of providing the sparse linear algebra routines with a similar scalable problem definition and verification scheme. In essence, the values of the discretized grid (including the boundary) may be encapsulated into the source vector. Similarly, as the finite difference method produces linear operators, we may encapsulate the elements accessed as nonzeros in a row and the weights as their values. We may define a matrix for each of the 7 basic operators discussed in Chapter 5. By themselves, these operators each translate into one SpMV operation.

For example, given a N×N×N grid and the 3D homogeneous Laplacian operator, we produce a matrix with N^3 columns and N^3 rows with 7 nonzeros in the majority of rows (fewer nonzeros appear in rows corresponding to grid boundaries). With the simplest mapping of grid points to array (vector) indices, we produce a septa-diagonal matrix. To fully test the breadth of complexity, we may change the enumeration (mapping) of grid points within a row, plane, or in 3D, the entire volume. Doing so permutes the rows and columns of the matrix. Furthermore, to exacerbate load balancing challenges, we may randomly add explicit zeros to the matrix. Thus, the number of nonzeros per row can be variable and significantly larger than 7. These additional elaborations to the problem definition will be explored in a future release of this testbed, along with matrix versions of remaining differential operators.

4.2.3 Verification Scheme

To verify these kernels, we may find the solution to the differential operator analytically. We may then descretize and enumerate the solution grid commensurate with how we descretized and enumerated the input grid. We then compare element by element and in element wise norm. This verification code is in the Structured Grid section of the reference implementations because operators for comparison and initialization of grids reside there. The test set for this operator is described in the following table. All the included functions have homogeneous boundary conditions on the three dimensional unit cube. As discussed in Section 5.3, the parameter τ (called *toughness* in the code) is a user specified difficulty parameter, which is modified in the calling function to ensure that the function is actually zero on the boundary of the cube. A coefficient is placed in in the first function to make the numerical difficulty in evaluating its Laplacian similar to that of the second.

4.2.4 Functional Reference Implementation

The following is the simplest CSR implementation where `x[ ]` and `y[ ]` are the destination vectors, `Pointers[ ]` is the row pointer array, `Column[ ]` is the array of nonzero column indices, and `Values[ ]` is the array of nonzero values.

```
nnz=0;
for(row=0; row<N; row++) {
  double y0 = 0.0;
  while(nnz<Pointers[row+1]){
    y0 += Value[nnz] * x[Column[nnz]];
    nnz++;
  }
```

$u(x,y,z)$	$\nabla^2 u$
$\frac{1}{40}sin(\tau\pi x)sin(\tau\pi y)sin(\tau\pi z)$	$-\frac{1}{40}(3\tau^2\pi^2)sin(\tau\pi x)sin(\tau\pi y)sin(\tau\pi z)$
$\tau(x^2-x)(y^2-y)(z^2-z)$	$2\tau((x^2-x)(y^2-y)+(x^2-x)(z^2-z)+(y^2-y)(z^2-z))$

Table 4.1: Test functions for Homogeneous Laplacian to be evaluated with SpMV and exact solutions to the continuous problem. Sampling will provide a check against the finite difference solution.

```
    y[row] = y0;
}
```

4.2.5 Literature Survey of Optimizations

SpMV has seen a plethora of optimizations. Broadly speaking these can be classified into minimizing memory traffic (armotization of meta data, exploitation of symmetry, elimination of cache capacity misses), maximizing bandwidth (e.g. prefetching), and improving in-core performance (maximizing ILP and DLP, whilst minimizing instruction overheads per nonzero). Williams et al's recent multicore papers provide excellent information on performance optimization [105, 106].

4.3 Sparse Triangular Solve (SpTS)

Like dense triangular solve, sparse triangular solve (SpTS) solves either $\mathbf{L}\vec{x} = \vec{b}$ or $\mathbf{U}\vec{x} = \vec{b}$. However, unlike the dense world where parallelism abounds and is predictable as one proceeds, the sparsity of $\mathbf{L}$ or $\mathbf{U}$ immensely complicates the efficient execution of this kernel.

4.3.1 Formal Mathematical Definition

Recursively, we often define triangular solve as:
$x_i = \frac{1}{L_{i,i}}(b_i - \sum_{j=1}^{i-1} L_{i,j} \times x_j)$.

Although such a definition suggests operating on rows, it is possible to restructure the algorithm to operate on columns of $\mathbf{L}$ by starting with the solution to a simpler problem $\mathbf{I}\vec{x} = \vec{b}$ ($\vec{x} = \vec{b}$) and refining it by deducting the off-diagonal elements of $\mathbf{L}$. In SpTS, with most $L_{i,j} == 0$, the computation required per row or column is substantially less.

4.3.2 Scalable Problem Definition

Solve the linear system $Ax = b$ where A is a lower triangular matrix and b is a random vector. To generate matrices, a simplified version of the matrix generation scheme described in [60] is followed. A lower triangular matrix with a randomized distribution of non-zeros is generated. Diagonal entries are selected to ensure the the matrix is non-singular. Below diagonal entries of the matrix are distributed in bands according to their distance from the diagonal. The approximate number of nonzeros in each band is specified by the user. The current implementation may cause collisions of matrix entries. The input generator removes such entries and formats the matrix into CSR format. The user should consider that this method may produce slightly fewer nonzeros in the matrix than requested because of collisions. Blocked structures are not considered for this release.

4.3.3 Verification Scheme

To verify the solutions to the kernel, a matrix is generated as described above and a randomized vector x is generated. Matrix multiplication is performed to generate the right hand side vector b. The computed solution is then checked to match the original solution with a maximum difference and l_2 relative error of less than 10^{-10}. Strategies for more sophisticated verification are in consideration for a future release.

4.3.4 Functional Reference Implementation

The following Matlab code solves $Ax = b$ where A is a lower triangular matrix in CSR format:

```
function x = spts(b, n, rowPtr, columnIndices, values)

x(1) = b(1) / values(1);
for rowNum = 2:n
    sum = 0.0;
    for j = rowPtr(rowNum) : rowPtr(rowNum+1) - 2
        sum = sum + values(j) * x(columnIndices(j)) ;
    end
    x(rowNum) = (b(rowNum) - sum) / values(rowPtr(rowNum+1) - 1) ;
end
```

4.3.5 Literature Survey of Optimizations

A survey of optimizations will be included in a subsequent revision of this report.

4.4 Matrix Powers Kernel ($\mathrm{A}^k\mathrm{x}$)

In isolation, the matrix powers ($\mathrm{A}^k\mathrm{x}$) kernel has limited applicability. However, when included in an iterative solver like GMRES, optimized $\mathrm{A}^k\mathrm{x}$ implementations (in conjunction with TSQR) can substantially accelerate performance.

4.4.1 Formal Mathematical Definition

The matrix powers kernel creates an $n \times (k+1)$ matrix $\mathbf{Y}$ whose columns are defined as $\mathbf{Y}_i = \mathbf{A}^{i-1}\vec{x}$. or simply put, $\mathbf{Y} = [\vec{x}, \mathbf{A}\vec{x}, \mathbf{A}^2\vec{x}, \ldots, \mathbf{A}^k\vec{x}]$. Once again, $\mathbf{A}$ is a sparse matrix, $\vec{x}$ is a dense vector, and $\mathbf{Y}$ is a tall skinny dense matrix.

4.4.2 Scalable Problem Definition

We may reuse the finite difference PDE solver (explicit method) discussed in Chapter 5.

4.4.3 Verification Scheme

To verify this problem we discretize the analytic solution in both space and time (each of k steps). We then compare point by point for each power k. In essence this is verifying the time evolution a time $s \cdot \Delta t$ matches the analytic solution for all $s \leq k$. For simplicity we may also only compare the value at $k \cdot \Delta t$

4.4.4 Functional Reference Implementation

The following is a simple matrix powers kernel based on the aforementioned CSR SpMV kernel.

```
for(kk=0;kk<k;k++}{
  nnz=0;
  for(row=0; row<N; row++) {
    double y0 = 0.0;
```

```
      while(nnz<Pointers[row+1]){
        y0 += Value[nnz] * x[kk][Column[nnz]];
        nnz++;
      }
      x[kk+1][row] = y0;
    }
  }
```

4.4.5 Literature Survey of Optimizations

This kernel has only recently garnered performance optimization attention as a result of multicore processors being increasingly memory bandwidth starved. Naively, one computes each matrix power successively. Doing so requires reading the matrix k times. However, one may reorganize the matrix powers kernel in a communication avoiding-fashion, and asymptotically, for certain matrix classes, eliminate all but one read of the matrix. Typically, this optimization requires computation of subsets of several matrix powers. Recent publications in this area have shown significant speedups [81].

4.5 Sparse Matrix-Matrix Multiplication

This kernel and its relevant details will be included in a future release of this testbed.

4.6 Conjugate Gradient (CG)

As discussed in the dense linear algebra chapter, there are many interesting problems that can be formulated as the solution to a system of linear equations. In this chapter, we'll focus on the case where the matrix is sparse ($\mathbf{A} \cdot \vec{x} = \vec{b}$) . Conjugate gradient (CG) is an iterative method for solving systems of linear equations whose matrix is symmetric positive definite. The method proceeds by taking an approximation and the residual ($\vec{r} = \vec{b} - \mathbf{A} \cdot \vec{x}$) and formulating a new solution based on a projection. The result is a sequence of approximations to $\vec{x}$, labeled $x^{(i)}$. For further reading, we direct the reader to [25].

4.6.1 Formal Mathematical Definition

Algorithm 15, (reproduced from [25] with permission), represents a common implementation of the Conjugate Gradient Method. The terms were defined in the previous section.

1: compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2: **for** $i = 1, 2, \ldots$ **do**
3: $\quad \rho_{i-1} = {r^{(i-1)}}^T r^{(i-1)}$
4: $\quad$ **if** $i = 1$ **then**
5: $\quad\quad p^{(1)} = r^{(0)}$
6: $\quad$ **else**
7: $\quad\quad \beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
8: $\quad\quad p^{(i)} = r^{(i-1)} + \beta_{i-1} p^{(i-1)}$
9: $\quad$ **end if**
10: $\quad q^{(i)} = Ap^{(i)}$
11: $\quad \alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
12: $\quad x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
13: $\quad r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
14: $\quad$ check convergence, and continue if necessary.
15: **end for**

Algorithm 15: Conjugate Gradient (CG) iterative method. Reproduced from [25] with permission.

4.6.2 Scalable Problem Definition

Two separate problems are considered.

First is the solution of a Solve the linear system $Ax = b$ where A is a symmetric positive definite matrix and b is a random vector. As for SpTS, to generate matrices, a simplified version of the matrix generation scheme described in [60] is followed. The scheme is modified to produce a symmetric diagonally-dominant matrix. Off diagonal entries are generated first, then diagonal entries are selected to force the matrix to be diagonally dominant.

Second, we may use the heat equation problem described in Chapter 5, but recast it as a sparse problem as we did for SpMV. At each time step, we wish to solve $\mathbf{A} \cdot \vec{u_{t+1}} = \vec{u_t}$ where A is the Helmholtz matrix.

4.6.3 Verification Scheme

The verification scheme for the linear system solve is the same as that used for SpTS. The verification scheme for the PDE solve is the same as that used in the structured grid chapter.

4.6.4 Functional Reference Implementation

The following Matlab code solves $Ax = b$ where A is a symmetric, positive-definite matrix in CSR format. The vector "guess" is an initial guess to the solution of the sytem:

```
function x = conjugateGradient(b, n, rowPtr, columnIndices, values, guess, maxIt)

neps = eps() ;
x = guess;

%compute residual
r = b - spmv(x, n, n, rowPtr, columnIndices, values);
rho = norm(r)^2 ;   % L_2 norm squared

iter = 0 ;
while( sqrt(rho) > neps * norm(b) )

    if( iter == 0 )
        p = r;
    else
        beta = rho / lastRho;
        p = r + beta * p;
    end

    w = spmv(p, n, n, rowPtr, columnIndices, values);
    alpha = rho / (p' * w);
    x = x + alpha * p;
    r = r - alpha * w;
    lastRho = rho;
    rho = norm(r)^2;

    iter = iter + 1;
    if iter > maxIt
        return ;
    end
end
```

4.6.5 Literature Survey of Optimizations

A survey of optimizations will be included in a subsequent revision of this report.

4.7 GMRES

Once again, we wish to solve $\mathbf{A} \cdot \vec{x} = \vec{b}$, but with the caveat that $\mathbf{A}$ is not symmetric (a prerequisite for conjugate gradient). The Generalized Minimal Residual (GMRES) method can solve such unsymmetric systems. However, unlike CG, where only a couple vectors needed to be stored, GMRES requires all previous orthogonal vectors be kept. To minimize the resultant storage and computational impact, a restarted version of GMRES is commonly employed. Thus, GMRES is parameterized by the restart length m. Also crucial to the performance of this algorithm is the selection of a preconditioner matrix. The selection of these parameters is discussed in [25].

4.7.1 Formal Mathematical Definition

For a formal definition of GMRES, we direct the reader to page 18 of [25].

4.7.2 Scalable Problem Definition

As in the Conjugate Gradient section, the problem definition for this section is the solution of a Solve the linear system $Ax = b$. Here A is a general nonsingular matrix and b is a random vector. As before, a simplified version of the matrix generation scheme described in [60] is followed. This version produces general matrices with a banded structure of nonzeros.

