Iowa State University
Digital Repository

Graduate Theses and Dissertations | Iowa State University Capstones, Theses and Dissertations

2019

# Using machine learning to improve dense and sparse matrix multiplication kernels

Brandon Groth
*Iowa State University*

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Applied Mathematics Commons, and the Computer Sciences Commons

## Recommended Citation

Groth, Brandon, "Using machine learning to improve dense and sparse matrix multiplication kernels" (2019). *Graduate Theses and Dissertations.* 17688.
https://lib.dr.iastate.edu/etd/17688

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

**Using machine learning to improve dense and sparse matrix multiplication kernels**

by

**Brandon Micheal Groth**

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Applied Mathematics

Program of Study Committee:
Glenn R. Luecke, Major Professor
James Rossmanith
Zhijun Wu
Jin Tian
Kris De Brabanter

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

Copyright © Brandon Micheal Groth, 2019. All rights reserved.

## DEDICATION

I would like to dedicate this thesis to my wife Maria and our puppy Tiger. Without my girls support I would not have been able to complete this work. I would also like to thank my parents for supporting me throughout my undergraduate and graduate years. It has been a long

# TABLE OF CONTENTS

Page


LIST OF TABLES . . . v

LIST OF FIGURES . . . vi

ACKNOWLEDGMENTS . . . vii

ABSTRACT . . . viii

CHAPTER 1. INTRODUCTION . . . 1
- 1.1 Thesis Overview . . . 1
- 1.2 Terms and Definitions . . . 1
- 1.3 High Performance Computing . . . 3
  - 1.3.1 Early Supercomputers . . . 3
  - 1.3.2 Modern Supercomputers . . . 5
  - 1.3.3 Computer Architecture Classification . . . 7
  - 1.3.4 HPC Programming Languages . . . 8
  - 1.3.5 HPC Parallel APIs . . . 11
- 1.4 References . . . 14

CHAPTER 2. USING MACHINE LEARNING TO IMPROVE SHARED-MEMORY GENERAL MATRIX-MATRIX MULTIPLICATION . . . 17
- 2.1 Abstract . . . 17
- 2.2 Introduction . . . 17
- 2.3 Background . . . 18
  - 2.3.1 Code Optimizations for Matrix Multiplication . . . 18
  - 2.3.2 Matrix Multiplication - Recursion . . . 21
  - 2.3.3 Basic Linear Algebra Subprograms . . . 23
  - 2.3.4 Supervised Learning . . . 25
  - 2.3.5 Model Training and Evaluation . . . 27
  - 2.3.6 Support Vector Machines . . . 30
  - 2.3.7 Decision Trees . . . 32
- 2.4 Methodology . . . 36
  - 2.4.1 NUMA Considerations . . . 39
- 2.5 Algorithm Selection . . . 40
  - 2.5.1 Data Set Analysis . . . 41
  - 2.5.2 The C5.0 Classifier . . . 43
  - 2.5.3 ML_GEMM . . . 45
- 2.6 Related Work . . . 46

2.7 Conclusion . . . . . 47
2.8 Appendix . . . . . 48
2.9 References . . . . . 50

CHAPTER 3. CSR-DU ML: A MINIMAL DELTA UNITS LIBRARY USING MACHINE LEARNING . . . . . 53
3.1 Sparse Matrix Formats . . . . . 53
3.1.1 Compressed Sparse Row . . . . . 54
3.1.2 Blocked Compressed Sparse Row . . . . . 55
3.1.3 Sparse Diagonal Format . . . . . 57
3.1.4 ELL Format . . . . . 58
3.1.5 Sparse Matrix Libraries . . . . . 59
3.2 CSR Delta Units . . . . . 59
3.2.1 Compressed Sparse eXtended . . . . . 61
3.3 CSR-DU and CSX Implementation . . . . . 65
3.3.1 Implementation . . . . . 65
3.3.2 CSR-DU Parallel Implementation . . . . . 71
3.4 References . . . . . 78

CHAPTER 4. CONCLUSION . . . . . 79
4.1 Project Summaries . . . . . 79
4.2 Future Work . . . . . 80

# LIST OF TABLES

Page

Table 2.1 Rectangular Matrix Distribution . . . 42

Table 2.2 Speedup Results for GEMM Algorithms. Each row represents when the denominator GEMM algorithm was fastest. . . . 42

Table 2.3 C5.0 Decision Tree Properties . . . 45

Table 2.4 C5.0 Decision Tree Class Accuracy . . . 45

Table 2.5 C5.0 Confusion Matrix. Rows indicate class predictions of the classifier $f(\vec{x}_i)$ and columns represent the true class values $y_i$. . . . 45

Table 2.6 ML_GEMM vs MKL Speedup Analysis. Each row represents when the denominator algorithm was selected as fastest. . . . 46

Table 3.1 CSX Horizontal Structure Transformations . . . 62

Table 3.2 Bitarray Read and Write Implementation with Byte-aligned Offsets . . . 65

Table 3.3 CSR-DU Header Byte Format. . . . 66

# LIST OF FIGURES

**Page**

Figure 2.1 Algorithm Scalability with for Square GEMM . . . . . . . . . . . . . . . . . . 38
Figure 2.2 Algorithm Scalability with for Non-square GEMM . . . . . . . . . . . . . . . 38
Figure 2.3 Non-MKL Data Locations 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Figure 2.4 Non-MKL Data Locations 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Figure 3.1 CSX SpMV Delta Unit Flow Chart . . . . . . . . . . . . . . . . . . . . . . . . . 64
Figure 3.2 CSR-DU U64 Header Bitarray . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Figure 3.3 CSR-DU OpenMP Delta Unit SpMV Flow Chart . . . . . . . . . . . . . . . . . 77

# ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Glenn Luecke for his support through this research and the writing of this thesis. Furthermore, I would like to thank Dr. Janis Keuper from Fraunhofer Institute for Industrial Mathematics for inspiring our work in dense matrix-matrix multiplication. I would also like to thank my committee members for their efforts and contributions to this work, included Dr. Zhijun Wu for suggesting to explore sparse matrix multiplication algorithms as our next project.

## ABSTRACT

This work is comprised of two different projects in numerical linear algebra. The first project is about using machine learning to speed up dense matrix-matrix multiplication computations on a shared-memory computer architecture. We found that found basic loop-based matrix-matrix multiplication algorithms tied to a decision tree algorithm selector were competitive to using Intel's Math Kernel Library for the same computation. The second project is a preliminary report about re-implementing an encoding format for spare matrix-vector multiplication called Compressed Spare eXtended (CSX). The goal for the second project is to use machine learning to aid in encoding matrix substructures in the CSX format without using exhaustive search and a Just-In-Time compiler.

# CHAPTER 1. INTRODUCTION

## 1.1 Thesis Overview

This dissertation is organized as follows. Chapter 1 contains background material for high performance computing, including hardware history, high performance computing languages, and high performance computing libraries. Chapter 2 contains is an extended paper on using machine learning to speed up a matrix-matrix multiplication kernel using machine learning. Additionally, Chapter 2 contains background information for code optimizations in matrix multiplication, various matrix multiplication algorithms, as well as a brief overview of topics in supervised machine learning related to the paper. Chapter 3 pertains to an ongoing project in sparse matrix-vector multiplication format implementations. We present preliminary results for the CSR-DU library redesigned with OpenMP. Lastly in Chapter 4, final conclusions are given.

## 1.2 Terms and Definitions

**High Performance Computing** (HPC): Aggregating compute power in hardware and software in a way that executes tasks significantly faster compared to single commodity computers.

**Application Program Interface** (API): A list of procedures and functions offered by an application.

**Input/output** (I/O): is a communication process to send information between a computer system and the outside world, such as with a human via a computer display device or with another computer system.

**Central Processing Unit** (CPU): An electronic circuit that performs computational instructions in a computer including arithmetic, logical, and I/O operations.

**CPU Cache Memory**: A small, but very fast memory bank located on the CPU to quickly stream data back and forth from the processor.

**CPU Main Memory**: A memory bank connected to the CPU via a memory bus, often with a much larger and slower storage capacity compared to cache memory.

**Hardware Accelerator**: A hardware accelerator is computer hardware that is specially designed to perform a computational task more quickly and/or more efficiently than a CPU.

**Graphics Processing Unit** (GPU): A hardware accelerator designed to quickly and efficiently create and alter images for a computer display device.

**General-purpose Graphics Processing Unit** (GPGPU): A GPU hardware accelerator that is used for computational tasks that process non-graphical data as if it were an image.

**Thread**: The smallest sequence of instructions that can be managed independently by a computer or operating system scheduler.

**Process**: An instance of a computer program that is being executed by at least one thread in a designated memory space.

**Multithreading**: The ability of a CPU to provide multiple threads of execution concurrently.

**Multiprocessing**: The use of multiple CPUs in a computer system, often where the CPUs share a common memory bank.

**Shared-memory Architecture**: A computer architecture that connects at least two CPUs to a shared memory bank, allowing the CPUs to access the shared memory simultaneously.

**Shared-memory Computing**: A programming model that leverages a shared-memory architecture, usually through threading, to allow communication between connected CPUs.

**Uniform Memory Architecture** (UMA): A shared-memory architecture where all CPUs share a single block of memory.

**Non-Uniform Memory Architecture** (NUMA): A shared-memory architecture where subgroups of CPUs share a local memory bank and each subgroup is connected to other subgroups via an interconnect to form a global shared memory bank.

**Message Passing**: A communication model where invoking processes sends a signal to receiver processes to execute an operation.

**Distributed Computing**: A programming model that uses computers connected to network to perform computations via message passing.

## 1.3 High Performance Computing

### 1.3.1 Early Supercomputers

The beginnings of so called "supercomputers" starts in 1960 with the completion of the Lawrence Atomic Research Computer (LARC) built by Sperry Rand (also known as Sperry Corporation) for the United States Navy Research and Development Center (Cole, nd). Soon afterwards in 1961, the IBM 7030 (also known as IBM Stretch) was created as the first transistor-based supercomputer for Los Alamos National Laboratory (IBM Archives, nd). The aim of IBM Stretch was to be 100 times faster than the IBM 704 (the fastest computer of the time), but fell short of the mark at approximately a 30x speedup. The lack of results proved to be an embarrassment for IBM and the purchase price of the Stretch was slashed in half in order to appease those who already ordered the system. Further sales were halted, but the Stretch remained a basis for future IBM designs.

The first commercial success story that defined the supercomputing market was the CDC 6600 made by Control Data Corporation in 1964 (Thornton, 1980). The computer outperformed the IBM Stretch by a factor of three and was the fastest computer for five years. The CDC 6600 was the first supercomputer to use Silicon transistors and a refrigeration unit to prevent overheating.