4.7.3 Verification Scheme

The verification scheme for the linear system solve is the same as that used for SpTS.

4.7.4 Functional Reference Implementation

The reference implementation may be found in the source code download. For the current release the reference implementation of the kernel is included only in the Matlab version.

4.7.5 Literature Survey of Optimizations

A survey of optimizations will be included in a subsequent revision of this report.

4.8 Sparse LU (SpLU)

This kernel and its relevant details will be included in a future release of this testbed.

Chapter 5

Finite Difference Methods on Structured Grids

In this section we discuss the numerical solutions to partial differential equations. As the title suggests, we've made two restrictions for this "motif": all grids are structured, and the numerical solutions are attained via the finite difference method. We will examine the fundamental partial derivatives, the fundamental differential operators including Laplacian($\nabla^2 u$), Divergence($\nabla \cdot \mathbf{F}$), Gradient(∇u), and Curl($\nabla \times \mathbf{F}$), as well as approaches to solving partial differential equations. For purposes of this chapter, we will examine the 3D heat equation.

5.1 Structured Grid

All PDE's operate on some continuous medium. When providing computational solutions, it is necessary to create a discrete representation of this medium. To that end, we employ a "structured grid" to represent the medium. Concurrently, as we've restricted ourselves to examining solutions via the finite difference method, we represent the continuous medium via uniform sampling in each dimension of the medium.

Given a continuous function $u(x, y, z, t)$ indexed by real numbers, we may construct a sampled grid $u_{i,j,k,t}$ indexed by integers: $u_{i,j,k,t} = u(i \cdot h, j \cdot h, k \cdot h, t \cdot h)$. Such a relationship is visualized in in Figure 5.1. We may replace one of the spacial dimensions with time, but the finite difference method remains the same (5.2).

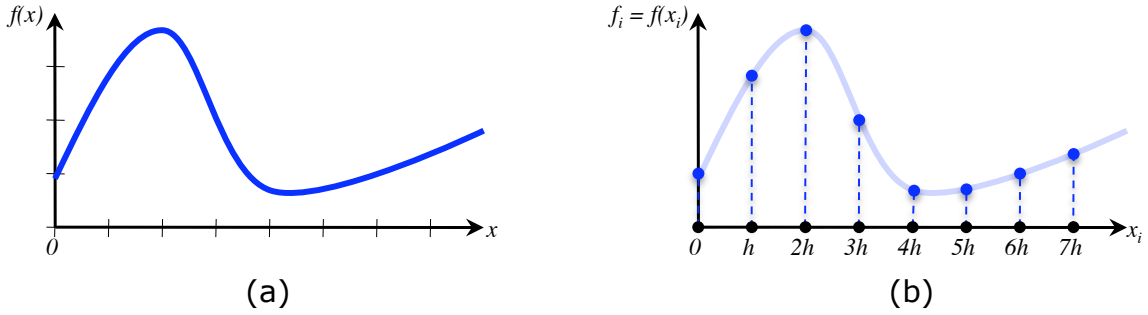

Figure 5.1: Simple 1D example showing uniform sampling (h) of a continuous function $f(x)$. The result is a set of coordinates $x_i = i \cdot h$ and the corresponding values of $f(x_i)$.

We describe each element of the sampled grid as a "point". Note, although not necessary, we have preserved the dimensionality of the continuous function when expressing the sample set. In the same way we sampled a Cartesian grid, we could have sampled a polar or spherical grid (although the finite difference method would have been more complex). Moreover, we correlate the boundary of the medium to the points we sample. That is, the grid is only representative of a specific range in x, y, and z.

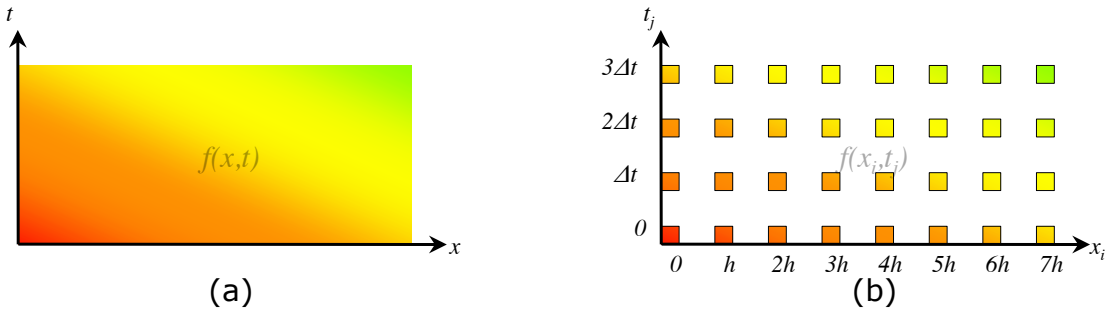

Figure 5.2: (a) Temperature of a continuous medium $f(x,t)$. (b) Space and time are uniformly descritized and $f(x,t)$ is sampled.

It should be noted that the grid need not be restricted to scalar grids (e.g. Temperature), but could easily be a Cartesian vector grid $\mathbf{F}_{i,j,k,t}$. Where $\mathbf{F}_{i,j,k,t}$ is a composed of 3 components: $Fx_{i,j,k,t}$, $Fy_{i,j,k,t}$, and $Fz_{i,j,k,t}$. The most obvious examples of latter include velocity and magnetic fields.

5.2 Linear Algebra Correlation

The resultant structured grids are often stored as an array upon which we apply a stencil operator (described in the following sections). However, we may alternately view the grid as a vector and the stencil as a sparse matrix. In such scenarios, applying the stencil to the grid is simply a sparse matrix-vector multiplication. The matrix captures both the structure (connectivity of points) as well as the functionality (weights) of the stencil operator. In the case of structured grids, the connectivity is trivially calculated and the functionality is isomorphic — the same stencil weights are applied to every point. As such, there is a substantial performance boost when such computations are "matrix-free". This chapter (and by extension, this motif) is premised on this benefit.

5.3 Partial Derivatives

For a 3-dimensional, time-varying continuum $u(x,y,z,t)$, we may define four partial derivatives: $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial u}{\partial z}$, and $\frac{\partial u}{\partial t}$. In essence, these primitives kernels are analogous to the BLAS kernels found in dense linear algebra. Typically, one uses these to construct solvers for complex equations.

5.3.1 Formal Mathematical Definition

The most basic implementation of the finite difference method can use backwards, forwards or central differencing. All produce first order accuracy. That is the accuracy is linearly proportional to the discretization. Table 5.1 lists the 12 combinations. The constants (α, β, etc...) are derived from the discretization that produces the grid. To apply one of these derivatives, one must construct two grids: one for $u_{i,j,k,t}$ and one for its derivative. Then we "sweep" over all i, j, k, and t applying the finite difference method at each point. Often, when calculating spatial derivatives, t is fixed. As such, one typically creates u_{i,j,k,t_0}, and one only iterates over i,j,k.

5.3.2 Scalable Problem Definition

Structured grid calculations are easily scalable. One need only increase the resolution of the sampling. To that end, we may define $U(x,y,z,t)$ in such a way that we may sample it symbolically. For example, if we let $U(x,y,z,t) = x \cdot sin(y) \cdot cos(z) \cdot e^{-t}$, then it is trivial to generate $u_{i,j,k,t} = u(i \cdot h, j \cdot h, k \cdot h, t \cdot h)$ We may then define an interval for the variables: 0...3 for example.

	Backward	Central	Forward
$\frac{\partial}{\partial x}$	$\alpha(u_{i,j,k,t} - u_{i-1,j,k,t})$	$\frac{\alpha}{2}(u_{i+1,j,k,t} - u_{i-1,j,k,t})$	$\alpha(u_{i+1,j,k,t} - u_{i,j,k,t})$
$\frac{\partial}{\partial y}$	$\beta(u_{i,j,k,t} - u_{i,j-1,k,t})$	$\frac{\beta}{2}(u_{i,j+1,k,t} - u_{i,j-1,k,t})$	$\beta(u_{i,j+1,k,t} - u_{i,j,k,t})$
$\frac{\partial}{\partial z}$	$\gamma(u_{i,j,k,t} - u_{i,j,k-1,t})$	$\frac{\gamma}{2}(u_{i,j,k+1,t} - u_{i,j,k-1,t})$	$\gamma(u_{i,j,k+1,t} - u_{i,j,k,t})$
$\frac{\partial}{\partial t}$	$\delta(u_{i,j,k,t} - u_{i,j,k,t-1})$	$\frac{\delta}{2}(u_{i,j,k,t+1} - u_{i,j,k,t-1})$	$\delta(u_{i,j,k,t+1} - u_{i,j,k,t})$

Table 5.1: Partial derivatives via first order backward, central, and forward finite difference method. If the discretization is uniform in x,y,z, then we define $\alpha = \beta = \gamma = \frac{1}{h}$

Verification Scheme

We may symbolically manipulate our scalable problem definition to exactly calculate the partial derivatives (Table 5.2). We may then sample the exactly calculated derivative in the same manner we sampled the original function. The result can be compared point-wise with the result obtained via the finite difference method. Each point should be within the default tolerance of 10^{-5} and the element-wise l_2 relative error should be below the tolerance of 10^{-5}, or user specified values.

The specific test set for this differential operator, shown with the corresponding partial derivatives, is as follows. The parameter τ (called *toughness* in the code) is a user specified difficulty parameter. Larger values of τ result in more numerically challenging functions of the same type. For example, raising the value of τ makes one of the trigonometric functions in the test set have a higher frequency, and thus larger truncation errors when using the finite differencing scheme. For this version of this kernel, there is no dependence on the time variable t.

$u(x,y,z)$	$\frac{\partial}{\partial x}u(x,y,z)$	$\frac{\partial}{\partial y}u(x,y,z)$	$\frac{\partial}{\partial z}u(x,y,z)$
$sin(\tau x)sin(2\tau y)sin(4\tau z)$	$\tau cos(\tau x)sin(2\tau y)sin(4\tau z)$	$2\tau sin(\tau x)cos(2\tau y)sin(4\tau z)$	$4\tau sin(\tau x)sin(2\tau y)cos(4\tau z)$
$\tau xyz + (\tau xyz)^2 + (\tau xyz)^3$	$\tau yz + 2x(\tau yz)^2 + 3x^2(\tau yz)^3$	$\tau xz + 2y(\tau xz)^2 + 3y^2(\tau xz)^3$	$\tau xy + 2z(\tau xyz)^2 + 3z^2(\tau xyz)^3$
$e^{\tau xyz}$	$\tau yze^{\tau xyz}$	$\tau xze^{\tau xyz}$	$\tau xye^{\tau xyz}$

Table 5.2: Test functions for Partial Derivatives and exact solutions to the continuous problem. Sampling will provide a check against the finite difference solution.

5.3.3 Functional Reference Implementation

```
1: for k = i to kmax do
2:   for j = i to jmax do
3:     for i = 1 to imax do
4:       ResultantGrid(i,j,k,t) = Stencil(Grid,i,j,k,t)
5:     end for
6:   end for
7: end for
```

Algorithm 16: Typical structure of a stencil sweep on a 3D grid

5.3.4 Literature Survey of Optimizations

There are many common optimizations associated with structured grid calculations. When coping with the limitations of compilers which may not be able to disambiguate two pointers (the grids), loop unrolling of the spatial loops allows for the expression of more parallelism. Depending on the differential operator and traversal of the grid, the finite difference method can produce a particular reuse pattern. On cache-based architectures, restructuring the grid traversal can will allow for the creation of a working set in cache that can minimize capacity misses. Recent publications [48, 73] have examined these and many other optimizations in the context of CPUs and GPUs.

5.4 Gradient

The gradient (∇u) operates on a scalar field, but unlike the Laplacian, it produces a (Cartesian) vector field that points in the direction of steepest ascent at that point.

5.4.1 Formal Mathematical Definition

The gradient is defined as $\nabla u = \frac{\partial u}{\partial x}\hat{i} + \frac{\partial u}{\partial y}\hat{j} + \frac{\partial u}{\partial z}\hat{k}$ where $\hat{i}$, $\hat{j}$, and $\hat{k}$ point in the x, y, and z directions respectively. To implement the gradient via finite differencing, for each desired time t, we sweep over all i, j, and k performing one of the following finite differences.

Backward	Central	Forward
$\begin{bmatrix} \alpha(u_{i,j,k,t} - u_{i-1,j,k,t}) \\ \beta(u_{i,j,k,t} - u_{i,j-1,k,t}) \\ \gamma(u_{i,j,k,t} - u_{i,j,k-1,t}) \end{bmatrix}$	$\begin{bmatrix} \frac{\alpha}{2}(u_{i+1,j,k,t} - u_{i-1,j,k,t}) \\ \frac{\beta}{2}(u_{i,j+1,k,t} - u_{i,j-1,k,t}) \\ \frac{\gamma}{2}(u_{i,j,k+1,t} - u_{i,j,k-1,t}) \end{bmatrix}$	$\begin{bmatrix} \alpha(u_{i+1,j,k,t} - u_{i,j,k,t}) \\ \beta(u_{i,j+1,k,t} - u_{i,j,k,t}) \\ \gamma(u_{i,j,k+1,t} - u_{i,j,k,t}) \end{bmatrix}$

Table 5.3: Gradient ($\nabla u_{i,j,k,t}$) via first order backward, central, and forward finite difference method. If the discretization is uniform in x,y,z, then we define $\alpha = \beta = \gamma = \frac{1}{h}$

5.4.2 Scalable Problem Definition

One may reuse the scalable problem definition created for the partial derivatives. The analytic solutions are vectors formed by concatenating the three partial derivatives displayed in 5.2.

5.4.3 Verification Scheme

One may reuse the verification scheme created for the partial derivatives.

5.4.4 Functional Reference Implementation

See Algorithm 16.

5.4.5 Literature Survey of Optimizations

see Section 5.3.4

5.5 Divergence

Divergence ($\nabla \cdot F$) measures the outflow or inflow (a scalar quantity) at each point in a vector field.

5.5.1 Formal Mathematical Definition

The divergence is defined as $\nabla F = \frac{\partial}{\partial x} F \cdot \hat{i} + \frac{\partial}{\partial y} F \cdot \hat{j} + \frac{\partial}{\partial z} F \cdot \hat{k}$. To implement the divergence via finite differencing, for each desired time t, we sweep over all i, j, and k performing one of the following finite differences.

Backward	Central	Forward
$\alpha(Fx_{i,j,k,t} - Fx_{i-1,j,k,t}) +$ $\beta(Fy_{i,j,k,t} - Fy_{i,j-1,k,t}) +$ $\gamma(Fz_{i,j,k,t} - Fz_{i,j,k-1,t})$	$\frac{\alpha}{2}(Fx_{i+1,j,k,t} - Fx_{i-1,j,k,t}) +$ $\frac{\beta}{2}(Fy_{i,j+1,k,t} - Fy_{i,j-1,k,t}) +$ $\frac{\gamma}{2}(Fz_{i,j,k+1,t} - Fz_{i,j,k-1,t})$	$\alpha(Fx_{i+1,j,k,t} - Fx_{i,j,k,t}) +$ $\beta(Fy_{i,j+1,k,t} - Fy_{i,j,k,t}) +$ $\gamma(Fz_{i,j,k+1,t} - Fz_{i,j,k,t})$

Table 5.4: Divergence ($\nabla \cdot \mathbf{F}_{i,j,k,t}$) via first order backward, central, and forward finite difference method. If the discretization is uniform in x,y,z, then we define $\alpha = \beta = \gamma = \frac{1}{h}$

5.5.2 Scalable Problem Definition

One may reuse the scalable problem definition created for the partial derivatives.

5.5.3 Verification Scheme

Because divergence operates on a vector field, a modified set of functions is used for testing. Since divergence-free vector fields are common in applications, one such field is included. As above, τ is a toughness parameter which influences the numerical challenges of the problem.

5.5.4 Functional Reference Implementation

See Algorithm 16.

5.5.5 Literature Survey of Optimizations

see Section 5.3.4

5.6 Curl

Where as divergence measured the outflow or inflow at each point in a vector field, curl ($\nabla \times F$) quantifies the rotation of a vector field. As such, curl takes a vector field and produces a vector field as the result.

$u_x(x,y,z)$	$u_y(x,y,z)$	$u_z(x,y,z)$	$\nabla \cdot u$
$sin(\tau x)$	$sin(2\tau y)$	$sin(4\tau z)$	$4\tau cos(\tau x) + 2\tau cos(2\tau y) + 4\tau cos(4\tau z)$
τxy	τxyz	$-\tau yz - \frac{1}{2}\tau xz^2$	0
$e^{\tau x}$	$e^{2\tau y}$	$e^{4\tau z}$	$\tau(e^{\tau x} + 2e^{2\tau y} + 4e^{4\tau z})$

Table 5.5: Test functions for Divergence and exact solutions to the continuous problem. Sampling will provide a check against the finite difference solution.

5.6.1 Formal Mathematical Definition

The divergence is defined as $\nabla \times F = (\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z})\hat{i} + (\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x})\hat{j} + (\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y})\hat{k}$. To implement the curl via finite differencing, for each desired time t, we sweep over all i, j, and k performing one of the following finite differences.

Backward difference:

$$\nabla \times \mathbf{F}_{i,j,k,t} = \begin{bmatrix} \beta(Fz_{i,j,k} - Fz_{i,j-1,k}) - \gamma(Fy_{i,j,k} - Fy_{i,j,k-1}) \\ \gamma(Fx_{i,j,k} - Fx_{i,j,k-1}) - \alpha(Fz_{i,j,k} - Fz_{i-1,j,k}) \\ \alpha(Fy_{i,j,k} - Fy_{i-1,j,k}) - \beta(Fx_{i,j,k} - Fx_{i,j-1,k}) \end{bmatrix}$$

Central difference:

$$\nabla \times \mathbf{F}_{i,j,k,t} = \begin{bmatrix} \frac{\beta}{2}(Fz_{i,j+1,k} - Fz_{i,j-1,k}) - \frac{\gamma}{2}(Fy_{i,j,k+1} - Fy_{i,j,k-1}) \\ \frac{\gamma}{2}(Fx_{i,j,k+1} - Fx_{i,j,k-1}) - \frac{\alpha}{2}(Fz_{i+1,j,k} - Fz_{i-1,j,k}) \\ \frac{\alpha}{2}(Fy_{i+1,j,k} - Fy_{i-1,j,k}) - \frac{\beta}{2}(Fx_{i,j+1,k} - Fx_{i,j-1,k}) \end{bmatrix}$$

Forward difference:

$$\nabla \times \mathbf{F}_{i,j,k,t} = \begin{bmatrix} \beta(Fz_{i,j+1,k} - Fz_{i,j,k}) - \gamma(Fy_{i,j,k+1} - Fy_{i,j,k}) \\ \gamma(Fx_{i,j,k+1} - Fx_{i,j,k}) - \alpha(Fz_{i+1,j,k} - Fz_{i,j,k}) \\ \alpha(Fy_{i+1,j,k} - Fy_{i,j,k}) - \beta(Fx_{i,j+1,k} - Fx_{i,j,k}) \end{bmatrix}$$

When the discretization is uniform in x,y,z, then we define $\alpha = \beta = \gamma = \frac{1}{h}$

5.6.2 Scalable Problem Definition

One may reuse the scalable problem definition created for the partial derivatives.

5.6.3 Verification Scheme

Because curl operates on a vector field and outputs a vector field, a modified set of functions is used for testing. Since conservative vector fields are common in applications, one such field is included. As above, τ is a toughness parameter which influences the numerical challenges of the problem.

5.6.4 Functional Reference Implementation

See Algorithm 16.

5.6.5 Literature Survey of Optimizations

see Section 5.3.4

$u_x(x,y,z)$	$u_y(x,y,z)$	$u_z(x,y,z)$	$\nabla \times u$
$sin(\tau z)$	$sin(2\tau x)$	$sin(4\tau y)$	$(4\tau cos(4\tau y), \tau cos(\tau z), 2\tau cos(2x))$
$\tau x^2 yz$	$\frac{1}{3}\tau z^3$	$\frac{1}{3}\tau y^3$	$(0,0,0)$
$e^{\tau z}$	$e^{2\tau x}$	$e^{4\tau y}$	$(4\tau e^{4\tau y}, \tau e^{\tau z}, 2\tau e^{2\tau x})$

Table 5.6: Test functions for Curl and exact solutions to the continuous problem. Sampling will provide a check against the finite difference solution.

5.7 Laplacian

The laplacian operator ($\nabla^2 u$) is commonly conceptualized as the divergence of the gradient ($\nabla \cdot \nabla u$) or alternately as $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$. This operator is commonly found in a wide variety of partial differential equations.

5.7.1 Formal Mathematical Definition

As with the partial derivatives, we sweep over all i, j, and k performing the laplacian operator at each point. Using the central finite difference method the Laplacian is simply:
$\nabla^2 u_{i,j,k,t} = \alpha(u_{i-1,j,k,t} + u_{i+1,j,k,t}) + \beta(u_{i,j-1,k,t} + u_{i,j+1,k,t}) + \gamma(u_{i,j,k-1,t} + u_{i,j,k+1,t}) - 2(\alpha + \beta + \gamma)u_{i,j,k,t}$
Please note, if the discretization is uniform in x,y,z, then we redefine $\alpha = \beta = \gamma = \frac{1}{h^2}$.

5.7.2 Scalable Problem Definition

One may reuse the scalable problem definition created for the partial derivatives.

5.7.3 Verification Scheme

One may reuse the verification scheme created for the partial derivatives. The analytic solutions to the evaluating the Laplacian of each function is in the table below.

$u(x,y,z)$	$\nabla^2 u$
$sin(\tau x)sin(2\tau y)sin(4\tau z)$	$-21\tau(sin(\tau x)sin(2\tau y)sin(4\tau z))$
$\tau xyz + (\tau xyz)^2 + (\tau xyz)^3$	$2\tau^2(1 + 3\tau xyz)(x^2y^2 + x^2z^2 + y^2z^2)$
$e^{\tau xyz}$	$\tau^2(x^2y^2 + x^2z^2 + y^2z^2)e^{\tau xyz}$

Table 5.7: Test functions for Laplacian and exact solutions to the continuous problem. Sampling will provide a check against the finite difference solution.

5.7.4 Functional Reference Implementation

See Algorithm 16.

5.7.5 Literature Survey of Optimizations

see Section 5.3.4

5.8 Solve Discretized PDE (forward/explicit)

Partial differential equations form the core of many scientific computing applications. Broadly speaking, they are built upon the differential operators discussed in the preceding sections. The following sections use the canonical heat equation ($\frac{\partial u}{\partial t} = C\nabla^2 u$) as a motivating example to explore alternate approaches to solving this parabolic PDE. We commence with an explicit method for evaluation of this PDE.

5.8.1 Formal Mathematical Definition

In the explicit solution, we use the forward difference for the time derivative that we discussed in Section 5.3, and the central difference laplacian from Section 5.7. Simply put, to follow the time evolution of this PDE, we evaluate $\vec{y} = \mathbf{A} \cdot \vec{x}$ where $\mathbf{A}$ is $dt\nabla^2 - \mathbf{I}$, $\vec{x} = Grid(i, j, k, t)$, and $\vec{y} = Grid(i, j, k, t + 1)$. Clearly, we must start with the initial conditions (u_{i,j,k,t_0}), but simply replace the matrix-vector product with a stencil sweep.

Alternately, the grid may be transfered to a vector and matrix-vector products may still be used. This scheme is numerically identical to using stencil sweeps, but different in data-structure. The matrix version requires the matrix to be read multiple times, rather than only two constants for the stencil method. As such memory traffic for the matrix-based is higher and thus not preferred. Both the stencil- and matrix-based versions are included in the reference implementations.

5.8.2 Scalable Problem Definition

For our test problem we select the region of interest to be the three dimensional unit cube and set $C = 1$. Initial conditions are $u(x, y, z, 0) = sin(2\pi x)sin(2\pi y)sin(2\pi z)$. Boundary conditions are homogeneous Dirichlet boundary conditions, that is, zero on all boundaries for all times t. Like the previous sections, we may choose whatever spatial discretization we desire. The time discretization may also be chosen to any desired value. The explicit method to solve such an equation is only conditionally stable, so the user should select the spatial and time discretization such that $dt/h^2 < 1/6$ to satisfy the Courant-Friedrichs-Lewy condition for numerical stability. See [88], chapter 20 for further description of this condition.

5.8.3 Verification Scheme

The analytic solution to this equation is $u(x, y, z, t) = e^{-12\pi^2 t}sin(2\pi x)sin(2\pi y)sin(2\pi z)$. Given this exact solution, one may sample it using the finite difference method, and then by the same scheme as we verified the derivatives, we may verify the heat equation.

5.8.4 Functional Reference Implementation

Algorithm 16 is used to evaluate the operator at each timestep. In the matrix based version, the evaluation of the differential operator at all timesteps can be evaulated with a single call to the Matrix-Powers kernel. See Chapter 4.

5.8.5 Literature Survey of Optimizations

See Section 5.3.4. Additionally, temporal blocking has become a valuable approach to further improving locality [102].

5.9 Solve Discretized PDE (backdward/implicit) using CG

Rather than the forward difference time derivative discussed in the previous section, we may solve the same equation using a backwards time derivative. The result is that at each time step, we must solve a system of linear equations in the form $\mathbf{A} \cdot \vec{x} = \vec{b}$ where $\mathbf{A}$ is the Helmholtz operator $(-dt\nabla^2 + \mathbf{I})$, $\vec{x} = Grid(i, j, k, t + 1)$, and $\vec{b} = Grid(i, j, k, t)$. We may solve this system of linear equations using the conjugate gradient method discussed in Chapter 4 with the caveat that the matrix-vector products may be replaced with stencil sweeps.

As in the explicit method, the grid may be transferred to a vector and matrix-vector products may still be used inside the CG solve. As above, this adds memory traffic and is not preferred. Both versions are included here as well.

5.9.1 Formal Mathematical Definition

Each timestep in Chapter 4.

5.9.2 Scalable Problem Definition

One may use the same problem generation scheme as the explicit solution (see Section 5.8).

5.9.3 Verification Scheme

One may use the same verification scheme as the explicit solution (see Section 5.8).

5.9.4 Functional Reference Implementation

See Section 4.6.4 for discussion of the reference implementation of CG. See the source code download for details specific to the heat equation solve, including replacing the SpMV operation with stencil operations.