Over one hundred CDC 6600's were sold to customers all over the world until the crown to its successor, the CDC 7600. The new computer was machine-code backwards compatible with the 6600, and boasted many other design improvements over its predecessor. Control Data Corporation released several other computers throughout the 1970's and 1980's under the CDC Cyber brand consisting of supercomputers, commercial mainframe computers, and minicomputers.

After the CDC 6600, Control Data Corporation and IBM were in a computing race for the next 15 years. However, the lead designed of the CDC 6600, Seymour Cray, left Control Data Corporation in 1972 to start his own company called Cray Research. By 1976, Cray Research released the Cray-1 system which went on to be the most successful computers in history. Cray-1 was the first computer to implement a fast vector processing unit, capable of performing a single operation on a chunk of data quickly. There were previous attempts at vector processing, such as the CDC STAR-100, but had to be implemented in a way that was severely detrimental to performance. Cray Research would go on to build the Cray X-MP (1982) and the Cray-2 (1985). The Cray-2's unique waterfall cooling system became part of popular culture as it was featured in movies. Further vector systems from Cray were the Cray-C90 (1991) and the Cray-T90 series (1995), with the later marking the end of the Cray Research vector processing computers.

Another influential computer that beat the Cray-1 to launch by twelve months was the ILLIAC IV designed by the Illinois Automatic Computer team at the University of Illinois at Urbana-Champaign (Barnes et al., 1968). The IllIAC IV was the first massively parallel computer housing housing four quadrants of 64 floating point units (FPUs), where each quadrant could run separate programs or be grouped into a single job. It was also the first computer to be network-available through the Advanced Research Projects Agency Network (ARPANET). Although the ILLIAC IV was considered a failure due to its massive budget and long project length, the design influenced future machines for the next decade. Most notably, the Connection Machine (CM), built in 1980 by MIT, used upwards of 65,000 microprocessors connected by a network to allow data sharing (Hillis, 1986). Furthermore, the LINKS-1 prototype graphics system, made by a team of professors and students at Osaka University in 1982, was specially designed to render 3-D graphics via ray

tracing (IPSJ Computer Museum, nd). The design used 514 processors to compute the illumination of pixels for an image in parallel. The LINKS-1 was the first supercomputer in Japan, and was considered the worlds fastest computer until 1984.

### 1.3.2 Modern Supercomputers

By the late 1990's, general-purpose commercial CPUs had improved to the point that they could be used as off-the-shelf components in supercomputers. The first supercomputer to take advantage of this was the Intel ASCI Red supercomputer built at Sandia National Laboratories in 1997. The ASCI Red used more than 9000 Intel Pentium commercial processors connected via message passing to gain the title as the worlds fastest computer from 1996 to 1999 (TOP500 Project, 1997). The supercomputer was designed with a cabinet architecture, where each cabinet contains nodes connected in a local-area-network, and each node has several CPUs with local memory that can do computations, I/O, and send messages. While ASCI Red was not the first cabinet cluster supercomputer, its reliability and dominance in the TOP500 for three years moved the industry towards the architecture.

Clusters continued to become faster over time using the newest CPUs and memory packages. However, processor development hit the so called "Power Wall" by the mid 2000's, where increasing the clock speed of the CPU meant dramatically increasing the amount of heat the processor would produce (Patterson and Hennessy, 2013). Heat density has always been a major concern for clusters, as higher temperatures reduced system lifetimes and can cause daily failures in the system. Since processors couldn't get much faster without expensive cooling systems, the only avenues were to move to using multiple cores and perform the same computations using less power.

The IBM Blue Gene systems were the first supercomputer to explicitly use low power processors in order to control heat density via air conditioning at room temperature (journal of Research and staff, 2008). Performance per watt was becoming a major concern for customers as supercomputers starting ballooning in size, with heat dissipation being one of the largest maintenance costs. Due

to these concerns, the Green500 list was made by the Top500 organization in 2007 to reorder the fastest 500 computers in terms of performance per watt (Green500, 2007).

The last major industrial shift in modern supercomputing has been the use of hardware accelerators for computation. Accelerators are specialized compute units that have better performance and/or power efficiency for specific tasks compared to CPUs. Accelerators come in many forms, but the most popular general-purpose accelerators have been the NVIDIA Tesla GPUs and the Intel Xeon Phi coprocessors (Lindholm et al., 2008) (Rahman, 2013). Both of these accelerators are classified as many-core architectures, where many smaller cores are used in parallel to perform a task. The TESLA's have hundreds to thousands of CUDA cores which are specifically designed for graphic processing tasks, while the Xeon Phi's uses 50-70 low-power, wide vector processors. Both accelerators have been featured in many Top500 supercomputers, such as Summit computer at Oak Ridge National Laboratory using 9216 IBM Power9 processors with 27,646 NVIDIA Tesla V100 GPUs (McCorkle, 2018). Summit is presently the world's fastest computer clocking in at 148,600.0 TFLOPS since 2018 and is the third most energy efficient computer (TOP500 Project, 2019) (Green500, 2019). The largest cluster using the Xeon Phi architecture currently is Trinity at Los Alamos National Laboratory, using 301,952 Haswell Intel Xeon processors and 678,912 Knight's Landing Xeon Phi processors to have a performance of 20,158.7 TFLOPS (TOP500 Project, 2019). While the Knight's Corner Xeon Phi was a traditional accelerator device used in-conjunction with a CPU, the Knight's Landing Xeon Phi was designed as an alternative CPU for highly parallel tasks. Unfortunately, the Intel Xeon Phi product line was cancelled in 2018 due to poor adoption. Beyond Summit, it is believed that hundreds of thousands of next-generation GPUs will be required to break the so called exascale barrier, where a single computer will be able to perform at least 1 exaFLOP (1 million TFLOPS) on the Linpack Benchmark for linear algebra problems (Dongarra, 1988).

### 1.3.3 Computer Architecture Classification

Flynn's taxonomy is a classification of serial and parallel computer architectures (Flynn, 1972). The classification is based on how computer instructions interact with program data. The original four classifications are single instruction, single data (SISD), single instruction, multiple data (SIMD), multiple instruction, single data (MISD), and multiple instruction, multiple data (MIMD). Example architectures for each type are:

- SISD: uniprocessor computers,
- SIMD: vector processors and GPUs,
- MISD: fault-tolerant distributed systems that need to agree on a single result,
- MIMD: multi-core processors and distributed systems.

Parallel programming takes advantage of either SIMD and/or MIMD architectures. GPU architectures are classified as being a subset of SIMD called single instruction, multiple threads (SIMT). While Flynn's taxonomy definition of SIMD makes no distinction in how a single instruction is used on multiple pieces of data, NVIDIA uses SIMT to explain their GPU architecture. Also, the MIMD classification leaves open the possibility that each CPU in a distributed system is a SISD or SIMD architecture.

The MIMD classification can be split into two additional classifications called same program, multiple data (SPMD) and multiple program, multiple data (MPMD). SPMD is used to describe many independent processors executing the same program, where each processor can have different entry and exit points within the program (Darema et al., 1988). MPMD is used to describe manager/worker architectures, where the manager runs one program to distribute work, while the workers run another program with the data received to return some result. SPMD is the most common parallel programming architecture used in high performance computing.

### 1.3.4 HPC Programming Languages

#### 1.3.4.1 Fortran

The first high-level programming languages to exist was FORTRAN, short for Formula Translation, created by John Backus and his team at IBM in 1954 (Backus, 1981). The idea of FORTRAN was to create a non-assembly language that was similar to mathematical equations, as writing assembly code was very cumbersome. By 1960, the first FORTRAN compiler was released to much skepticism. All code prior was handmade in assembly language, so no one knew if a compiler could compete. But after showing the amount of code needed to write a program in FORTRAN could be significantly reduced with use of a compiler, FORTRAN quickly because the most supported language with over 40 versions of the FORTRAN compiler by 1963. In an interview in 1979, Backus said, "Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs" (Bergstein, 2007). Backus was awarded the Turing Award from the Association for Computing Machinery in 1977 for the creation of FORTRAN and for his publications on computer language specifications (Booch, nd).

The original version of FORTRAN used many constructs that modern programming languages include today, such as if statements, assignment statements, do loops, I/O, and exception handling. As FORTRAN's popularity grew, new versions of FORTRAN were released to meet customer requests. FORTRAN II (1958) added function and subroutine constructs and additional data types, FORTRAN III, while never released, added the ability to include inline assembly into the code, and FORTRAN IV (1962) added Boolean logic. In 1966, the American Standards Institute (now called American National Standards Institute or ANSI) published two standards called FORTRAN (based on FORTRAN IV) and Basic FORTRAN (a stripped-down version of FORTRAN II). The former version became known as FORTRAN 66, which included several additional improvements over the original FORTRAN IV such as Hollerith constants (an early character data type), code comments, and variable identifiers up to six letters in length. Customers continued to expand the language, forcing the ANSI committee to publish a new version of FORTRAN in 1978 known as

FORTRAN 77 Brainerd (1978). Additional improvements included replacing Hollerith constants with the character data type, adding block if statements, improved I/O, and intrinsic functions.

The next the next version of FORTRAN was slow to market, as shifts in industry were happening too rapidly to keep up. The first "modern" version of FORTRAN was released by ANSI in 1991 called Fortran 90 (Adams et al., 1992). The largest changes included modules, dynamic memory functionality, array programming operations, pointers, a switch construct called select case, and much more. Fortran 95 was only a incremental improvement to Fortran 90 that released in 1997. Changes made were largely to clean up some Fortran 90 specifications, but it did add functionality from a extension of Fortran 90 called High Performance Fortran (Kennedy et al., 2011). High Performance Fortran was used to aid in adding vectorization and parallelism to Fortran 90 programs, but was never widely adopted. Fortran 2003, released in 2004, was a major revision that added object-oriented programming support, as well as Interoperability with C (i.e. the ability to use C programs with Fortran) (Reid, 2007). Again, Fortran 2003 received a minor update in 2010 called Fortran 2008 (Reid, 2014). The largest change was the inclusion of another parallel extension called Coarray Fortran (Numrich and Reid, 1998). Coarrays are data sharing construct used to distribute data to images (i.e. processors) in a shared-memory or distributed system. The latest version of Fortran is Fortran 2018, which further extended the functionality of coarrays and improved interoperability with C (Reid, 2018).