5.9.5 Literature Survey of Optimizations

A survey of optimizations will be included in a subsequent revision of this report.

5.10 Solve Discretized PDE (backward/implicit) using multigrid

Rather than using conjugate gradient to solve the implicit PDE of the previous section, we may use a multigrid method. Multigrid methods use a set of grids, discretized to different widths, to efficiently solve the linear systems. They are effective for the system defined for the implicit method for the Heat Equation, as well as for Poisson's equation and others. The following description is taken from [51] with permission:

"In contrast to other iterative schemes that we have discussed so far, multigrid's convergence rate is *independent* of the problem size N, instead of slowing down for larger problems. As a consequence it can solve problems with n unknowns in $O(n)$ time or for a constant amount of work per unknown. This is optimal, modulo the (modest) constant hidden inside the $O(\cdot)$...

Multigrid uses coarse grids to do *divide-and-conquer* in two related senses. First, it obtains an initial solution for an N-by-N grid by using an (N/2)-by-(N/2) grid as an approximation, taking every other grid point from the N-by-N grid. The coarser (N/2)-by-(N/2) is in turn approximated by an (N/4)-by-(N/4), and so on recursively. The second way multigrid uses divide-and-conquer is in the *frequency domain*. This requires us to think of the error as a sum of eigenvectors, or sine-curves of different frequencies. Then, intuitively, the work that we do on a particular grid will attenuate the error in half of the frequency components not attenuated on the coarser grids. In particular, the work performed on a particular grid–averaging the solution at each grid point with its neighbors, a variation of Jacobi's method–makes the solution smoother, which is equivalent to getting rid of the high-frequency error."

5.10.1 Formal Mathematical Definition

Multigrid solvers rely on the following three core operations.

First, a relaxation operator. This is usually an iterative method which solves the linear system. Standard Gauss-Seidel is used here, which is an averaging with nearest neighbors weighted according to the associated differential operators. This is but one standard method which dampens the high-frequecy error as discussed above.

Second, a restriction operator. This operation transfers the current problem from an N^3 grid (because our model problem is in three dimensions) to an $(N/2)^3$ grid. The restriction operator selected is the injection operator, with which the coarse grid takes the value of the fine grid at the equivalent point in space.

Third, an interpolation operator. This operation transfers the current problem from an $(N/2)^3$ grid to an N^3 grid. The interpolation operator used performs a weighted, nearest neighbor average of points. For the three dimensional model problem, this scheme first copies existing points, then averages to fill in rows, then columns, then entire planes. Each average is computed between two nearest neighbor points, but for the columns and planes, their nearest neighbors are themselves averages, making the average in effect of increasingly more points of the coarse grid.

These methods are then put together in the following way, called the V-Cycle:

1: **for** $i = 2$ to $maxDepth$ **do**
2: Apply the relaxation operator to the current grid $i - 1$.
3: Apply the restriction operator to the residual and right hand side.
4: Assign the restricted residual and right hand side to the next, more coarse, grid i.
5: **end for**
6: Perform a full solve on the remaining residual at the coarsest level $maxDepth$.
7: **for** $i = maxDepth$ to 2 **do**
8: Apply the interpolation operator on grid i to get the fine grid and right hand side.
9: Add the fine grained approximation to the residual to the existing next, more fine, grid $i - 1$.
10: Apply the relaxation operator to the next, more fine, grid $i - 1$.
11: **end for**

Algorithm 17: Simplified description of V-Cycle

Both [51] and [36] contain more detail and explicit instructions for implementing such methods, along with information on solving a variety of differential equations and applications.

5.10.2 Scalable Problem Definition

One may use the same verification scheme as the explicit solution.

5.10.3 Verification Scheme

One may use the same verification scheme as the explicit solution.

5.10.4 Functional Reference Implementation

Following are simplified reference implementations of the restriction and interpolation operators. For the reference implementation of the relaxation scheme, Gauss-Seidel, and the V-Cycle see the source download.

```
function coarse = fineToCoarse(coarse, fine)
%    Injection operator from fine grid to coarse grid.
%    If coarse grid is size n in each dimension, fine grid will be of size 2*n-1.
%
%    Input:
%     grid coarse      Preallocated 3D coarse array of length (n + 1) / 2 in each dimension
%     grid fine        3D fine array of length n in each dimension (which is computed here)
%
```

```
%     Output:
%     grid coarse      3D array formed using injection operator from fine grid

    [n n n] = size(fine) ;
    coarse = fine(1:2:n, 1:2:n, 1:2:n);
end

function fine = coarseToFine(coarse, fine)
%    Converts a coarse grid to a fine grid.
%    Uses an averaging operator.
%    If coarse grid is size n in each dimension, fine grid will be of size 2*n-1.
%
%   Input:
%     grid coarse     3D coarse array of length n in each dimension (which is computed here)
%     grid fine       Preallocated 3D fine array n of length 2*n - 1 in each dimension
%
%     Output:
%     grid fine       3D array that is the linear interpolation of the coarse grid

    [n n n] = size(coarse);
    fineN = 2*n-1;

    %copy existing points first
    for i=1:n
        for j = 1:n
            for k = 1:n
                fine(2*i - 1, 2*j - 1, 2*k - 1) = coarse(i,j,k);
            end
        end
    end

    % average odd numbered columns in odd numbered planes in x direction
    for i=2:2:fineN-1
        for j = 1:2:fineN
            for k = 1:2:fineN
                fine(i,j,k) = .5 * (fine(i-1,j,k) + fine(i+1,j,k));
            end
        end
    end

    % average even numbered columns in odd numbered planes in y direction
    for i=1:fineN
        for j = 2:2:fineN-1
            for k = 1:2:fineN
                fine(i,j,k) = .5 * (fine(i,j-1,k) + fine(i,j+1,k));
            end
        end
    end

    % average entire even numbered planes in z direction
    for i=1:fineN
        for j = 1:fineN
            for k = 2:2:fineN-1
                fine(i,j,k) = .5 * (fine(i,j,k-1) + fine(i,j,k+1));
            end
        end
    end
end
```

5.10.5 Literature Survey of Optimizations

Both [51] and [36] contain many more details about the performance and applicability of multigrid methods. Both include extensive lists of references including many on optimizations.

Chapter 6

Finite Volume Methods on Structured Grids

In this section we discuss the numerical solutions to partial differential equations. As the title suggests, we've made two restrictions for this "motif": all grids are structured, and the numerical solutions are attained via the finite volume method. We will examine the fundamental partial derivatives, the fundamental differential operators including Laplacian($\nabla^2 u$), Divergence($\nabla \cdot \mathbf{F}$), Gradient(∇u), and Curl($\nabla \times \mathbf{F}$), as well as approaches to solving partial differential equations.

These kernels will be included in a future release of this testbed.

Chapter 7

Spectral Methods

Spectral methods, which are schemes based, one way or another, on the fast Fourier transform (FFT), are used heavily in speech processing and, more generally, in the larger world of digital signal processing. Cell phones are just one example of a modern consumer technology that employs these methods. However, spectral methods and FFTs are useful in an even broader range of scientific computations that have little to do with signals, such as convolutions and Poisson partial differential equations.

7.1 Lexicon/Terminology for Motif

Typically, a sampling method is performed on a continuous function. The result is a discrete data set (vector) upon which transformations are performed.

7.2 1D Fast Fourier Transform (FFT)

The 1D Fast Fourier Transform (FFT) in is merely a fast algorithm for evaluating the discrete Fourier transform (DFT) on a 1D input.

7.2.1 Formal Mathematical Definition

On a one-dimensional complex vector $(x_0, x_1, \cdots x_{n-1})$ of length n, the DFT is defined as

$$\begin{aligned} X_k &= \sum_{j=0}^{n-1} x_j e^{-2\pi i j k/n}, \quad 0 \le k < n \\ &= \sum_{j=0}^{n-1} x_j \left[\cos(-2\pi i j k/n) + i \sin(-2\pi i j k/n)\right], \quad 0 \le k < n. \end{aligned} \tag{7.1}$$

If one were to compute an n-point 1-D DFT using (7.1), the total cost would be $2n^2$ operations, even if we presume that all the exponential factors had been precomputed. Using any of the known FFT algorithms reduces this cost to $5n \log_2 n$, at least in the case that n is a power of two. If x is purely real, then there exist simple transformations that permit the DFT to be computed on a complex vector of length $n/2$ (and these transformations also work if the DFT is being computed with an FFT). See [21]

The above formulas define what are normally referred to as "forward" DFTs. The formulas for the "inverse" DFT are the same as the above, except the sign of the exponent is positive rather than negative, and a $1/n$ factor is placed in front of the summation sign.

7.2.2 Scalable Problem Definition

Compute a 1-D complex-to-complex FFT of as large a power-of-two size as possible on 64-bit IEEE data. The initial data is generated by a pseudo-random number generator of your choice.

7.2.3 Verification Scheme

The result of forward and inverse FFT must match the original data with a RMS error of no greater than 10^{-10}. More rigorous tests are provided subsequent kernels which use the FFT as a sub-kernel.

7.2.4 Functional Reference Implementation

There are many versions of the FFT algorithm. They go by names such as "decimation in time," "decimation in frequency," "Gentleman-Sande FFT," "Stockham FFT," "Pease's FFT," and others. One challenge in computing the FFT derives from the fact that when computing an FFT on data dimensioned by a power of two (by far the most common kind), most FFT algorithms access data by power-of-two strides. Such accesses typically result in poor performance both on vector computers and also on cache-memory systems. Another challenge is in performing the "bit-reversal" permutation required in some FFTs.

The best scheme for many cache memory systems is known as the "four-step FFT," which in effect performs the 1-D FFT as a 2-D FFT (except for some exponential factors) [20]. Although it has been "discovered" several times in the past decade or two, this algorithm was actually first presented over forty years ago in a paper by Gentleman and Sande [63, pg. 569]. This scheme is as follows. Let $n = n_1 n_2$ be the size of the transform. Note that n does not necessarily need to be a power of two, although as with all FFT algorithms it is most efficient when n is a power of two. This particular algorithm is most efficient when n_1 and n_2 are chosen as close as possible to $\sqrt{n}$. In the following and hereafter, matrices will be assumed to be stored in memory columnwise as in the Fortran language. The FFT of n complex input data values can then be obtained by performing the following four steps:

1. Perform n_1 simultaneous n_2-point FFTs on the input data considered as a $n_1 \times n_2$ complex matrix.
2. Multiply the resulting data, considered as a $n_1 \times n_2$ matrix A_{jk}, by $e^{-2\pi ijk/n}$.
3. Transpose the resulting $n_1 \times n_2$ complex matrix into a $n_2 \times n_1$ matrix.
4. Perform n_2 simultaneous n_1-point FFTs on the resulting $n_2 \times n_1$ matrix.

The individual 1-D FFTs, which typically fit in a cache memory, may in turn be performed by any of several schemes, such as Stockham's algorithm [19, 99].

Several important features of this algorithm should be noted: first of all, note that both of the simultaneous FFT steps can be performed using exclusively unit stride data access, which is optimal on virtually any computer system. Secondly, this algorithm produces an ordered transform (provided the simultaneous FFTs are ordered). It is not necessary to perform a bit reversal permutation, which is inefficient on many advanced computer systems. Finally, note that only three passes through the external data set are required to perform this algorithm. The second step can be performed on a block of data after the first step, before it is returned to memory.

A good reference for the FFT is Van Loan's book [99].

7.2.5 Literature Survey of Optimizations

AMD, Apple, Cray and IBM, among other vendors, offer highly tuned FFT libraries [5, 44, 70]. There are also some excellent self-tuning FFT libraries, notably the FFTW [58] library, which run on a wide variety of modern-day systems. These libraries make it simple for scientists to achieve very high performance on FFTs. For example, on a single core of a 3.2 GHz Xeon-based MacPro, FFTW performs a 2^{24}-point complex 64-bit FFT at 1.28 Gflop/s, and the Apple FFT package runs this calculation at 2.83 Gflop/s [44].

A good reference for the DFT is the book by Briggs and Henson [35]. A good reference for the FFT is Van Loan's book [99].

7.3 3D Fast Fourier Transform (FFT)

The 3D FFT is a simple extension of the previous section to a 3D data set. This method can be modified to become an efficient implementation of a 1D FFT on very large datasets.

7.3.1 Formal Mathematical Definition

The DFT of a three-dimensional array ($x_{j,k,l}$, $0 \le j < n_1$, $0 \le k < n_2$, $0 \le l < n_3$) of size $n = n_1 n_2 n_3$ is defined as

$$X_{p,q,r} = \sum_{l=0}^{n_3} \sum_{k=0}^{n_2} \sum_{j=0}^{n_1} x_{j,k,l} e^{-2\pi ijklpqr/n}, \quad 0 \le p < n_1, 0 \le q < n_2, 0 \le r < n_3. \tag{7.2}$$

This can be computed by taking the FFT along each index in sequence.

7.3.2 Scalable Problem Definition

Compute a complex-to-complex FFT a three dimensional cube where each dimension is as large a power-of-two size as possible. Use the same random number generators as the 1-D case.

7.3.3 Verification Scheme

The result of forward and inverse FFT must match the original data with a RMS error of no greater than 10^{-10}.

7.3.4 Functional Reference Implementation

The reference implementation provided uses a straight forward organization and calls the associated one dimensional routines. Data for each individual FFT is copied into a contiguous section of memory before the one dimensional routine is called.

7.3.5 Literature Survey of Optimizations

See previous section.

7.4 Perform Convolution via FFT

Convolutions arise in a surprisingly large and diverse set of applications. For example, consider the multiplication of two high-precision integers $x = (x_0, x_1, \cdots, x_{n-1})$ and $y = (y_0, y_1, \cdots, y_{n-1})$, where each x_i is an integer in the range $[0, 2^b - 1]$, and collectively represent the bn-long binary expansion of x, and similarly for y. Note that from elementary school arithmetic, if we defer for the time being the release of carries, then the $2n$-long multiplication "pyramid" z resulting from multiplying x and y is merely

$$z = (x_0 y_0, x_0 y_1 + x_1 y_0, x_0 y_2 + x_1 y_1 + x_2 y_0, \cdots, x_{n-2} y_{n-1} + x_{n-1} y_{n-2}, x_{n-1} y_{n-1}). \tag{7.3}$$

But this is nothing more than the "linear" or "acyclic" convolution $A(x, y)$ of the vectors x and y, which can easily be written as a circular convolution by first extending x and y to length $2n$ by padding with zeroes, and then writing

$$z_k = \sum_{j=0}^{2n-1} x_j y_{k-j}, \quad 0 \le k < 2n \tag{7.4}$$

where the subscript $k - j$ is interpreted as $k - j + 2n$ if $k - j < 0$.

A related concept is the *correlation* of two vectors:

$$z_k = \sum_{j=0}^{n-1} x_j y_{j+k}, \quad 0 \le k < n \tag{7.5}$$

where the subscript $j + k$ is interpreted as $j + k - n$ if $j + k > n$.

7.4.1 Formal Mathematical Definition

As mentioned above, the *circular* convolution $C(x, y)$ of two n-long vectors $x = (x_0, x_1, \cdots, x_{n-1})$ and $y = (y_0, y_1, \cdots, y_{n-1})$ is defined as the vector

$$C_k(x, y) = \sum_{j=0}^{n-1} x_j y_{k-j}, \quad 0 \le k < n \tag{7.6}$$

where the subscript $k - j$ is interpreted as $k - j + n$ if $k - j < 0$. The *linear* or *acyclic* convolution $A(x, y)$ of x and y is defined as the vector

$$A_k(x, y) = \sum_{j=0}^{2n-1} x_j y_{k-j}, \quad 0 \le k < 2n \tag{7.7}$$

where the subscript $k - j$ is interpreted as $k - j + 2n$ if $k - j < 0$, and where it is understood that $x_k = y_k = 0$ for $n \le k < 2n$. In other words, if we "pad" the n-long input vectors x and y with n zeroes each, then the acyclic convolution of x and y is simply the $2n$-long circular convolution of the extended x and y vectors.

The reason that convolutions are of interest in this context is that FFTs can be used to dramatically accelerate the computation of a convolution. In particular, if we denote by $F(x)$ the forward discrete Fourier transform of x and $F^{-1}(x)$ the inverse discrete Fourier transform of x, then the convolution theorem says that that circular convolution $C(x, y)$ is

$$C(x, y) = F^{-1}(F(x)F(y)). \tag{7.8}$$

As before, if the input vectors are purely real rather than complex, simple transformations exist that permit these operations to be done twice as fast.

7.4.2 Scalable Problem Definition

Compute the 1-D real acyclic convolution of two large power-of-two sized vectors, where the initial data are pseudorandomly generated integers in the range $[0, 2^b - 1]$ for some integer b, where b is chosen as large as possible so that the final acyclic convolution data are exactly represented as 64-bit IEEE values (this normally means that $2b + \log_2 n \le 53$).

7.4.3 Verification Scheme

The accuracy is verified by two comparisons. When the convolution of integer data is computed, output data should be integers within a rounding tolerance, so that property is checked on the full output data. One potential vulnerability to this scheme is to use data that is too large to be exactly represented in double precision. As such, the bound above is checked by the verification scheme, and exceeding the bound will result in failure.

To ensure that the answer is not simply random integers, the first ten elements of the convolution are calculated naively and compared with the output of the FFT based algorithm. Any discrepancy here causes failure.

7.4.4 Functional Reference Implementation

The implementation provided uses real-to-complex and complex-to-real ffts for evaluating the convolution. The details of packing data for these algorithms are available in [21].

7.4.5 Literature Survey of Optimizations

See previous section.

7.5 Solve Discretized PDE via FFT

In this section, we solve the dicretized Heat Equation as described in Chapter 5 by transforming it into the frequency domain, performing the requisite operations, and then returning to the spatial domain. This method differs from previous methods to numerically solve the heat equation in that no nearest-neighbor or finite-differencing operations are performed.

7.5.1 Formal Mathematical Definition

The following description is modified from [23]. Select the problem sizes such that the number of internal points of the grid in each dimension m, n and p are all powers of two. In particular, parameter values and names are modified and information that is irrelevant to this testbed is omitted.

"Consider the PDE

$$\frac{\partial u(x,t)}{\partial t} = \nabla^2 u(x,t)$$

where x is a position in 3-dimensional space. When a Fourier transform is applied to each side, this equation becomes

$$\frac{\partial v(z,t)}{\partial t} = -4\pi^2 |z|^2 v(z,t)$$

where $v(z,t)$ is the Fourier transform of $u(x,t)$. This has the solution

$$v(z,t) = e^{-4\pi^2|z|^2 t} v(z,0)$$

Now consider the discrete version of the original PDE. Following the above, it can be solved by computing the forward 3-D discrete Fourier transform (DFT) of the original state array $u(x,0)$, multiplying the results by certain exponentials, and then performing an inverse 3-D DFT.

Compute the forward 3-D DFT of U, using a 3-D fast Fourier transform (FFT) routine, and call the result V. Set $t = dt$. Then compute

$$W_{i,j,k} = e^{-4\pi^2(\bar{i}^2+\bar{j}^2+\bar{k}^2)t} V_{i,j,k}$$

where $\bar{i}$ is defined as i for $0 \le i < m/2$ and $i - m$ for $m/2 \le i < m$. The indices $\bar{j}$ and $\bar{k}$ are similarly defined with n and p. Then compute an inverse 3-D DFT on W, using a 3-D FFT routine, and call the result the array X. Increment t by dt. Then repeat the above process, from the computation of W through the incrementing of t, until the step $t = tSteps$ has been completed. The V array and the array of exponential terms for $t = dt$ need only be computed once. Note that the array of exponential terms for $t > 1$ can be obtained as the t-th power of the array for $t = dt$."

7.5.2 Scalable Problem Definition

The problem definition is the same as section 5.8.

7.5.3 Verification Scheme

The verification scheme is the same as section 5.8

7.5.4 Functional Reference Implementation

The following reference implementation in Matlab is a simplification of the reference implementation in the source code. Specifically, it uses the library routine for the 3D FFT, whereas the code gives the user a choice between library or our implementation.

```
function soln = solve3DHeatSpectral(dt, tSteps, l, h)
%
%   Spectral method for Heat Equation.
%
%   input:
%      double dt          Time step for solve
%      int tSteps         Total number of timesteps to perform
%      double l           Width of cube, same on all dimensions
%      double h           Grid height
%
%   output:
%      4D real soln      Solution to heat equation. indexed (x,y,z,t)

f = @(x,y,z) sin(2*pi*x) * sin(2*pi*y) * sin(2*pi*z);

N = l/h - 1;
M = tSteps ;
soln = zeros(N,N,N,M); % don't include the ghost zones here

%set initial conditions
initTemp = initGrid3D(f, h, l, l, l);

%remove ghost zones
initTemp = initTemp(2:N+1, 2:N+1, 2:N+1);
soln(:,:,:,1) = initTemp(:,:,:);

initFFT = fftn(soln(:,:,:,1));

    %compute the exponential lookup table for advancing data
    twiddles = zeros(N,N,N);

    for k = 0:N-1
        for j = 0:N-1
            for i = 0:N-1

                if i < N/2
                    iBar = i;
                else
                    iBar = i-N;
                end

                if j < N/2
                    jBar = j;
                else
                    jBar = j-N;
                end

                if k < N/2
                    kBar = k;
                else
                    kBar = k-N;
                end

                twiddles(i+1, j+1, k+1) = exp(-4*pi^2 * (iBar^2 + jBar^2 + kBar^2));
            end
        end
    end

t = 0;
```

```
for j = 2:tSteps
   t = t + dt;
   soln(:,:,:,j) = (twiddles .^ t) .* initFFT ;
   soln(:,:,:,j) = ifftn(soln(:,:,:,j)) ;
end
```

7.5.5 Literature Survey of Optimizations

See previous section.

Chapter 8

Particle-Particle Methods

Particle methods simulate the time evolution of discrete entities (particles). These particles can anything be atomic nuclei or molecules in a gas, to planets or superclusters of galaxies.

8.1 Lexicon/Terminology for Motif

The particles may have physical characteristics like mass, charge, position, velocity, spin, etc... and interact with each other via a force function based on the these characteristics. Once the force has been calculated, particles may be accelerated and moved the appropriate distance given the time step. Typical particle simulations iterate through time in a series of force calculation and push phases.

8.2 2D/3D N^2 Direct

This kernel models the 2D/3D direct all-to-all particle simulation by calculating forces, accelerating particles, and moving them.

8.2.1 Formal Mathematical Definition

Given a system of N particles with masses $\{m_1, \ldots, m_N\}$, Cartesian positions $\{\vec{x_1}, \ldots, \vec{x_N}\}$ and Cartesian velocities $\{\vec{v_1}, \ldots, \vec{v_N}\}$, we wish to model the time evolution of such particles given the introduction of an interaction force $\{\vec{F_1}, \ldots, \vec{F_N}\}$ applied to each particle. Two forces are included. The first is a simple gravitational force:

$$\vec{F_i} = \sum_{\substack{j=1 \\ j \neq i}}^{N} -G \frac{m_i m_j}{|\vec{x_j} - \vec{x_i}|^3} (\vec{x_j} - \vec{x_i})$$

As the code iterates through time, it first calculates all $\vec{F_i}$, then uses them to accelerate and move particles: $\vec{v_i} = \vec{v_i} + \frac{\vec{F_i}}{m_i} \Delta t$, $\vec{x_i} = \vec{x_i} + \vec{v_i} \Delta t$.

The second is a repulsive force which is zero outside a certain range:

$$\vec{F_i} = \sum_{\substack{j=1 \\ j \neq i}}^{N} \alpha_{i,j} (\vec{x_j} - \vec{x_i})$$

where $\alpha_{i,j}$ is defined according to the following parameters:

$$c = 0.01, \quad min\_r = c/100, \quad r2 = max(||\vec{x_j} - \vec{x_i}||_2^2, min\_r^2), \quad r = \sqrt{r2}$$

$$\alpha_{i,j} = \begin{cases} 0 & \text{if } ||\vec{x_j} - \vec{x_i}||_2^2 > c^2 \\ \frac{1 - \frac{c}{r}}{r2} & \text{otherwise} \end{cases}$$

8.2.2 Scalable Problem Definition

Set particles randomly in the desired region, with random initial velocities. Move the system through the desired number of time steps as described in the previous section.

8.2.3 Verification Scheme

Since full solutions to the n-body problem are not known, this kernel is difficult to verify analytically. However, coding for the naïve method is straightforward and visualizations for associated data are available for a heuristic check. One such visualizer is available at [46]. The two dimensional reference implementations output data in a format compatible with this visualizer.

8.2.4 Functional Reference Implementation

We direct the reader to the source code included with this report.

8.2.5 Literature Survey of Optimizations

A survey of optimizations will be included in a subsequent revision of this report.

8.3 2D/3D N^2 Direct (with cut-off)

In many physical simulations, some forces are near field (they can be ignored at sufficiently long distances) while others are far field (cannot be ignored). This kernel assumes the force can be ignored when the particle is outside the specified cutoff distance. As such, this cut-off in distance would allow one to reduce the numerical complexity of many calculations from $O(N^2)$ by a factor of around the size of the region over cutoff distance c to the size of the region over c^3. This reduction is highly dependent on the value of cutoff distance c, the spatial layout of the particles, the size of the region and the dimension of the problem.

The user should be aware that this method is usually not accurate enough for physical forces with vary as $\frac{1}{r^2}$, where r is the distance between the particles.

8.3.1 Formal Mathematical Definition

The force used is the same as one of the second of the two forces available for the naïve method.

$$\vec{F_i} = \sum_{\substack{j=1 \\ j \neq i}}^{N} \alpha_{i,j}(\vec{x_j} - \vec{x_i})$$

where $\alpha_{i,j}$ is defined according to the following parameters:

$$c = 0.01, \quad min\_r = c/100, \quad r2 = max(||\vec{x_j} - \vec{x_i}||_2^2, min\_r^2), \quad r = \sqrt{r2}$$

$$\alpha_{i,j} = \begin{cases} 0 & \text{if } ||\vec{x_j} - \vec{x_i}||_2^2 > c^2 \\ \frac{1 - \frac{c}{r}}{r2} & \text{otherwise} \end{cases}$$

8.3.2 Scalable Problem Definition

See previous section.

8.3.3 Verification Scheme

We compute the root-mean-squared error between the force computed by the Barnes-Hut algorithm and direct $O(N^2)$ computation. If N is large, we can speed up the verification process by comparing only a subset of m particles.