Fortran has been used in high performance computing since its conception to great effect. The marketing company enlyft shows that there are over 8000 companies using Fortran in some capacity, with 51% of all companies residing in the United States mostly comprised of higher education, software development, and research (Companies using Fortran, nd). However, according to the TIOBE index for programming language popularity as of October 2019, Fortran has a 0.44% market share (TIOBE Index, 2019). Unfortunately, the number of Fortran compilers available for free or commercial use has been dwindling as Fortran has lost popularity to other languages. For high performance computing, a few of the compilers currently available are made by ARM, Cray, GNU, HPE, IBM, Intel, and The Portland Group. CPU vendors such as ARM, IBM, and Intel

use their compilers for optimizations on their hardware, often neglecting other architectures. HPC vendors such as Cray and HPE have compilers that support many CPU architectures as they sell clusters containing CPUs from various companies.

#### 1.3.4.2 C

The C programming language was developed by Denis Richie at Bell Telephone Laboratories as a means to develop the Unix operating system (Ritchie, 1996). C was born from its "parent" language B, released in 1969, and its "grandparent," Basic Combined Programming Language (BCPL), released in 1967. The C compiler was completed in 1972 and was included with Version 2 Unix. By 1973, C and the C compiler were was mature enough to rewrite the entire Unix operating system in C to make Version 4 Unix. Prior to this, all operating systems were written in assembly language. The first standard for C came in 1978 with the release of The C Programming Language book, which became known as K&R C (Kernighan and Ritchie, 1978). The standard introduced standard I/O, additional data types, and several arithmetic operators. Due to the lack of an official standard, both Institute of Electrical and Electronics Engineers (IEEE) and ANSI published standards known as POSIX C, released in 1988, and ANSI C, released in 1989. In 1990, the International Organization for Standardization (ISO) adopted ANSI C with minor changes as its standard, which has lead to the versions C89 and C90 being mostly equivalent. As popular as C is, only two major standards have been released since ANSI C being C99, released in 1999, and C11, released in 2011. C99 additions included inline functions, additional data types, and the one-line comment, while C11 added parallel multi-threading, additional macros, and structure support. C18, the current standard of C released in 2018, only clarified or fixed definitions from C11.

The C language, unlike Fortran, was never designed to be a mathematical or high performance computing language. However, due to C's popularity, additional libraries were added to make it a competitive language to Fortran. Other than multi-threading, the C standard leaves parallel constructs to be implemented by outside libraries or the compiler. One such extension to C that

has been used in high performance computing is called Uniform Parallel C (UPC) (Bonachea, 2013). UPC is a C99 extension using the PGAS model to perform parallel operations on local and distributed systems. The current version of UPC is version 1.3 released in 2013. Further work on UPC has been ported to C++ under the name UPC++ (Bachan et al., 2017).

### 1.3.5 HPC Parallel APIs

The first threading library that has been used extensively for parallelism is POSIX threads (pthreads) (IEEE, 1996). Pthreads is a parallel execution model for multithreading in C, offering thread management and synchronization, as well as other parallel constructs. Pthreads is considered a low-level threading library where the user must usher each thread through the execution. The pthreads API uses a fork-join execution model where threads are "forked" via a function call to do work, then joined with an explicit barrier. An examle of a Hello World program written in C with pthreads is given (Barney, 2011).

Listing 1.1 Hello World with Pthreads

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #define NUM_THREADS     5
4  void *PrintHello(void *threadid)
5  {
6     long tid; tid = (long)threadid;
7     printf("Hello World! It's me, thread #%ld!\n", tid);
8     pthread_exit(NULL);
9  }
10 int main (int argc, char *argv[])
11 {
12    pthread_t threads[NUM_THREADS]; int rc; long t;
13    for(t=0; t<NUM_THREADS; t++){
14       rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
15       if(rc){ printf("code from pthread_create() is %d\n", rc); exit(-1); }
16    }
```

```
17      pthread_exit(NULL);
18  }
```

Open Multi-Processing (OpenMP) is a multithreading API that supports Fortran, C, and C++ released in 1997 (Dagum and Menon, 1998). Versions 1.0 and 2.0 of OpenMP allowed for the parallelization of loops common in vector and matrix computations. Loop-based parallelism in OpenMP allows threads to be responsible for a number of iterations of a loop construct. A typical loop can be parallelized simply by adding the following command prior to a loop

```
1 <language_directive> omp parallel <language_loop> private(<loop_index>)
```

where `<language_directive>` is the language-specific directive specifier (`!$omp` in Fortran and `#pragma` in C/C++), `<language_loop>` is the language-specific iterative loop construct (`do` in Fortran and `for` in C/C++), and `<loop_index>` is a loop index variable. Version 3.0 of OpenMP introduced task-based parallelism, which extended the use of OpenMP to other kernels (OpenMP Architecture Review Board, 2008). Updates 4.0 and 5.0 have added additional functionality to tasks, SIMD directives, and other language-specific options. Tasks in OpenMP are sequences of instructions that can computed independently, but don not have to be contained within a loop construct. Two examples of operations well-suited for tasks are loops of unknown length and recursion. Another example of Hello World is given using OpenMP in C.

Listing 1.2 Hello World with OpenMP

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  int main(int argc, char* argv[])
5  {
6    #pragma omp parallel
7    {
8      printf("Hello World from thread = %d\n", omp_get_thread_num());
9    }
10 }
```

Message Passing Interface (MPI) is a distributed message passing standard released in 1994 (Message Passing Interface Forum, 1994). MPI is designed to use the following communication models: point-to-point (one sender to one receiver), collective (messages involving a communication group), and one-sided (accessing other memory banks). MPI-1 specified the point-to-point in both blocking and non-blocking forms. Blocking routines in MPI do not continue execution in a program until the message has completed, whereas non-blocking routines allow execution to continue until an explicit blocking statement after the message is encountered. In MPI-2, functionality for parallel I/O and the first one-sided routines were added known as remote memory access (RMA) was added. Unfortunately, the one-sided routines were not well-received by the HPC community, so MPI-3 created a new RMA standard in 2008 (Message Passing Interface Forum, 2012). Another major feature in MPI 3.0 was support for non-blocking collective routines. The latest version of the MPI standard is MPI-3.1 which was , released in 2015. MPI has interfaces for both Fortran, C, and C++, however the C++ interface has been depreciated as of MPI-3. Popular current implementations of MPI are OpenMPI, made by the Open MPI Project, and MPICH, made by Argonne National Laboratory (Graham and Woodall, 2006) (Gropp, 2002). A final example of Hello World writen in C with MPI is given below.

Listing 1.3 Hello World with MPI

```
1  #include <mpi.h>
2  #include <stdio.h>
3  int main(int argc, char** argv)
4  {
5    int world_rank, world_size;
6    MPI_Init(&argc, &argv);
7    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
8    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
9    printf("Hello world from rank %d out of %d processors\n", world_rank, world_size);
10   MPI_Finalize();
11 }
```

## 1.4 References

Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T., and Wagener, J. L. (1992). *Fortran 90 Handbook: Complete ANSI/ISO Reference*. McGraw-Hill, Inc., New York, NY, USA.

Bachan, J., Bonachea, D., Hargrove, P. H., Hofmeyr, S., Jacquelin, M., Kamil, A., van Straalen, B., and Baden, S. B. (2017). The upc++ pgas library for exascale computing. In *Proceedings of the Second Annual PGAS Applications Workshop*, PAW17, pages 7:1–7:4, New York, NY, USA. ACM.

Backus, J. (1981). The history of fortran i, ii, and iii. In Wexelblat, R. L., editor, *History of Programming Languages*, pages 25–74. ACM, New York, NY, USA.

Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L., and Stokes, R. A. (1968). The illiac iv computer. *IEEE Trans. Comput.*, 17(8):746–757.

Barney, B. (2011). A "hello world" pthreads program. https://computing.llnl.gov/tutorials/pthreads/samples/hello.c.

Bergstein, B. (2007). Fortran created john backus dies. http://www.nbcnews.com/id/17704662/ns/technology_and_science-tech_and_gadgets/t/fortran-creator-john-backus-dies.

Bonachea, D. (2013). Upc language and library specifications, version 1.3. Technical report, Lawrence Berkeley National Lab.

Booch, G. (n.d.). John backus. https://amturing.acm.org/award_winners/backus_0703524.cfm.

Brainerd, W. (1978). Fortran 77. *Commun. ACM*, 21(10):806–820.

Cole, C. (n.d.). The remington rand univac larc. http://www.computer-history.info/Page4.dir/pages/LARC.dir/LARC.Cole.html. Accessed: 10/04/2019.

Companies using Fortran (n.d.). Companies using fortran. https://enlyft.com/tech/products/fortran.

Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55.

Darema, F., George, D., Norton, V., and Pfister, G. (1988). A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11 – 24.

Dongarra, J. (1988). The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, UK. Springer-Verlag.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

Graham, R. L. and Woodall, Timothy S.and Squyres, J. M. (2006). Open mpi: A flexible high performance mpi. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg. Springer Berlin Heidelberg.

Green500 (2007). The green500 list - november 2007. http://www.green500.org/lists/green200711/.

Green500 (2019). The green500 list - june 2019. http://www.green500.org/lists/green2019/6/.

Gropp, W. (2002). Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK. Springer-Verlag.

Hillis, W. D., editor (1986). *The Connection Machine*. MIT Press, Cambridge, MA, USA.

IBM Archives (n.d.). 704 data processing system. https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html. Accessed: 10/04/2019.

IEEE (1996). Ieee 1003.1c-1995 - standard for information technology–portable operating system interface (posix(r)) - system application program interface (api) amendment 2: Threads extension (c language). Technical report, IEEE Standards Association.

IPSJ Computer Museum (n.d.). Links-1 computer graphics system. http://museum.ipsj.or.jp/en/computer/other/0013.html. Accessed: 10/04/2019.

journal of Research, I. and staff, D. (2008). Overview of the ibm blue gene/p project. *IBM J. Res. Dev.*, 52(1/2):199–220.

Kennedy, K., Koelbel, C., and Zima, H. (2011). The rise and fall of high performance fortran. *Commun. ACM*, 54(11):74–82.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55.

McCorkle, M. L. (2018). Ornl launches summit supercomputer. https://www.ornl.gov/news/ornl-launches-summit-supercomputer.

Message Passing Interface Forum (1994). Mpi: A message-passing interface standard. Technical report, Message Passing Interface Forum, Knoxville, TN, USA.

Message Passing Interface Forum (2012). MPI: A Message-Passing Interface Standard Version 3.0. Technical report, Message Passing Interface Forum, Knoxville, TN, USA.

Numrich, R. W. and Reid, J. (1998). Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31.

OpenMP Architecture Review Board (2008). Openmp application program interface version 3.0 may 2008. Technical report, OpenMP Architecture Review Board.

Patterson, D. A. and Hennessy, J. L. (2013). *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Rahman, R. (2013). *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition.

Reid, J. (2007). The new features of fortran 2003. *SIGPLAN Fortran Forum*, 26(1):10–33.

Reid, J. (2014). The new features of fortran 2008. *SIGPLAN Fortran Forum*, 33(2):21–37.

Reid, J. (2018). The new features of fortran 2018. *SIGPLAN Fortran Forum*, 37(1):5–43.

Ritchie, D. M. (1996). The development of the c programming language. In Bergin, Jr., T. J. and Gibson, Jr., R. G., editors, *History of Programming languages—II*, pages 671–698. ACM, New York, NY, USA.

Thornton, J. E. (1980). The cdc 6600 project. *Annals of the History of Computing*, 2(4):338–348.

TIOBE Index (2019). Tiobe index for october 2019. https://www.tiobe.com/tiobe-index/.

TOP500 Project (1997). TOP500 List - June 1997. Accessed: 10/06/2019.

TOP500 Project (2019). TOP500 List - June 2019 . Accessed: 10/06/2019.

# CHAPTER 2. USING MACHINE LEARNING TO IMPROVE SHARED-MEMORY GENERAL MATRIX-MATRIX MULTIPLICATION

An extended paper accepted by *PDPTA'18 - The 24th International Conference on Parallel and Distributed Processing Techniques and Applications*

Brandon M. Groth and Glenn R. Luecke

## 2.1 Abstract

General matrix-matrix multiplication (GEMM) is frequently used to train neural networks for machine learning applications. For example, the backpropagation algorithm, which is implemented via matrix multiplication, is used to train feed-forward neural networks. The GEMM algorithm in Intel's Math Kernel Library (MKL) sometimes does not perform well on matrices required by backpropagation. To address this problem, the authors selected six GEMM algorithms and compared their performance to Intel's GEMM. This was accomplished by using a multi-class algorithm selector for finding the fastest of these seven algorithms. The algorithm selection was shown to be 93% accurate and the maximum speedup of non-MKL algorithms over MKL ranged from 3.77x to 11.22x. Our library, ML_GEMM, gave a maximum speedup of 23.83x over MKL.

## 2.2 Introduction

The Basic Linear Algebra Subprograms (BLAS) were developed to provide standard building blocks for performing basic vector and matrix operations (Lawson et al., 1979). These subprograms were designed to be implemented on various hardware to facilitate the fast and portable computation of linear equations solvers, matrix decompositions, eigenvalue solvers, etc. In particular, the subroutine called General Matrix Multiplication (GEMM) is the BLAS routine used for computing a variety of matrix-matrix multiplications. In this paper, the authors consider only single precision

matrix-matrix multiplication (SGEMM) to compute $C \leftarrow \alpha AB + \beta C$ with $\alpha, \beta \neq 0$. GEMM is used extensively in scientific computing and more recently in machine learning (e.g. training of neural networks). Computer vendors provide optimized versions of GEMM for their hardware, such as Intel's Math Kernel Library (MKL). Intel's MKL provides both serial and multi-threaded versions of GEMM. Unfortunately, at this time, GEMM in Intel's MKL does not provide good performance for some matrices. This is problematic for some fields in machine learning. For example, deep learning uses these highly rectangular matrices in training feed-forward neural networks (Keuper and Preundt, 2016). The purpose of this work is to provide a new GEMM library call, named ML_GEMM, that overcomes some of these performance problems on shared-memory computers. This is accomplished by comparing seven algorithms of GEMM (including Intels GEMM) and then use machine learning techniques to decide which algorithm gives the best performance for a given matrix dimension.

## 2.3 Background

### 2.3.1 Code Optimizations for Matrix Multiplication

For the matrices $A \in \mathbb{C}^{m\times k}$ and $B \in \mathbb{C}^{k\times n}$, the matrix multiplication operation $AB$ is defined as

$$(AB)_{ij} = \sum_{l=1}^{k} A_{il}B_{lj} \text{ for } i = 1 : m,\ j = 1 : n. \tag{2.1}$$

The matrix multiplication kernel is classified as requiring $\mathcal{O}(2mnk)$ floating point operations (flops) on $\mathcal{O}(mk + kn)$ elements to compute. In pseudo-code using row-order, this amounts to

Listing 2.1 Matrix Multiplication Algorithm

```
1 for i=1:m
2   for j=1:n
3     sum = 0;
4     for l=1:k
5       sum += a(i,l)*b(l,j);
6     c(i,j) = sum
```

A good first optimization that doesn't drastically change the code is loop unrolling. Loop unrolling is the process of expanding a basic block inside a loop by removing branching statements. A basic block is a sequence of code statements that contain no branching statements other than at the entry or exit of the block. An example of loop unrolling can be demonstrated by the following transformation to the inner `l` loop:

Listing 2.2 Simple Unrolled Loop

```
1  for l=1:x:k
2    sum += a(i,l)*b(l,j);
3    sum += a(i,l+1)*b(l+1,j);
4    ...
5    sum += a(i,l+x-1)*b(l+x-1,j);
```

where `x` is the unroll distance and should be chosen such that `k` is an integer multiple of `x`. In the example, the number of if statements executed in the for loop will be reduced from k to k/x. Since loop unrolling expands the basic block is expanded inside the for loop, there is more opportunities for pipelining machine instructions. The case where `k` is not an integer multiple of `x` can be transformed as

Listing 2.3 Full Unrolled Loop

```
1  unroll_full_blocks = k / x;
2  unroll_remainder_block = k % x;
3
4  while --unroll_full_blocks > 0
5    sum += a(i,l)*b(l,j);
6    sum += a(i,l+1)*b(l+1,j);
7    ...
8    sum += a(i,l+x-1)*b(l+x-1,j);
9
10 switch unroll_remainder_block
11   case x-1: sum += a(i,l+x-1)*b(l+x-1,j);
12   ...
```

```
13    case 1: sum += a(i,l+1)*b(l+1,j);
14    case 0: ; // do nothing
```

The switch statement utilizes fall-through as no case contains a break. This transformation is known as an untangled Duff's device. However, there are some caveats that should be considered when unrolling loops. The lines of code to maintain, debug, ect. becomes quite large in comparison to the original loop. Also, modern compilers are capable to do this transformation automatically. Compiler-based unrolling may include more optimizations beyond what would normally be coded by the user, such as the selection of the good unroll distance.

Another common optimization is known as tile blocking. Since dense matrix multiplication has fewer memory accesses compared to the number of flops, we can hold on to sections of each matrix until we are done with them. Given a computer with `M` bytes of cache and `b` bytes per cache line, we want to divide matrices $A$ and $B$ into $\sqrt{M} \times \sqrt{M}$ blocks (Hennessy and Patterson, 2017). The following code achieves this by adding three blocking loops using tile size $T = \mathcal{O}\left(\sqrt{M}\right)$:

Listing 2.4 Blocked Matrix Multiplication Algorithm

```
1  for ii = 1:T:m
2  for jj = 1:T:n
3  for ll = 1:T:k
4    for i = ii:min(ii+T, m)
5      for j = jj:min(jj+T, n)
6        sum = 0;
7        for l = ll:min(ll+T, k)
8          sum += a(i,l)*b(l,j);
9        c(i,j) += sum;
```

In the traditional row-order matrix multiplication, a cache miss can occur at every access of matrix B. So we have $\mathcal{O}(mnk)$ cache misses at the worst case, which can dominate the performance of the kernel since cache miss resolution can take 100's of CPU clock cycles. In an ideal cache model, the number of cache misses with tile blocking is reduced to $\mathcal{O}\left(\frac{mnk}{b\sqrt{M}}\right)$, which allow the arithmetic operations to dominate the kernel rather than cache misses. Currently, modern compilers are also

able to do this transformation! Intel compilers offer a pragma/directive called `block_loop` which replaces the original non-blocked loop with a blocked loop at compile time (Intel, 2019b) (Intel, 2019a).

### 2.3.2 Matrix Multiplication - Recursion

Another class of matrix multiplication algorithms exist that exploit the Divide-and-Conquer paradigm using submatrix blocks of $A$ and $B$. By using the definitions

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{2.2}$$

where $A$, $B$, and $C$ are are $2^n \times 2^n$ square matrices and $A_{ij}, B_{i,j}$, and $C_{ij}$ are $2^{n-1} \times 2^{n-1}$ square matrices, then the matrix multiplication becomes

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \tag{2.3}$$

Here, there are 8 submatrix multiplications and 4 sub-matrix additions. Unfortunately, this decomposition does not offer any improvements over the standard loop-based matrix multiplication. For the recursion, the original matrices $A$ and $B$ are halved in size to make the submatrices $A_{ij}$ and $B_{ij}$ until they are small enough to be computed as a base case, which is often a loop-based matrix multiplication. In the general case where the matrices are not square or the matrix dimensions are not even, the submatrices with have different dimensions. By setting $maxDim = \max(m, n, k)$, it has been shown that splitting the matrices along $maxDim$ and stopping recursion after $maxDim$ reaches a threshold leads to improved performance over 2.3. This version of recursion can be demonstrated in the following psuedo-code.

Listing 2.5 Matrix Multiplication Recursion

```
1 function MM_Recursion(A,B,C,m,n,k)
2   if max(m,n,k) < threshold
3     return C = AB via loop-based matrix multiply
4   else
```

```
5      // Rows of C and A are longest - split into 2
6      if max(m,n,k) == m
7         Let A = (A_1; A_2) with m_top = m/2 and m_bot = m - m/2.
8         Create C_1 with dimensions m_top × n and C_2 with dimensions m_bot × n.
9         C_1 = MM_Recursion(A_1, B, m_top, n, k)
10        C_2 = MM_Recursion(A_2, B, m_bot, n, k)
11        return (C_1; C_2)
12     // Columns of C and B are longest - split into 2
13     else if max(m,n,k) == n
14        Let B = (B_1  B_2) and with n_left = n/2 and n_right = n - n/2.
15        Create C_1 with dimensions m × n_left and C_2 with dimensions m × n_right.
16        C_1 = MM_Recursion(A, B_1, m, n_left, k)
17        C_2 = MM_Recursion(A, B_2, m, n_right, k)
18        return C = (C_1  C_2)
19     // Both columns of A and rows of B are split into 2 ; C remains same size
20     else
21        Let A = (A_1  A_2), B = (B_1; B_2) with k_1 = k/2, and k_2 = k - k/2.
22        Create C_1 and C_2 with dimensions m × n.
23        C_1 = MM_Recursion(A_1, B_1, C_1, m, n, k_1)
24        C_2 = MM_Recursion(A_2, B_2, C_2, m, n, k_2)
25        return C = C_1 + C_2
26  end function
```

---

Strassen (1969) took a different approach by using a clever combinations of the submatrices to reduce one submatrix multiply, while adding several submatrix additions and subtractions. The Strassen algorithm defines following matrices (assuming the matrix multiplications are defined):

- $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
- $M_2 = (A_{21} + A_{22})B_{11}$

- $M_3 = A_{11}(B_{12} - B_{22})$
- $M_4 = A_{22}(B_{21} - B_{11})$
- $M_5 = (A_{11} + A_{12})B_{11}$
- $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
- $M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$.