8.3.4 Functional Reference Implementation

We direct the reader to the source code included with this report.

8.3.5 Literature Survey of Optimizations

A survey of optimizations will be included in a subsequent revision of this report.

Chapter 9

Particle-Mesh Methods

Rather than examining the individual forces between particles, we may simulate particle-particle interactions by examining the impact of the distribution of particles on the field and the impact of the field on particle velocities. In doing so, we may reduce the computational complexity of such simulations from $O(N^2)$ to $O(N)$. Although such an improvement often entails programming challenges, it allows for simulations of far larger systems over longer time scales. In this section, we introduce particle-mesh methods and enumerate the key kernels required to implement them.

The kernels for this motif will be included in a future release of this testbed.

Chapter 10

Particle-Tree Methods

An alternate approach to particle simulations is to calculate forces hierarchically. The result is a particle-tree computation whose computational performance can far exceed the direct method.

10.1 Lexicon/Terminology for Motif

Particle-tree methods use the core aspects of the direct particle-particle calculations but adds an auxiliary data structure, a tree, to accelerate force calculations.

Tree-based methods can be split into two phases: (i) a *hierarchical tree representation* for organizing particles spatially; and (ii) *fast approximate evaluation,* in which we compute the interaction between the N particles. Barnes-Hut and FMM are two popular tree-based approximation techniques. Barnes-Hut is simpler to program, while FMM guarantees an user-specified accuracy.

10.2 2D/3D Barnes Hut

Barnes-Hut is a tree-based force calculation algorithm which has an asymptotic complexity of $O(N \log N)$, in contrast to the direct method which requires $O(N^2)$ time. The improved asymptotic time complexity comes at a cost of accuracy. The original paper by Barnes and Hut [24] contains many more details on the algorithm and its complexity.

10.2.1 Formal Mathematical Definition

Each iteration of the Barnes-Hut method is composed of two key steps.

Tree construction. The particle data is compactly stored in a quad-tree (in two dimensions) or an octree (in three dimensions) representation. Initially all the particles are within a single box (or cube), and the domain is recursively partitioned into four (or eight) sub-boxes until each leaf box has no more than a single particle in it. The tree representation would have "leaf" and "non-leaf" boxes. Each non-leaf box stores an appropriate representation of the particles in its leaf boxes. The topology of the oct-tree is determined by the distribution of points.

Force evaluation. Once the tree is constructed, a "post-order" traversal of the tree is performed. A ratio $\frac{D}{r}$ is computed to determine whether to approximate the force or not. D is the dimension of the box and r is the distance of the particle from the center of mass of the box. If this ratio is less than some constant θ, then we can compute the force due to all the particles in the box by only using the center of mass of the particles in the box.

10.2.2 Scalable Problem Definition

The experimental setup is identical to the scheme employed in the direct N^2 methods.

10.2.3 Verification Scheme

We compute the root-mean-squared error between the force computed by the Barnes-Hut algorithm and direct $O(N^2)$ computation. If N is large, we can speed up the verification process by comparing only a subset of m particles.

10.2.4 Functional Reference Implementation

The current reference implementation is simple implementation according to the description in [24] with a few details from the implementation in [74]. We direct the reader to the source code for details.

10.2.5 Literature Survey of Optimizations

The original paper by Barnes and Hut [24], a parallel implementation [65], the LoneStar benchmark suite implementation [74].

10.3 2D/3D Fast Multipole Method

The kernels for this motif will be included in a future release of this testbed.

Chapter 11

Monte Carlo Methods

The term *Monte Carlo methods* encompasses a broad range of computational methods that are based on pseudorandom, statistical-based sampling. For instance, computing high-dimensional integrals using conventional numerical integration techniques (e.g., Gaussian quadrature or tanh-sinh quadrature) becomes infeasible, even on a highly parallel system, once the dimension exceeds six or so. This is because if a given integrand function requires, say, 100 function evaluations to achieve a given accuracy in one dimension, then it typically requires $100^2 = 10,000$ in two dimension, $100^3 = 1,000,000$ in three dimensions, and, in general, 100^n in n dimensions. For such integrals, Monte Carlo or Quasi-Monte Carlo methods are often the only method available. There are many applications in fields ranging from physics to economics.

11.1 Lexicon/Terminology for Motif

11.2 Quasi-Monte Carlo Integration

In this section we investigate numerical integration by Quasi-Monte Carlo (qMC) methods. This involves evaluating an integral by sampling points according to a specific random number generator averaging the values of the integrand at such points.

11.2.1 Formal Mathematical Definition

The details of qMC integration schemes are described in detail in [45]. Standard Monte Carlo integration schemes experience convergence as $O(1/\sqrt{N})$. By contrast, qMC schemes employ a specific random number generator to achieve convergence of $O(\ln^d N/\sqrt{N})$ or $O(\ln^{d-1} N/\sqrt{N})$, a substantial asymptotic improvement. This improvement lies in that "qMC sequences are *not* random in the classical sense. In fact, the points belonging to such sequences tend to avoid each other. Truly random points can be expected to exhibit gaps and clumps."[45]

The integrals selected for test problems in this section are the so-called Box-Integrals, as explored in [18]. They are defined as follows:

$$\begin{aligned} B_n(s) &:= \int_{\vec{r}\in[0,1]^n} |\vec{r}|^s \; \mathcal{D}\vec{r} \\ &= \int_0^1 \cdots \int_0^1 \left(r_1^2 + \cdots + r_n^2\right)^{s/2} \; dr_1 \cdots dr_n, \end{aligned} \tag{11.1}$$

Such integrals can be thought of as the average distance between two points in an n-dimensional cube. In this case, note that all regions of integration are the n-dimensional unit cube, which eliminates the need for the sophisticated sampling methods that are often required for regions with complicated boundaries.

11.2.2 Scalable Problem Definition

The problem may be scaled in complexity by increasing the dimension n. Selecting larger or non-integer positive s or any negative s results in more challenging integrals.

11.2.3 Verification Scheme

High-precision numerical and analytical results are known for these integrals, so meaningful verification is possible for a variety of dimensions and digits of accuracy. In particular, a formula that reduces the computation to a one dimensional integral is known. As such, standard one dimensional quadrature algorithms can be used for comparisons where direct n-dimensional schemes are prohibitively expensive. In addition, a new scheme by Crandall [43] has permitted resolution of these integrals via summing an infinite series, which expands the dimensions (up to over one million!) of the relevant integrals available for verification.

11.2.4 Functional Reference Implementation

Details of the random number generation are found in [45]. The Matlab reference implementation of the seed function and random number generator is as follows:

```
function [x, d, q, K, primes] = seed(n, Nmax, Dimension)
%
%    Calculate the seed for random number generator.
%    Generates vectors of random points.
%
%    Input:
%         int n                  Starting value of integers to generate.
%         int Nmax               Maximum number of points to generate.
%         int dimension          Dimension of random numbers to generate.
%
%    Output:
%         double vector x        Current random vector.
%         double matrix d        2d array for "odometer constants".
%         double matrix q        Address of 2d array for inverses of integer powers.
%         double k               Parameter.
%         vector primes          Array of first prime numbers of length 'dimension'

K = zeros(Dimension,1);
primes = getFirstNPrimes(Dimension);
%precompute K vector to get array dimensions
for i=1:Dimension
    K(i) = ceil( log(Nmax + 1) / log(primes(i))) ;
end

q = zeros(Dimension, max(K(:)) );
d = zeros(Dimension, max(K(:)) );
x = zeros(Dimension, 1);

for i=1:Dimension
    k = n;
    x(i) = 0;

    for j = 1:K(i)
        if j == 1
            q(i,j) = 1.0 / primes(i) ;
        else
            q(i,j) = q(i,j-1) / primes(i) ;
        end
        d(i,j) = mod(k, primes(i)) ;
```

```
        k = (k - d(i,j)) / primes(i) ;
        x(i) = x(i) + d(i,j) * q(i,j) ;
    end
end


function [x, d] = mcRandom(x, d, q, K, primes, Dimension)
%   Calculate the seed for random number generator.
%   Generates vectors of random points.
%
%   Input:
%        double vector x         Current random vector.
%        double matrix d         2d array for "odometer constants".
%        double matrix q         Address of 2d array for inverses of integer powers.
%        double k                Parameter.
%        vector primes           Array of first prime numbers of length 'dimension'
%
%   Output:
%        double vector x         Current random vector.
%        double matrix d         2d array for "odometer constants".

for i=1:Dimension
    for j = 1:K(i)
       d(i,j) = d(i,j) + 1;
       x(i) = x(i) + q(i,j);

       if d(i,j) < primes(i)
           break
       end

       d(i,j) = 0;

       if j == 1
           x(i) = x(i) - 1.0 ;
       else
           x(i) = x(i) - q(i,j-1) ;
       end
    end
end
```

The integration scheme itself if simple and easily parallelized. The reference implementation is as follows:

```
function result = qMCintegrate(integrand, seedValue, points, Dimension, s)
%   Integrates the supplied function on the unit cube of the specified dimension.
%
%   Input:
%        double      integrand(vector x, int n, double s)
%                                    Integrand function.
%                                    Evaluates at x, a 'dimension' length double precision array.
%                                    s - double precision parameter for integration function
%        int         seedValue       First index of random sequence to use.
%        int         points          Number of points for which to evaluate function.
%        int         dimension       Dimension of problem.
%        double      s               Parameter for integration function.
%
%        Output:
%             double result          Integral of integrand on 'dimension' dimensional unit cube.
```

```
[x, d, q, K, primes] = seed(seedValue, seedValue + points + 1, Dimension) ;

result = 0;
for j = 1:points
    [x, d] = mcRandom(x, d, q, K, primes, Dimension) ;
    result = result + integrand(x, Dimension, s) ;
end

result = result / points;
```

11.2.5 Literature Survey of Optimizations

Since this integration scheme is embarrassingly parallel, so optimizing this portion of the kernel is simple. Divide the number of points by the number of processors desired, then start each the sequence for each processor the corresponding number of points later. Reduce each individual sum for the final result. The authors are not aware of any further optimizations to the random number generator itself, but it is possible that some exist if performance is crucial.

Chapter 12

Graph Computations

12.1 Lexicon/Terminology for Motif

A *graph* or a *network* is an intuitive and useful abstraction for analyzing relational data. Unique entities are represented as *vertices*, and the interactions between them are depicted as *edges*. The vertices and edges can further be assigned attributes based on the information they encapsulate. Analyzing topological characteristics of a network abstraction (for instance, quantifying the structural *connectivity* pattern, or identifying entities that are *central* to the network), typically provides deeper and valuable insight into the data being studied.

Graph theory is the study of all aspects related to graphs. There are several well-known problem definitions that either operate on a graph or modify the graph in a systematic manner. We refer to these as *graph problems* or *graph-theoretic computations*. Most of these problems can be solved using multiple independent approaches, and we will refer to the solution procedures as *graph algorithms*. Keeping with the theme of this reference testbed, the kernels we discuss in this section are actually graph problems, and one is free to choose the appropriate algorithm to solve these problems.

The collective knowledge and literature on graph problems is vast and varied. We pick two illustrative examples in this version of our testbed, and both of them typically involve graph traversal in their solution strategy. The first kernel *Breadth-first search* is a fundamental and well-known graph problem, and is extensively used in several applications. The second problem *Betweenness Centrality* originates from the field of social network analysis, and is based on computation of shortest paths counts between all pairs of vertices in the graph.

12.2 Breadth-First Search (BFS)

12.2.1 Formal Mathematical Definition

Let G denote an undirected graph. We use V and E to refer to the vertex and edge sets of G. Let $s \in V$ denote the *source vertex*, a distinguished vertex in the graph. Each edge $e \in E$ is assigned a weight of unity. A *path* from vertex s to t is defined as a sequence of edges $\langle u_i, u_{i+1} \rangle$, $0 \leq i < l$, where $u_0 = s$ and $u_l = t$. The *length* of a path is the sum of the weights of edges. We use $d(s,t)$ to denote the *distance* between vertices s and t, or the length of the shortest path connecting s and t.

Given the source vertex s, Breadth-First Search (BFS) systematically explores the edges of G to discover every vertex that is reachable from s. All vertices at a distance k (or *level* k) from vertex s are first visited, before visiting any vertices at distance $k+1$. We require that the distance from s to each reachable vertex is returned as the final output. Optionally, the *breadth-first tree* rooted at s containing all the reachable vertices can be returned.

12.2.2 Scalable Problem Definition

Let n denote the number of vertices in the graph, and m denote the number of edges. We provide a collection of synthetic random graph generators where n and m can be varied to create arbitrarily-sized graphs. We

constrain the generators to generate very sparse networks, where $m = O(n)$. However, it is possible to generate Vertices are assigned integer identifiers between 0 and n. Edges, represented as pairs of vertices, are written to disk by the generator.

12.2.3 Verification Scheme

We include an option in the synthetic generator to generate fully-connected graphs (i.e., the graph has one large connected component) with vertices ordered by their distance from a specified source vertex (say, vertex with integer identifier 0). One can use this generator graph configuration to check if the vertex-distance pairs obtained as the output of the BFS algorithm are ordered in the same manner.