Then the submatrices of $C$ are computed as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}. \tag{2.4}$$

The Strassen Algorithm uses 7 submatrix multiplications and 18 submatrix additions/subtractions. A major caveat of the Strassen algorithm is that it requires additional memory to store each intermediate $M_i$ matrix and the recursion doesn't perform well for “small” matrices. New work has been done to improve the Strassen algorithm that reduces these restrictions and has been shown to competitive with loop-based matrix multiplication (Huang et al., 2016).

From the Master Theorem for divide-and-conquer recursion, the Strassen Algorithm has a computational complexity of $\approx \mathcal{O}\left(n^{2.8074}\right)$ (Bentley et al., 1980). Additional methods that built on Strassen's work have pushed the computational complexity down to $\approx \mathcal{O}\left(n^{2.3729}\right)$ (Coppersmith and Winograd, 1987) (Le Gall, 2014). However, the Big-O constants associated these improvements have been so large that it has been impractical to implement these algorithms in practice (Le Gall, 2014).

### 2.3.3 Basic Linear Algebra Subprograms

The Basic Linear Algebra Subprograms (BLAS) are a collection of low-level kernels commonly used in linear algebra computations (Lawson et al., 1979). The original implementation of BLAS was released in 1979, and is known as a ”reference implementation”. Furthermore, the BLAS Technical (BLAST) Forum standardized the interface for other to implement (Dongarra, 2002).

Some notable BLAS libraries that are available today are: Accelerate (Apple), ATLAS (open source: MPL), cuBLAS (NVIDIA), EigenBLAS (open source: BSD), GotoBLAS (open source: BSD), Math Kernel Library (Intel), and Netlib BLAS/CBLAS (public domain). In recent years, chipmakers such as Intel and NVIDIA have provided BLAS implementations that are optimized for their respective hardware offerings. Often, the hardware-optimized BLAS libraries offers significant performance improvements over base implementations that make no assumptions on hardware capabilities.

BLAS implementations typically have three levels, where each level is categorized by different data structures and algorithmic complexities. The three levels of BLAS routines are defined as:

- Level 1 - vector-vector compute kernels,
- Level 2 - matrix-vector compute kernels and matrix-vector equation solvers,
- Level 3 - matrix-matrix compute kernels and matrix-matrix equation solvers.

While there are several versions of matrix multiplication in BLAS, the most general version is called General Matrix-Matrix multiplication, denoted as GEMM. To understand what GEMM computes, let $OP_i(X)$ define one of the following matrix functions: the identity operation $X$, the matrix transpose $X^T$, or the matrix conjugate transpose $X^H$. Furthermore, let $\alpha, \beta \in \mathbb{C}$, $OP_1(A) \in \mathbb{C}^{m\times k}$, $OP_2(B) \in \mathbb{C}^{k\times n}$, and $C \in \mathbb{C}^{m\times n}$. The GEMM routine computes

$$C \leftarrow \alpha OP_1(A)OP_2(B) + \beta C. \tag{2.5}$$

The interface of BLAS GEMM is given as

```
<type>GEMM(transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
```

where `transa` and `transb` are character codes to compute $OP_1(A)$ and $OP_2(B)$ from $A$ and $B$, and `lda`, `ldb`, and `ldc` are the leading dimensions of matrices $A$, $B$, and $C$ respectively. The leading dimension of the matrix is defined as the number of rows of the matrix $X$ prior to transforming to $OP(X)$. Lastly, the type prefix of the GEMM interface corresponds to the data type being used for all constants and matrices. These types are:

- Single Precision Real: SGEMM (32 bits),

- Double Precision Real: DGEMM (64 bits),
- Single Precision Complex: CGEMM (32+32 bits),
- Double Precision Complex: ZGEMM (64+64 bits).

### 2.3.4 Supervised Learning

Supervised learning form of machine learning used in artificial intelligence to generate a mapping from data examples to an output. Given a data input $x \in \mathbb{R}^m$ and output label $y\ in \mathbb{R}^n$, we assume there is some unknown function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ such that $f(x) = y$. An input-output pair $(x, y)$ is called a training example, and a training set is comprised of many input-output pairs $E = \{(x_i, y_i)\}_{i=1}^N$ (also called a data set to avoid confusion in later definitions). The goal of supervised learning is to learn a hypothesis function $h$ from an hypothesis space $H$ using the training set so that $h(x_i) \approx y_i$ for $i = 1, .., N$. The following are some classical supervised learning algorithms:

- Linear Regression: Learn a linear function that fits the output.
- Logistic Regression: Learn a soft-threshold function that fits the output.
- Support Vector Machines: Separates a dimensional space into decision boundaries (Cortes and Vapnik, 1995).
- Artificial Neural Networks: Use a system of artificial neurons to fit the output.
- Decision Trees: Use a tree of Boolean decision nodes to separate outputs.
- Bayesian Classification: Use Bayes theorem to guide maximum likelihood of an output.
- Genetic Programming: Use random evolution to generate fit models (Koza, 1990).
- Ensembles: Use many different models to vote for the correct output.

Often, the search to find an optimal hypothesis function reduces to an optimization problem. To define this problem, first let $f(x) = y$, $h(x) = \hat{y}$, and let $L$ be a loss function such that

$$L(x, y, \hat{y}) = \mathit{Utility}(\text{use } y \mid x) - \mathit{Utility}(\text{use } \hat{y} \mid x). \tag{2.6}$$

Some popular choices of $L$ are the $L_1$ or $L_2$ norms, as well as an indicator function $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1. To define loss in terms of the training set $E$, we use the Empirical Loss function given a loss function $L$ and training set $E$

$$\mathit{EmpiricalLoss}_{L,E}(h) = \frac{1}{|E|} \sum_{(x,y) \in E} L(y, h(x)) \tag{2.7}$$

Lastly, there is often a trade-off between making a model simple versus making a model overly complex. From Occam's razor principle, we want to keep models as simple as possible, as they tend to generalize new information better. By adding a weighted complexity function to the Empirical Loss, we arrive at a general Cost function

$$\mathit{Cost}_{L,E,\lambda}(h) = \mathit{EmpiricalLoss}_{L,E}(h) + \lambda\,\mathit{Complexity}(h) \tag{2.8}$$

Reducing complexity in a model is referred to as regularization. Adding regularization to a model incurs a computational cost based on the type of complexity being reduced, but can impose nice properties to the final model such as smoothness or bounds on the solution. Popular choices of regularization are the squared $L_1$ or $L_2$ norms, as well as specialized functions or operations based on $H$.

The optimization problem to find the best hypothesis function in $H$ is defined as

$$h^* = \underset{h \in H}{\arg\min}\ \mathit{Cost}_{L,E,\lambda}(h) \tag{2.9}$$

The optimization problem is often solved using gradient descent methods via derivatives of $\mathit{Cost}(h)$, or more specifically the gradient of the parameters $w$ that define the hypothesis function $h := h_w$. In the context of supervised learning, new examples can be streamed individually from the training set. Due to this, varying strategies have been proposed that can speed up convergence such as mini-batch gradient descent, as well as stochastic gradient descent (Kiefer and Wolfowitz, 1952).

Instead of finding $\nabla_w Cost(h_w)$ across $E$, mini-batch uses a subset $\hat{E} \subset E$ per gradient step, while stochastic gradient descent selects a random training example $(x, y) \in E$ per gradient step. The difference between the three variants of gradient descent is a trade-off between the size of gradient computation versus how many times the parameter vector is updated. Furthermore, the convergence rate of each gradient descent version may be different pending on the properties of the cost function.

Listing 2.6 Gradient Descent

```
1 w ← random initialization
2 until w converges
3   w ← w − η∇_w Cost_{L,E,λ}(h_w)
```

Listing 2.7 Mini-batch Gradient Descent

```
1 w ← random initialization
2 Split E = E_1 ∪ ... ∪ E_k with E_i ∩ E_j = ∅
3 until w converges
4   for i = 1:k
5     w ← w − η∇_w Cost_{L,E_i,λ}(h_w)
```

Listing 2.8 Stochastic Gradient Descent

```
1 w ← random initialization
2 until w converges
3   for i = 1:|E|
4     Select Ê = (x,y) ∈ E randomly
5     w ← w − η∇_w Cost_{L,Ê,λ}(h_w)
```

### 2.3.5 Model Training and Evaluation

Another consideration in supervised learning is model training and evaluation. In most supervised learning problems, it is infeasible to store or stream a data set that represents the entirety of all possible data. Since a model $h$ can't learn something it has never seen, $E$ should be large enough

to contain some information about all desired features to be captured. In order to ensure that $h$ is indeed learning generalizations from the data set, it is customary to split $E$ into three distinct sets called a training set, validation set, and test set ($E = E_{train} \cup E_{validation} \cup E_{test}$). $E_{train}$ is used to learn the model parameters $w$ in $h_w$, $E_{validation}$ is used to determine the parameters unassociated with $h$ called hyperparameters for an intermediate model, and $E_{test}$ is used to determine the performance of the final model via an accuracy metric. Hyperparameters are used to tune models and that must be set before training $h$ has started. Two examples of a hyperparameters are $\lambda$ used in $Cost(h)$ or the learning rate $\eta$ in gradient descent. While both $\lambda$ or $\eta$ will affect the training of $h$, the final model will not be associated with either parameter. Some models do not need hyperparameters and hence $E_{validation} = \emptyset$, while other models are dependent on hyperparameters to be trained in a reasonable amount of time or perform well. When validation is used, a portion of the data set is set aside to do a parameter search for the best set of hyperparameters. However, this can add a considerable amount of computation to the training process, as each new set of hyperparameters amounts to retraining $h_w$.

Furthermore, there are strategies to train $h$ outside hyperparameter search involving $E_{train}$. K-fold cross-validation divides $E_{train}$ into $k$ subsets, then learns $k$ intermediate models by withholding one subset each iteration. So if $E_{train} = E_{train}^{(1)} \cup \ldots \cup E_{train}^{(k)}$, where each $E_{train}^{(i)}$ has approximately the same size and class distribution, then each model $h_w^{(i)}$ is trained on set $E_{train} - E_{train}^{(i)}$. Afterward, an performance metric is used on $h_w^{(i)}$ to report an averaged cross-validation performance metric across all $k$ models using the withheld training data as a validation set:

$$Performance_{cv} = \frac{1}{k} \sum_{i=1}^{k} Performance\left(h_w^{(i)}, E_{train}^{(i)}\right). \tag{2.10}$$

Usually, after cross-validation is completed, all models $h_w^{(i)}$ are discarded and we train the final model $h_w$ on the entire training set $E_{train}$ using $Performance_{cv}$ as the estimated accuracy of unseen data for the model. There is also an extension called repeated k-fold cross-validation that repeats the k-fold cross validation process $r$ times with $E_{train}$ randomly shuffled each time. The estimated performance metric is reported over an average of $r \times k$ intermediate models $h_w^{(i,j)}$, and the final model is trained on the entire $E_{train}$ data set.

In the case of the accuracy computation, there are many ways to determine how well a model performs. In classification, the a common data structure to analyze performance is a confusion matrix. A confusion matrix holds the number of predictions $h(x)$ against the actual value $y$ in a row-column format. Let $[\cdot]$ be an Iverson bracket defined as $[p] = 1$ if $p$ is true, and 0 otherwise. Let $C \in \mathbb{R}^{c \times c}$ be a confusion matrix with $c$ distinct classes evaluating the performance of set $\hat{E} \subset E$, then each element of $C$ will be calculated as

$$C_{ij} = \sum_{(x,y) \in \hat{E}} [h(x) = i \wedge y = j]. \tag{2.11}$$

The diagonal of $C$ signifies when $h(x) = y$, or when the classifier predicted the correct output, while off-diagonal entries of $C$ all signify wrong predictions. A true positive for class $c_k$ is when $h(x) = c_k$ and $y = c_k$. The number of true positives for class $C_k$ $(TP_k)$ is given by $TP_k = C_{kk}$. For $\left|\hat{E}\right| = N$, the accuracy of $h$ on $\hat{E}$ is computed as

$$Accuracy = \frac{1}{N} \sum_{i=1}^{c} TP_i. \tag{2.12}$$

A false negative occurs for class $c_k$ when $h(x) = c_k$, but $y \neq c_k$. Similarly, a false positive occurs for class $c_k$ when $y = c_k$, but $h(x) \neq c_k$. The number of false negatives $(FN_k)$ and false positives $(FP_k)$ for class $c_k$ are computed as

$$FN_k = \sum_{j=1, j \neq k}^{c} C_{kj} \;\text{ and }\; FP_k = \sum_{i=1, i \neq k}^{c} C_{ik} \tag{2.13}$$

Popular quotients for identifying class accuracy are precision and recall. Precision for a class is the proportion of correct classifications over all predicted classifications. Recall for a class is the proportion of correct classifications over all actual classifications. The equations for precision and recall for class $c_k$ are given as

$$Precision_k = \frac{TP_k}{TP_k + FP_k} \;\text{ and }\; Recall_k = \frac{TP_k}{TP_k + FN_k}. \tag{2.14}$$

A popular performance metric that uses the precision and recall is called F-measure. A generalized version of F-measure is given as

$$F_\beta = \frac{(1+\beta^2)\, Precision\, Recall}{(\beta^2\, precision) + Recall} = \frac{(1+\beta^2) TP}{(1+\beta^2) TP + \beta^2 FN + FP} \tag{2.15}$$

where $\beta > 0$. Regular choices for $\beta$ are $\beta = 1$, giving the harmonic mean of precision and recall (also known as $F_1$-score), $\beta = 2$ placing more emphasis on correctly identifying false negatives, and $\beta = .5$ placing less emphasis on correctly identifying false negatives.

A similar performance metric to F-measure is called the Cohen's Kappa (Cohen, 1960). Cohen's kappa is normally used to judge the accuracy of two human observers, but has been shown to be useful in classification when class distributions are skewed. In the case of supervised learning, one observer is the model $h$, while the other observer is the true value $y$. The computation of Cohen's Kappa ($\kappa$) involves the observed accuracy and predicted accuracy of both observers on $\hat{E} \subset E$ where

$$\kappa = \frac{P_{observed} - P_{predicted}}{1 - P_{predicted}}. \tag{2.16}$$

Here, $P_{observed} = Accuracy$ and $P_{predicted}$ is computed as

$$P_{predicted} = \frac{1}{N^2} \sum_{k=1}^{c} n_k^{row} n_k^{col} = \frac{1}{N^2} \sum_{k=1}^{c} \left( \sum_{j=1}^{c} C_{kj} \right) \left( \sum_{i=1}^{c} C_{ik} \right), \tag{2.17}$$

where $n_k^{row}$ and $n_k^{col}$ are the row or column sums of $C$ for class $k$ respectively. While there is no way to classify a "good" $\kappa$ value, a higher $\kappa$ value is better with the maximum bound of $\kappa \leq 1$. If $\kappa < 0$, then it is said that the model prediction $h(x)$ and output value $y$ have no agreement, implying the classifier is useless.

### 2.3.6 Support Vector Machines

Support Vector Machines (SVM) are supervised learning classifiers that construct hyperplanes in high- or infinite-dimensional spaces to separate data points for classification or regression. To separate the data points, the goal of SVMs is to create a hyperplane between two populations that is maximal, which is known as a maximal-margin hyperplane. This is possible if the data set is linearly separable. Let a data set $X \in \mathbb{R}^n$ be defined by two populations $X_1$ and $X_2$ such that $X = X_1 \cup X_2$. Then we say $X$ is linearly separable if $\exists\, w \in \mathbb{R}^n$ and $\exists\, k \in \mathbb{R}$ such that for $x \in X_1$, $w^T x > k$ and for $\hat{x} \in X_2$, $w^T \hat{x} < k$. In the context of binary classification, we define $E = \{(x_i, y_i)\}$ with $y_i \in \{-1, 1\}$ with $|E| = N$.

Any hyperplane in $\mathbb{R}^N$ can be described as $w^T x - b = 0$, where $w$ is a normal to the hyperplane. If $E$ is linearly separable, there exists two parallel hyperplanes such that all examples with $y_i = 1$ are on one side of the first hyperplane and all examples with $y_i = -1$ are on the other side of the second hyperplane. The equations that define the first and second hyperplanes are $w^T x - b = 1$ and $w^T x - b = -1$, and the distance between these hyperplanes is $2/\|w\|$. To define soft and hard margins for how close the nearest training example can be, we have two different margins to consider. Assuming $E$ is linearly separable, then the hard-margin SVM optimization problem is given as:

$$\text{Minimize } \|w\| \tag{2.18}$$

$$\text{subject to } y_i(w^T x_i - b) \geq 1 \text{ for } i = 1 : N \tag{2.19}$$

If $E$ is not linearly separable, then we use a soft-margin approach. Let $\zeta_i = \max(0, 1 - y_i(w^T x_i - b))$. If $\zeta_i = 0$, then feature $x_i$ lies on or above a hyperplane. If $\zeta_i \neq 0$, then $\zeta_i$ is proportional to the distance from the margin. The we have the soft-margin SVM optimization primal problem is given as

$$\text{Minimize } \frac{1}{N} \sum_{i=1}^{N} \zeta_i + \lambda \|w\|^2 \tag{2.20}$$

$$\text{subject to } y_i(w^T x_i - b) \geq 1 - \zeta_i \text{ and } \zeta_i \geq 0 \text{ for } i = 1 : N. \tag{2.21}$$

As $\lambda \to 0$, the norm of $w$ vanishes and the soft-margin problem behaves like the hard-margin problem. A dual problem can be formulated for the soft-margin optimization problem via Lagrange multipliers. Set the following

$$f(w, \zeta) = \frac{1}{N} \sum_{i=1}^{N} \zeta_i + \lambda \|w\|^2, \tag{2.22}$$

$$g_i(w) = y_i(w^T x_i - b) - 1 + \zeta_i \tag{2.23}$$

$$\mathcal{L}_{w,b,\zeta,s,t} = f(w, \zeta) + \sum_{i=1}^{N} s_i g_i(w) + \sum_{i=1}^{n} t_i \zeta_i \tag{2.24}$$

After solving $\nabla_{w,s,t}\mathcal{L} = 0$, applying KKT conditions, and using previous definitions, a dual quadratic programming formulation emerges in the form of

$$\text{Maximize } \sum_{i=1}^{N} c_i + \sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j c_i c_j (x_i \cdot x_j) \tag{2.25}$$

$$\text{subject to } \sum_{i=1}^{N} c_i y_i = 0 \text{ and } 0 \leq c_i \leq (2n\lambda)^{-1} \text{ for } i = 1 : N, \tag{2.26}$$

where $c_i$ are defined by $w = \sum_{i=1}^{N} c_i x_i y_i$ (Bishop, 2006).

To extend this formulation to higher dimensions, the so called "kernel trick" is used. Let $K(x, z) = \phi(x) \cdot \phi(z)$, where $\phi(\cdot)$ is a coordinate transform. Then the only differences between the dual formulation and the transformed formulation is that we replace the dot product $x_i \cdot x_j$ with $k(x_i, x_j)$ and variables $c_i$ are now defined by $w = \sum_{i=1}^{N} c_i y_i \phi(x_i)$. Some popular kernel choices are:

- Polynomial Kernel: $k(x, z) = (x \cdot z)^d$,
- Inhomogeneous Polynomial Kernel: $k(x, z) = (1 + x \cdot z)^d$,
- Gaussian Kernel: $k(x, z) = \exp\left(-\gamma \|x - z\|^2\right)$ where $\gamma = 1/(2\sigma^2)$,
- Hyperbolic Tangent: $k(x, z) = \tanh(\gamma x \cdot z - c)$ for some $\gamma, c > 0$.

While quadratic programming techniques can be used to solve the SVM, faster algorithms exist. One of the most popular SVM training methods is called sequential minimal optimization (SMO) (Platt, 1998). SMO uses two sequential Lagrange multipliers as a sub-problem that can be solved analytically, which avoids the quadratic programming approach entirely. This was a major improvement, as good quadratic solvers were often proprietary and expensive. Today, the SMO algorithm is freely available in the LIBSVM library (open source: BSD), as well as many others (Chang and Lin, 2011).