12.2.4 Functional Reference Implementation

We implement an $O(m + n)$-work sequential BFS algorithm that maintains the candidate set of vertices to be explored in a FIFO queue. The queue is initially set to hold the source vertex s. For each vertex v in the queue, its neighbors are inspected and added to the queue if they have not been previously visited. A boolean array of size n is maintained to indicate whether a vertex has been visited or not. The space requirements of sequential BFS are $O(n)$, and each edge in the graph is visited at most twice for an undirected graph.

12.2.5 Literature Survey of Optimizations

The queue-based approach described above is the most popular technique for a serial implementation of BFS. Since there is very little arithmetic computation being performed in this approach (i.e., we are just required to determine the distance from source vertex to all other vertices, which would at most require n additions), the performance is highly dependent on the memory access pattern of the algorithm, which in turn depends on the graph topology and the data representation chosen for the graph. Optimizations to improve serial performance typically focus on improving the cache locality of the algorithm and minimizing the number of memory references. Techniques include reordering the vertex identifiers to improve locality, compressed representations of adjacencies, and internal graph representations that exploit specific topological characteristics of the input graph.

In case of parallel algorithms, there is the additional challenge of concurrent computation. Prior work on large-scale parallel BFS implementations are either motivated by, or are extensions of, two unique parallel algorithms. In the first level-based approach that is a natural extension of the serial algorithm [89, 90], vertices are visited level by level as the search progresses, and edges are partitioned (either implicitly or explicitly) among the processors. However, a problem with this approach is that the running time scales as $O(D)$, where D is the graph diameter. If $D = O(n)$, there is very little concurrency to exploit when visiting adjacencies in a single level. Ullman and Yannakakis [98] designed an alternate algorithm based on multiple concurrent graph searches to counter this problem. Instead of a level-synchronized search, the graph is explored using multiple path-limited parallel searches, and these searches are finally stitched together to obtain a breadth-first tree from the source vertex.

Here are selected recent research papers that investigate BFS optimizations on a variety of parallel systems:

- a shared-memory implementation utilizing fine-grained synchronization techniques [12].
- a distributed-memory, message-passing implementation with a "2D-partitioned" graph [108].
- an implementation for the Cell/B.E. processor with a bulk-synchronous parallel approach [100].
- an out-of-core (external memory) BFS study [4].
- GPU optimizations for BFS [66].

12.3 Betweenness Centrality

12.3.1 Formal Mathematical Definition

Betweenness centrality is a shortest paths enumeration-based centrality metric introduced by Anthonisse [6] and Freeman [57]. Let $\delta_{st}(v)$ denote the *pairwise dependency*, or the fraction of shortest paths between s and t that pass through v:

$$\delta_{st}(v)\frac{\sigma_{st}(v)}{\sigma_{st}}$$

Betweenness centrality of a vertex v is defined as

$$BC(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$$

12.3.2 Scalable Problem Definition

We utilize the graph generators discussed for the previous BFS kernel. We restrict the problem to consider only vertex betweenness for unweighted graphs.

12.3.3 Verification Scheme

We employ the scheme outlined in the HPCS SSCA graph analysis benchmark [14, 15], of which betweenness centrality computation is a key routine. We use a integer torus, a regular network in which the centrality values of all the vertices are equal, as an input instance. Further, there is a known analytic expression for vertex betweenness for a regular torus, and we use this as a validation check for exact betweenness computation.

12.3.4 Functional Reference Implementation

We implement the sequential algorithm for this problem designed by Brandes [32]. This algorithm computes the betweenness centrality score for all vertices in the graph in $O(mn + n^2 \log n)$ time for weighted graphs, and $O(mn)$ time for unweighted graphs. The main idea is as follows. We define the *dependency* of a source vertex $s \in V$ on a vertex $v \in V$ as

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$$

The betweenness centrality of a vertex v can be then expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$. Brandes shows in his paper that the dependency $\delta_s(v)$ satisfies the following recursive relation:

$$\delta_s(v) = \sum_{w : v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$$

The algorithm exploits the recursive nature of the dependency scores. First, n shortest path computations are performed, one for each $s \in V$. The predecessor sets $pred(s, v)$ are maintained during these computations. Next, for every $s \in V$, using the information from the shortest paths tree and predecessor sets along the paths, compute the dependencies $\delta_s(v)$ for all other $v \in V$. To compute the centrality value of a vertex v, we finally compute the sum of all dependency values. The $O(n^2)$ space requirements can be reduced to $O(m + n)$ by maintaining a *running centrality score*.

12.3.5 Literature Survey of Optimizations

A straightforward way of computing betweenness centrality for each vertex would be as follows:

1. compute the length and number of shortest paths between all pairs (s, t).
2. for each vertex v, calculate every possible pair-dependency $\delta_{st}(v)$ and sum them up.

The complexity is dominated by step 2, which requires $\Theta(n^3)$ time summation and $\Theta(n^2)$ storage of pair-dependencies. Brandes' algorithm achieves a significant improvement in running time for sparse networks, and most implementations employ this algorithm.

Brandes also documents variants of betweenness centrality [33]. The fastest-known multithreaded parallel implementations for betweenness centrality computation are discussed by Madduri et al. [13, 77]. Edmonds et al. [54] recently developed a distributed-memory parallel algorithm and implementation. Approximating centrality scores is also an important problem, and there are several techniques to do so [10, 56, 62].

Chapter 13

Sorting

13.1 Lexicon/Terminology for Motif

Sorting is an ubiquitous problem that we come across in a variety of applications. Hence we reserve a separate module in our testbed for sorting and its variants. The problem can be formally defined as follows: we are given as input a sequence of n elements or values $S = \langle e_1, e_2, \ldots, e_n \rangle$. Each element may have a key, say $k_i, 1 \leq i \leq n$, associated with it. The keys come from an ordered universe, i.e., there is a linear order $\leq$ defined on the keys. We extend this order to elements so that $e_i \leq e_j$ if and only if $k_i \leq k_j$. The problem of sorting is to return as output a sequence $S' = \langle e'_1, e'_2, \ldots, e'_n \rangle$ such that S' is a permutation of S and $e'_1 \leq e'_2 \leq \ldots \leq e'_n$. Note that the ordering of equivalent elements is arbitrary. The most frequent ordering relations employed are the increasing/decreasing order of numbers, and lexicographical ordering for strings and tuples.

There are several well-known algorithms for sorting, including quicksort, merge sort, insertion sort, radix sort, etc. The complexity lower bound of $O(n \log n)$ is optimal for comparison-based sorting algorithms. We leave the choice of the algorithm open in our problem description, and present three different variants of sorting that capture various key-value configurations.

13.2 Integer Sort

13.2.1 Formal Mathematical Definition

The input for this problem is a sequence of 32-bit non-negative integers, and we require that a permutation of these integers in non-decreasing order is returned as output. In terms of the sorting definition above, this problem is analogous to the case where the key and element are identical.

13.2.2 Scalable Problem Definition

We use synthetic generators for input data generation. The number of keys n and the maximum key value C are taken as input. We generate input data representative of several different distributions, including

- A uniform random distribution of keys in $[0, 2^{31} - 1]$, using the C random number generator library.
- The sequence of keys in reverse non-increasing order.
- An "almost-sorted" distribution, where the keys are initially sorted and a small percentage are then permuted.

13.2.3 Verification Scheme

The output sequence can be easily verified in linear time. We provide a verification routine that checks if the keys in the output sequence are in the right order.

13.2.4 Functional Reference Implementation

We provide reference implementations of the quicksort, merge sort, and radix sort algorithms for this problem.

13.3 100 Byte Sort

13.3.1 Formal Mathematical Definition

This problem definition is inspired by the Sort benchmark [94], and stresses the case where the key and values can span multiple memory words. For this problem, we mandate 10-byte keys and 90-byte values (or-100 byte records), identical to the Sort benchmark definition.

13.3.2 Scalable Problem Definition

We recommend using the `gensort` synthetic generator that is used for data generation in the Sort benchmark. We provide routines to read the data from disk.

13.3.3 Verification Scheme

The verification routine is similar to the previous problem – it parses the output sequence of key-value pairs, compares the keys as strings using the C library routine *strncmp*, and checks whether they are in sorted order. Additionally, we recommend using the `valsort` program used for validation in the Sort benchmark. This program checks if records are in sorted order, counts the number of duplicate keys, and calculates the sum of the CRC32 checksums of each record.

13.3.4 Functional Reference Implementation

We provide implementations of the Quicksort and Merge Sort algorithms for this problem.

13.4 Spatial Sort

13.4.1 Formal Mathematical Definition

Consider n elements denoting point coordinates in a bounded two-dimensional space (i.e., two double-precision floating point keys per element, representing the X and Y coordinates). Assume that the initial ordering of these elements is based on a uniform random distribution. This kernel requires shuffling of the elements and assignment of a new integer key to each element, such that the elements are binned into spatial regions that follow a "Z-order" (or "Morton order" [59, 82]), and further sorted by their X and Y coordinates. The number of spatial bins is dependent on a user-specified depth parameter used in construction of the implicit Z-order curve.

13.4.2 Scalable Problem Definition

The number of elements n a sizing coefficient c are specified as input parameters. We utilize a pseudo-random number generator to generate the X and Y point coordinates (lying in $(-c, c)$) for each each particle. The Z-order depth parameter is an input parameter for the kernel. Boundaries of the region in question at each recursive step are handled automatically.

13.4.3 Verification Scheme

The verification scheme in question checks whether spatial data is sorted only relative to the underlying geometry of the region in question. The algorithm is independent of key assignment or bit interleaving. When compared to checking only that the keys are sorted properly, the algorithm provides a superior check that the sorted particles match the desired geometry.

The algorithm first checks that each the particles are correctly placed according to the boundaries of each quadrant. Next, it recursively calls the algorithm on each quadrant. The algorithm runs in $O(n)$ time, uses little additional memory and in testing is far faster than sorting the particles. Full details are provided in [71].

13.4.4 Functional Reference Implementation

We utilize a quicksort-based routine to order the points. We direct the reader to the source code for details.

13.5 Literature Survey of Optimizations

The literature on sorting algorithms and optimizations is quite extensive. We refer the reader to

- Classical papers on serial and parallel sorting studies [1–3, 7, 16, 26–28, 30, 31, 34, 47].
- Recent research on sorting algorithms optimized for multicore architectures [61, 91, 92].
- The Sort benchmark page [94], which documents results from an annual competition for out-o-core sort of large-scale data sets.

Appendix

Random Number Generators

A random number generator with a based on a provably normal class of numbers is included with the C reference implementations and used wherever needed. The scheme was developed by David Bailey and Richard Crandall to have particularly good statistical properties, in particular a very large period of $2 * 3^{32}$ or about $3.7060404 * 10^{15}$. See [22] for details of the mathematics behind this generator. The included implementation is based on Bailey's original Fortran implementation available online at [67].

Acknowledgments

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Additionally, this research was supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

Bibliography

[1] Abali, B., Özgüner, F., and Bataineh, A. Balanced parallel sort on hypercube multiprocessors. *Parallel and Distributed Systems 4*, 5 (1993), 572–581.

[2] Aggarwal, A., and Vitter, J. The input/output complexity of sorting and related problems. *Commun. ACM 31* (1988), 1116–1127.

[3] Aigner, M., and West, D. Sorting by insertion of leading element. *J. Combinatorial Theory 45* (1987), 306–309.

[4] Ajwani, D., Dementiev, R., and Meyer, U. A computational study of external-memory BFS algorithms. In *Proc. 17th annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06)* (2006), ACM, pp. 601–610.

[5] AMD Core Math Library (ACML), Advanced Micro Devices, 2007.

[6] Anthonisse, J. The rush in a directed graph. Tech. Rep. BN 9/71, Stichting Mathematisch Centrum, 1971.

[7] Arge, L., Chase, J., Vitter, J., and Wickremesinghe, R. Efficient sorting using registers and caches. In *Proc. 4th Workshop on Algorithm Engineering (WAE 2000)* (Saarbrücken, Germany, Sept. 2000).

[8] Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, University of California, Berkeley, Dec. 2006.

[9] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Lee, E., Morgan, N., Necula, G., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. The parallel computing laboratory at U.C. Berkeley: A research agenda based on the Berkeley view. Tech. Rep. UCB/EECS-2008-23, University of California, Berkeley, Mar. 2008.

[10] Bader, D., Kintali, S., Madduri, K., and Mihail, M. Approximating betweenness centrality. In *Proc. 5th Int'l. Workshop on Algorithms and Models for the Web-Graph (WAW 2007)* (Dec. 2007), A. Bonato and F. Chung, Eds., vol. 4863 of *LNCS*, Springer, pp. 124–137.

[11] Bader, D., Li, Y., Li, T., and Sachdeva, V. BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2005), pp. 163–173.

[12] Bader, D., and Madduri, K. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. 35th Int'l. Conf. on Parallel Processing (ICPP 2006)* (Aug. 2006), IEEE Computer Society, pp. 523–530.

[13] Bader, D., and Madduri, K. Parallel algorithms for evaluating centrality indices in real- world networks. In *Proc. 35th Int'l. Conf. on Parallel Processing (ICPP 2006)* (Aug. 2006), IEEE Computer Society, pp. 539–550.

[14] Bader, D., and Madduri, K. HPCS SSCA graph analysis benchmark v2.3. http://www.graphanalysis.org, 2008.