### 2.3.7 Decision Trees

Decision trees are another supervised learning model which create a tree data structure of binary decision nodes that path to a single leaf classification node. The decision tree acts as a function

$h(x) = \text{pathToLeaf}(x)$, where each decision node on the path to the leaf will run a test on a specific feature $x^{(j)}$ (note that this is indexing the $j$th element in vector $x$). The binary decisions can test if $x^{(j)}$ belongs to a certain class or has a specific value ($x^{(j)} == k$) to testing if $x^{(j)}$ lies within range of values using $<, \leq, >$, or $\geq$. One advantage that decision trees have as a machine learning model is that the decision nodes are able to be interrupted as opposed to be being mathematical "black boxes". Given a particular $x$, it is possible to ascertain $h(x)$ by following the path of the tree. Some popular decision tree algorithms available are Classification And Regression Tree (CART) (Breiman et al., 1984), Iterative Dichotomiser 3 (ID3) (Quinlan, 1986), C4.5/C5.0 (Quinlan, 1993), and Random Forests (Ho, 1995) (Breiman, 2001).

To create a decision tree from a data set, a greedy recursive algorithm is used instead of the gradient descent optimization approach. The pseudo-code for learning a decision is provided in Listing 2.9 (Russell and Norvig, 2009).

Listing 2.9 Decision Tree Learning Algorithm

```
1  FUNCTION DT_Algorithm(examples, attributes, default)
2  // Base Cases
3  if( examples empty ) return node(default)
4  else if( all examples have same label ) return node(label)
5  else if( attributes empty ) return node( Mode(examples) )
6  else
7    best_attribute ← Choose_Best_Attribute(attributes, examples)
8    tree ← New tree with best_attribute at root
9    for( each value v_i of best_attribute )
10     examples_v ← { examples with best_attribute = v_i }
11     subtree ← DT_Algorithm(examples_v, attributes-best_attribute, Mode(examples))
12     Add branch from tree to subtree with label v_i
13   return tree
14 END FUNCTION
```

The `Mode()` function returns the most common label in the examples, breaking ties randomly. The difference between the non-Random Forest algorithms lies in how the best attribute is selected from

the function `Choose_Best_Attribute()`, as well as extra features outside of model learning such as tree pruning.

For splitting by the best attribute attributes, the algorithms CART, ID3, and C4.5/C5.0 all differ by the selection criterion. The CART algorithm uses the Gini index to select the best variable. Let $A$ be a set of attributes and $a \in A$ have $d$ classes with proportional populations $p_1, \ldots, p_c$. The Gini index for $a \in A$ is given as

$$G(a) = 1 - \sum_{i=1}^{d} p_i^2 \tag{2.27}$$

The Gini index favors splitting attributes that have larger population proportions. This is because the maximum value of $G(a)$ is achieved when all classes are equally proportioned with population $1/d$:

$$G(a) = 1 - \sum_{i=1}^{d} \left(\frac{1}{d}\right)^2 = 1 - \frac{d}{d^2} = 1 - \frac{1}{d}. \tag{2.28}$$

The CART algorithm always splits the examples $E$ into two data sets $E_1$ and $E_2$. If $|E| = N$, $|E_1| = N_1$, and $|E_2| = N_2$, define the Gini Split as

$$GiniSplit(a) = \frac{N_1}{N} Gini(E_1) + \frac{N_2}{N} Gini(E_2). \tag{2.29}$$

The CART algorithm selects `best_attribute` $= \arg\min_{a \in A} GiniSplit(a)$.

ID3 and C4.5/C5.0 use a different metric from information theory called information gain. To start, the entropy of a random variable $V$ having values $v_i$ with probability $P(v_i)$ is

$$H(V) = -\sum_i P(v_i) \log_2 P(v_i).$$

This means a Boolean random variable $V$ with probability $q$ has entropy

$$H(V) = -q \log_2 q - (1-q) \log_2 (1-q) := B(q). \tag{2.30}$$

For a Boolean attribute $a \in A$, if there are $p$ positive examples (ie. example $e$ has attribute $e.a = true$) and $q$ negative examples by splitting $E$ by $a$, then the entropy is

$$H(a) = B\left(\frac{p}{p+n}\right). \tag{2.31}$$

Now assume that $a$ has $d$ distinct values $v_1, \ldots, v_d$ such the set of examples is split into disjoint subsets $E = E_1 \cup \ldots \cup E_d$ such that $E_i = \{e \in E \,|\, e.a = v_i\}$. If each $E_k$ has $p_k$ positive and $n_k$ negative examples, the remainder entropy of choosing $a \in A$ is defined as

$$Remainder(a) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right). \tag{2.32}$$

The information gained by choosing $a \in A$ is the expected reduction in entropy given as

$$Gain(a) = H(a) - Remainder(a). \tag{2.33}$$

The algorithm ID3 selects the `best_attribute` $= \arg\max_{a \in A} Gain(a)$. The C4.5/C5.0 algorithm adds an additional criteria that encourages having skewed population sizes similar to the CART algorithm. To do this, we can analyze the sets $E_1, \ldots, E_d$ using the following metric

$$Split(a) = -\sum_{k=1}^{d} \frac{p_k + n_k}{p + n} \log_2\left(\frac{p_k + n_k}{p + n}\right). \tag{2.34}$$

If $Split(a)$ is large, then $|E_i|$ are approximately equal and if $Split(a)$ is small, then $|E_i|$ are skewed. The information gain is then modified as a ratio

$$GainRatio(a) = \frac{Gain(a)}{Split(a)}. \tag{2.35}$$

The algorithms C4.5/C5.0 select `best_attribute` $= \arg\max_{a \in A} GainRatio(a)$.

Tree pruning is a process of replacing "unimportant" branches with leaf nodes after the decision tree has been created. Tree pruning is a form of regularization, as it reduces model complexity without much affect on model accuracy. Let $T$ be a tree with classification error $Error(T)$ and $Leafs(T)$ number of leaves, and let $T_M$ be a tree after pruning decision node $M$ from $T$. The cost of the tree $T$ can be defined as

$$Cost(T) = Error(T) + \lambda\, Leafs(T) \tag{2.36}$$

for some given $\lambda$. Than an algorithm to prune a decision node is given in Listing 2.10.

Listing 2.10 Decision Tree Pruning Algorithm

```
1  Traverse decision nodes M from the bottom to the top of tree T
2    T_M = Prune(T, M)
3    if( Cost(T_M) < Cost(T) ) T = T_M
```

## 2.4 Methodology

There are many ways to compute GEMM, such as a library call or by manually coding the algorithm. Memory stride, cache awareness, parallelism, and the compiler (including compiler options) can have a major impact on the algorithm's performance. The following are the seven algorithms used in this paper:

- Algorithm 1 - Basic
- Algorithm 2 - Basic Transpose
- Algorithm 3 - Dot
- Algorithm 4 - Dot Transpose
- Algorithm 5 - Matmul (Fortran Intrinsic)
- Algorithm 6 - MKL
- Algorithm 7 - MKL Transpose

See Section 2.8 for the code used for each algorithm. For the Basic and Dot algorithms, the order of the loop indices for $(AB)_{ij} = \sum_{l=1}^{k} A_{il}B_{lj}$ was chosen to give stride-1 memory for the columns of $B$.

The transpose versions of these methods use a double transpose on matrix $A$ to compute $C \leftarrow \alpha(A^T)^T B + \beta C$. The first transpose is computed by an out-of-place matrix transpose. The transpose of $A^T$ is done via opposite indexing (i.e. $A(i,j) = A^T(j,i)$) for Algorithms and 2 and 4. For Algorithm 7, the `transa = 'T'` option is used to transpose $A^T$ within SGEMM. These transpose variations are considered because they give stride-1 memory access for the rows of $A$ even though

they require additional memory. This allows compilers to use vectorization, which may give a speedup that outweighs the cost for computing a matrix transpose. Furthermore, a small number of elements are added to each dimension to avoid cache problems related to storing arrays in exact powers of 2 Center (2017). For this paper, the authors define the group of non-MKL algorithms to be Algorithms 1-5 and Algorithm 7. The MKL Transpose algorithm is included in the non-MKL group because the transpose modification differentiates it from a typical MKL call.

Due to increasing processor core counts, developers are using multi-threading APIs, such as OpenMP, to parallelize their applications (van der Pas et al., 2017). In this paper, the authors choose to use OpenMP for multi-threading our GEMM implementations.

The computing resource that we choose to use for our experiments was the Condo HPC Cluster located at Iowa State University. Each node that was used on Condo had two 2.6 GHz 8-Core Intel E5-2640 v3 processors (Haswell) with 128 GB of memory. The operating system used on Condo is Red Hat Enterprise Linux 7.3 and the compiler used for all Fortran code was Intel 16.0. While Intel 17.x compilers were available, a bug prevented the use of Matmul with more than one OpenMP thread using the `-qopt-matmul` compiler option. After consulting the Intel Fortran Compiler Developer Guide and Reference, the authors choose to use the following compiler options after experimentation:

```
ifort -parallel -mkl=parallel -qopt-matmul -qopenmp -Ofast -xHost -ipo <source>
```

To use OpenMP on our the nested matrix loops, we add the following before the do loops:

```
!$OMP PARALLEL DO SHARED(<A>,B,C) PRIVATE(<priv_vars>) SCHEDULE(static)
```

where `<A>` is $A$ or $A^T$ and `<priv_vars>` contain loop indices and temporary variables inside the do loops. The OpenMP environment variables used for experiments were `OMP_NUM_THREADS=16`, and `OMP_PROC_BIND=true`. Please see the OpenMP specification for details of these environment variables.

The scalability of all seven algorithms is given in Figures 2.1 and 2.2. From these figures, the fastest algorithm changes with the corresponding matrix dimensions.