[15] Bader, D., Madduri, K., Gilbert, J., Shah, V., Kepner, J., Meuse, T., and Krishnamurthy, A. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch* (Nov. 2006).

[16] Bader, D. A., Helman, D. R., and JáJá, J. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithms 1*, 3 (1996), 1–42. www.jea.acm.org/1996/BaderPersonalized/.

[17] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS parallel benchmarks. *Int'l. Journal of High Performance Computing Applications 5*, 3 (1991), 63–73.

[18] Bailey, D., Borwein, J., and Crandall, R. Advances in the theory of box integrals. *Mathematics of Computation* (2009).

[19] Bailey, D. H. A high-performance fft algorithm for vector supercomputers. *International Journal of Supercomputer Applications 2* (1988), 82–87.

[20] Bailey, D. H. Ffts in external or hierarchical memory. *Journal of Supercomputing 4* (1990), 23–35.

[21] Bailey, D. H. Formulas for Computing FFTs on Vector Computers. 1994.

[22] Bailey, D. H., Crandall, R. E., and Example, F. Random generators and normal numbers. *Experimental Mathematics 11* (2000), 527–546.

[23] Bailey, D. H., and Frederickson, P. O. Performance results for two of the nas parallel benchmarks. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1991), ACM, pp. 166–173.

[24] Barnes, J., and Hut, P. A hierarchical $o(n \log n)$ force-calculation algorithm. *Nature 324*, 4 (1986), 446–449.

[25] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[26] Barve, R., and Vitter, J. A simple and efficient parallel disk mergesort. In *Proc. 11th Ann. Symp. Parallel Algorithms and Architectures (SPAA-99)* (Saint Malo, France, June 1999), ACM, pp. 232–241.

[27] Batcher, K. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conf. 32* (Reston, VA, 1968), pp. 307–314.

[28] Bentley, J. L., and McIlroy, M. D. Engineering a sort function. *Software - Practice and Experience 23*, 11 (1993), 1249–1265.

[29] Bienia, C., Kumar, S., Singh, J., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. Tech. Rep. TR-811-08, Princeton University, Jan. 2008.

[30] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3rd Ann. Symp. Parallel Algorithms and Architectures (SPAA-91)* (July 1991), ACM, pp. 3–16.

[31] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems 31*, 2 (1998), 135–167.

[32] Brandes, U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25*, 2 (2001), 163–177.

[33] Brandes, U. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks 30*, 2 (2008), 136–145.

[34] Brest, J., Vreže, A., and Žumer, V. A sorting algorithm on a PC cluster. In *Proc. ACM Symp. on Applied Computing (SAC 2000)* (Como, Italy, Mar. 2000), pp. 710–715.

[35] Briggs, W. L., and Henson, V. E. *The DFT: An Owners' Manual for the Discrete Fourier Transform*. SIAM, Philadelphia, 1987.

[36] Briggs, W. L., Henson, V. E., and McCormick, S. F. *A Multigrid Tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[37] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. Parallel Tiled QR Factorization for Multicore Architectures. *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE 20* (2008).

[38] Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., and Vuduc, R. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *International Conference on Parallel and Distributed Computing Systems (IPDPS)* (Atlanta, Georgia, 2010).

[39] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2009), pp. 44–54.

[40] Choi, J., Dongarra, J. J., Ostrouchov, L. S., Petitet, A. P., Walker, D. W., and Whaley, R. C. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming 5*, 3 (1996), 173–184.

[41] Chu, E., and George, A. QR Factorization of a Dense Matrix on a Hypercube Multiprocessor. *SIAM J. Sci. Stat. Comput. 11*, 5 (1990), 990–1028.

[42] Colella, P. Defining software requirements for scientific computing, 2004. DARPA HPCS presentation.

[43] Crandall, R. Theory of box series, 2010.

[44] Crandall, R., Klivington, J., and Mitchell, D. Large-Scale FFTs and Convolutions on Apple Hardware, 2008. ttp://images.apple.com/acg/pdf/FFTapps.pdf.

[45] Crandall, R., and Pomerance, C. Prime numbers: a computational perspective. second edition, 2005.

[46] CS 267 assignment page, U.C. Berkeley. http://www.cs.berkeley.edu/~agearh/cs267.sp10/hw2/.

[47] Culler, D., Dusseau, A., Martin, R., and Schauser, K. Fast parallel sorting under LogP: From theory to practice. In *Portability and Performance for Parallel Processing*. John Wiley & Sons, 1993, ch. 4, pp. 71–98.

[48] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proc. SC2008: High performance computing, networking, and storage conference* (nov 2008). https://hpcrd.lbl.gov/~swwilliams/research/papers/sc08_stencil.pdf.

[49] Demmel, J. *Applied Numerical Linear Algebra*. SIAM, 1997.

[50] Demmel, J., Marques, O., Parlett, B., and Vömel, C. A testing infrastructure for LAPACK's symmetric eigensolvers. Tech. Rep. 182, LAPACK Working Note, Apr. 2007.

[51] Demmel, J. W. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[52] Dongarra, J., Demmel, J., Husbands, P., and Luszczek, P. HPCS Library Study Effort . *Technical Report UT-CS-08-617, University of Tennessee* (2008).

[53] Dubey, P. A platform 2015 workload model: Recognition, Mining and Synthesis moves computers to the era of Tera. Tech. rep., Intel Corporation, 2005.

[54] Edmonds, N., Hoefler, T., and Lumsdaine, A. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *Proc. Int'l. Conf. on High Performance Computing (HiPC 2010)* (2010). to appear.

[55] The embedded microprocessor benchmark consortium. http://www.eembc.org.

[56] Eppstein, D., and Wang, J. Fast approximation of centrality. *Journal of Graph Algorithms and Applications 8*, 1 (2004), 39–45.

[57] Freeman, L. A set of measures of centrality based on betweenness. *Sociometry 40*, 1 (1977), 35–41.

[58] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[59] Gaede, V., and Günther, O. Multidimensional access methods. *ACM Computing Surveys (CSUR) 30*, 2 (1998), 170–231.

[60] Gahvari, H., Hoemmen, M., Demmel, J., and Yelick, K. Benchmarking Sparse Matrix-Vector Multiply in Five Minutes. In *SPEC Benchmark Workshop* (January 2007).

[61] Gedik, B., Bordawekar, R., and Yu, P. CellSort: High performance sorting on the Cell processor. In *Proc. VLDB* (2007), pp. 1286–1207.

[62] Geisberger, R., Sanders, P., and Schultes, D. Better approximation of betweenness centrality. In *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX 2008)* (Jan. 2008), SIAM, pp. 90–100.

[63] Gentlemen, W. M., and Sande, G. Fast fourier transforms-for fun and profit. *Managing Requirements Knowledge, International Workshop on 0* (1966), 563.

[64] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 3rd ed. The John Hopkins University Press, 1996.

[65] Grama, A., Kumar, V., and Sameh, A. Scalable parallel formulations of the Barnes-Hut method for n-body simulations. *Parallel Computing 24*, 5–6 (1998), 797–822.

[66] Harish, P., and Narayanan, P. Accelerating large graph algorithms on the GPU using CUDA. In *Proc. 14th Int'l. Conf. on High-Performance Computing (HiPC)* (2007), pp. 197–208.

[67] High-precision software directory. http://crd.lbl.gov/~dhbailey/mpdist/.

[68] HPC Challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[69] Husbands, P., and Yelick, K. Multi-threading and One-sided Communication in parallel LU Factorization. *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (2007), 1–10.

[70] Engineering and Scientific Subroutine Library, IBM, 2003.

[71] Kaiser, A. An $O(n)$ Algorithm for Verification of Data Sorted According to Space-Filling Curves, 2010. *Manuscript*.

[72] Kaiser, A., Williams, S., Madduri, K., Ibrahim, K., Bailey, D., Demmel, J., and Strohmaier, E. A Case for a Testbed of Kernels for Software/Hardware Co-Design Research. In *HotPar '10: Second USENIX Workshop on Hot Topics in Parallelism* (June 2010).

[73] Kamil, S., Chan, C., Oliker, L., Shalf, J., and Williams, S. An auto-tuning framework for parallel multicore stencil computations. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)* (Atlanta, Georgia, 2010).

[74] Kulkarni, M., Burtscher, M., Cascaval, C., and Pingali, K. Lonestar: a suite of parallel irregular programs. In *Proc. IEEE Int'l. Symp. on Performance Analysis of Systems and Software (ISPASS)* (Apr. 2009), pp. 65–76.

[75] Lam, M. D., Rothberg, E. E., and Wolf, M. E. The cache performance and optimizations of blocked algorithms. *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (1991), 63–74.

[76] Li, M.-L., Sasanka, R., Adve, S., Chen, Y.-K., and Debes, E. The ALPBench benchmark suite for complex multimedia applications. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2005), pp. 34–45.

[77] Madduri, K., Ediger, D., Jiang, K., Bader, D., and Chavarria-Miranda, D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP 2009)* (May 2009), IEEE Computer Society.

[78] Madduri, K., Williams, S., Ethier, S., Oliker, L., Shalf, J., Strohmaier, E., and Yelick, K. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *Proc. SC2009: High performance computing, networking, and storage conference* (2009).

[79] McCalpin, J. Memory bandwidth and machine balance in current high performance computers. *IEEE CS TCCA Newsletter* (Dec. 1995), 19–25.

[80] Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. STAMP: Stanford transactional applications for multi-processing. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Sept. 2008), pp. 35–46.

[81] Mohiyuddin, M., Hoemmen, M., Demmel, J., and Yelick, K. Minimizing communication in sparse matrix solvers. In *Proc. SC2009: High performance computing, networking, and storage conference* (2009). http://dx.doi.org/10.1145/1654059.1654096.

[82] Morton, G. A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ltd., 1966.

[83] Narayanan, R., Ozisikyilmaz, B., Zambreno, J., Memik, G., and Choudhary, A. MineBench: A benchmark suite for data mining workloads. In *Proc. IEEE Int'l. Symp. on Workload Characterization (IISWC)* (Oct. 2006), pp. 182–188.

[84] Parallel dwarfs. http://paralleldwarfs.codeplex.com/.

[85] The Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[86] Pase, D. The pChase memory benchmark page. http://pchase.org/.

[87] Petitet, A., Whaley, R., Dongarra, J., and Cleary, A. HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/.

[88] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2007.

[89] Quinn, M., and Deo, N. Parallel graph algorithms. *ACM Comput. Surv. 16*, 3 (1984), 319–348.

[90] Reghbati, A., and Corneil, D. Parallel computations in graph theory. *SIAM Journal of Computing 2*, 2 (1978), 230–237.

[91] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore GPUs. In *Proc. IEEE Int'l. Symp. on Parallel & Distributed Processing (IPDPS '09)* (2009), IEEE Computer Society, pp. 1–10.

[92] Satish, N., Kim, C., Chhugani, J., Lee, A. N. V., Kim, D., and Dubey, P. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proc. Int'l. Conf. on Management of Data (SIGMOD '10)* (2010), ACM, pp. 351–362.

[93] Singh, J., Weber, W.-D., and Gupta, A. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News 20*, 1 (1992), 5–44.

[94] Sort benchmark home page. http://sortbenchmark.org/.

[95] SPEC benchmarks. http://www.spec.org/benchmarks.html.

[96] SPIRAL Project. http://www.spiral.net.

[97] Strohmaier, E., Williams, S., Kaiser, A., Madduri, K., Ibrahim, K., Bailey, D., and Demmel, J. W. A kernel testbed for parallel architecture, language, and performance research. *AIP Conference Proceedings 1281*, 1 (2010), 1297–1300.

[98] Ullman, J., and Yannakakis, M. High-probability parallel transitive closure algorithms. In *Proc. 2nd Annual Symp. on Parallel Algorithms and Architectures (SPAA 1990)* (Crete, Greece, July 1990), ACM, pp. 200–209.

[99] Van Loan, C. *Computational frameworks for the fast Fourier transform.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[100] Villa, O., Scarpazza, D., Petrini, F., and Peinador, J. Challenges in mapping graph algorithms on advanced multi-core processors. In *Proc. 21st Intl. Parallel and Distributed Processsing Symp. (IPDPS 2007)* (Long Beach, CA, March 2007).

[101] Vuduc, R., Demmel, J., and Yelick, K. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series* (June 2005), Institute of Physics Publishing.

[102] Wellein, G., Hager, G., Zeiser, T., Wittmann, M., and Fehske, H. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. *Computer Software and Applications Conference, Annual International 1* (2009), 579–586.

[103] Whaley, R. C., Petitet, A., and Dongarra, J. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing 27(1-2)* (2001), 3–35.

[104] Williams, S., Carter, J., Oliker, L., Shalf, J., and Yelick, K. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)* (Miami, Florida, 2008).

[105] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Conference on Supercomputing (SC07)* (2007).

[106] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing - Special Issue on Revolutionary Technologies for Acceleration of Emerging Petascale Applications 35*, 3 (2008), 178–194.

[107] Yeh, T., Faloutsos, P., Patel, S., and Reinman, G. Parallax: an architecture for real-time physics. *SIGARCH Comput. Archit. News 35*, 2 (2007).

[108] Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., and Çatalyürek, Ü. V. A scalable distributed parallel breadth-first search algorithm on Bluegene/L. In *Proc. Supercomputing (SC 2005)* (Seattle, WA, Nov. 2005).