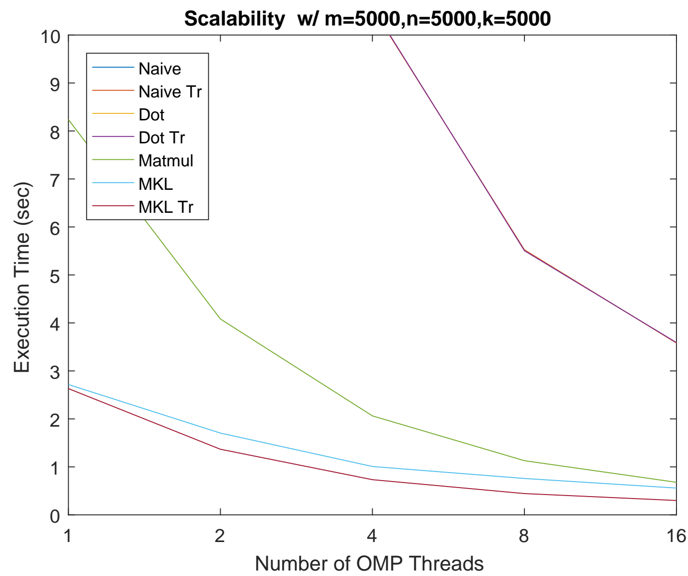


Figure 2.1 Algorithm Scalability with for Square GEMM

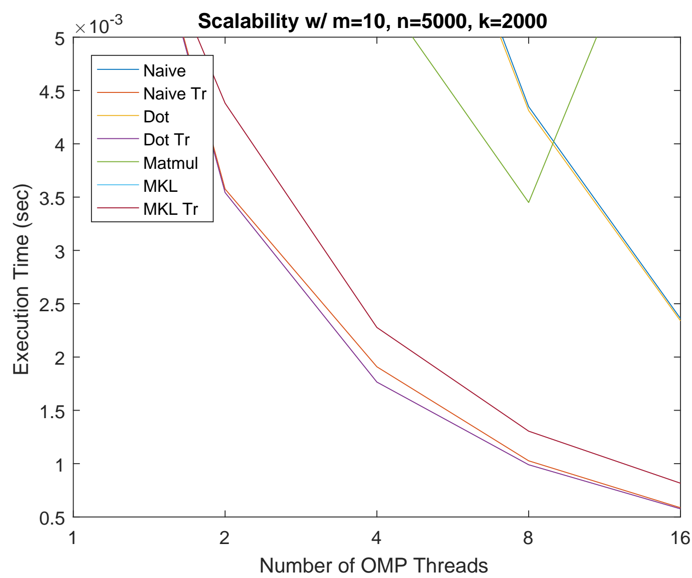


Figure 2.2 Algorithm Scalability with for Non-square GEMM

### 2.4.1 NUMA Considerations

Data locality in parallel programs can be a bottleneck in performance. Modern HPC nodes have Non-uniform Memory Access (NUMA) architectures, which mean that CPU's in a node share local memory via interconnects. In a NUMA architecture, CPU's that use their own local memory are faster than CPU's that have to reach out to non-local memory. Within the Linux operation system, "First-Touch" memory assignment is used to assign memory pages to the first thread that initializes a shared data object. Linux does not leverage NUMA architectures when doing First-Touch assignment to threads optimally; it must be done manually by the programmer. When entering an OpenMP region with thread 0 owning all data structures, every thread must access thread 0's memory bank. This can create memory bottlenecks. Most HPC nodes contain two sockets that each constitute a NUMA domain. Using poor memory allocation for threads that have to read/write to a different socket on a Haswell node can result in a 10-25% performance degradation. While tests using First Touch memory increased some of the GEMM implementation slightly (by 1-5%), First-Touch memory assignment would prevent the use of creating a subroutine interface for our GEMM classifier that could be swapped for current GEMM calls. This is because subroutines and functions can not be passed shared-memory arrays. If it is possible to in-line the GEMM subroutine where data initialization occurs, then First Touch access should be exploited. However, in-lining a GEMM routine would reduces portability and code-readability of an application.

In this project, the execution-time data was generated in Fortran, while data processing was done in R using the Caret package (Kuhn, 2012). Caret has support for a large number of machine learning algorithms, and does so with a common template for parameter search, training, testing, etc. The Caret framework allows users to train and test many different models within a single script.

Another consideration is how the compiler deals with known array or matrix dimensions. Often, when a compiler knows the dimensions of arrays or matrices at compile-time, it is able to use additional optimizations when using these data types. Some of these optimizations include cache blocking, pre-fetching, vectorization, loop splitting/peeling, etc. Since we are dynamically

allocating matrices $A, B$, and $C$ at run-time, the compiler can't use some of these optimizations. One approach to avoid this issue is to use a job script to loop through the matrix dimensions, compiling the code each time with a given matrix dimension (which is now known at compile-time), then submitting a job to the job queue. The following is pseduo-code used to launch set matrix dimensions via job scripts:

Listing 2.11 Data Collection Psuedo-Code

```
Get power mins/maxes from command line

do mPower = mPower_min, mPower_max
do nPower = nPower_min, nPower_max
do kPower = kPower_min, kPower_max
  Set m,n,k
  Allocate and initialize matrices
  do trial = 1,numTrials
    Flush cache
    Start OpenMP timer
    Compute parallel GEMM method
    End OpenMP timer
  end do
  Compute average time
  Restore C to original
  ...
  Write results to file
  Deallocate matrices
end do
```

## 2.5 Algorithm Selection

The data collected for this experiment were given as `m,n,k,algorithm`, where `m,n,k` are the matrix dimensions and `algorithm` is an integer label corresponding to the GEMM algorithm that gave the minimum average execution time. Let $S = \{2^3, 2^4, \ldots, 2^{16}\}$. In context of supervised

learning, the feature space is defined as $\vec{x} = (m, n, k) \in S^3$, and the target output is $y = algorithm \in \{1, 2, \ldots, 7\}$. This formulation defines a multi-class classification problem, where given input $\vec{x}$, we want to predict the target output $y$. Thus, for some classifier $f : S^3 \to \{1, 2, \ldots, 7\}$, we want to train $f$ such that for each training example $(\vec{x}_i, y_i)$, $f(\vec{x}_i) = y_i$.

Returning a prediction quickly is vital to our methods performance. For instance, say that $T_{Classify}, T_{MM}$, and $T_{MKL}$ are the times to perform the classification, the matrix multiplication chosen by the classifier, and the time to compute the Intel MKL GEMM method respectively. Ideally, when our classier will select a non-MKL algorithm, we want

$$T_{Classify} + T_{MM} \leq T_{MKL}. \tag{2.37}$$

If $T_{Classify}$ is small, then even when ML_GEMM classifies as MLK, there won't be much difference in performance. If $T_{Classify}$ is large, it would effectively render our ML_GEMM method unusable for any matrix multiplication that completed before $T_{Classify}$.

### 2.5.1 Data Set Analysis

For each GEMM algorithm, the authors created a $14 \times 14 \times 14$ cube of data for GEMM timings with matrix dimensions in $S$. At each index of this data cube, the best algorithm is labeled based on the lowest average execution time of the seven GEMM algorithms. The authors were able to successfully run 2640 of the 2744 possible grid points. The 104 missing grid points were due to insufficient memory to run the test.

To analyze when each algorithm was performing well on rectangular matrices, the authors looked at when matrices $A$ and $B$ were highly rectangular. A highly rectangular matrix was defined to be when the larger dimension was $2^5 = 32$ times larger than the smaller dimension. Since both matrices $A$ and $B$ could be rectangular, there are four cases: either $A$ or $B$ is rectangular, both are rectangular, or neither are rectangular. See Table 2.1 for the rectangular distribution for each algorithm. From the table, the non-MKL algorithms collectively account for 21.6% of cases when matrix $A$ is highly rectangular. Both Basic Transpose and Dot Transpose have a majority of their labeled cases in this category, while MKL Transpose has only one instance.

Table 2.2 presents a speedup analysis of the seven GEMM algorithms. Each row represents when the denominator algorithm was labeled the fastest, giving a minimum speedup of 1.00x. The authors compared each non-MKL algorithm to MKL (rows 1-6), and MKL was compared to the best non-MKL algorithm found (row 7). The non-MKL algorithms gave maximum speedup ranging from 2.82x-11.22x over MKL. Similarly, MKL gave a maximum speedup of 41.55x over our non-MKL algorithms.

Table 2.1 Rectangular Matrix Distribution

| Best Algorithm | Highly Rectangular | | | |
|---|---|---|---|---|
| | A | B | Both | None |
| Basic | 5 | 6 | 17 | 7 |
| Basic Transpose | 43 | 1 | 5 | 20 |
| Dot | 37 | 11 | 12 | 16 |
| Dot Transpose | 40 | 6 | 5 | 25 |
| Matmul | 0 | 0 | 0 | 15 |
| MKL | 456 | 532 | 323 | 990 |
| MKL Transpose | 1 | 35 | 14 | 18 |

Table 2.2 Speedup Results for GEMM Algorithms. Each row represents when the denominator GEMM algorithm was fastest.

| Speedup Computation | Speedup Bins | | | | Avg Speedup | Max Speedup |
|---|---|---|---|---|---|---|
| | 1.0-1.5 | 1.5-2.0 | 2.0-3.0 | >3.0 | | |
| MKL / Basic | 16 | 12 | 1 | 6 | 2.05 | 11.22 |
| MKL / Basic Transpose | 57 | 8 | 2 | 2 | 1.39 | 4.06 |
| MKL / Dot | 47 | 21 | 7 | 1 | 1.49 | 3.77 |
| MKL / Dot Transpose | 59 | 14 | 3 | 0 | 1.33 | 2.82 |
| MKL / Matmul | 13 | 1 | 0 | 1 | 1.42 | 4.25 |
| MKL / MKL Transpose | 60 | 4 | 3 | 1 | 4.16 | 7.23 |
| Best non-MKL / MKL | 1217 | 239 | 201 | 644 | 1.25 | 41.55 |

From the table, the MKL algorithm was optimal for a majority of cases, but the collection of simple algorithms were superior 12.8% of the time. Furthermore, the speedups of these algorithms over MKL is given in Table 2.2 when a non-MKL method was selected.

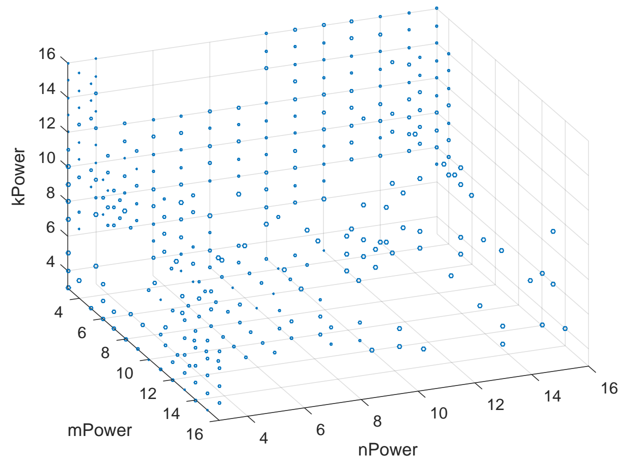


Figure 2.3 Non-MKL Data Locations 1

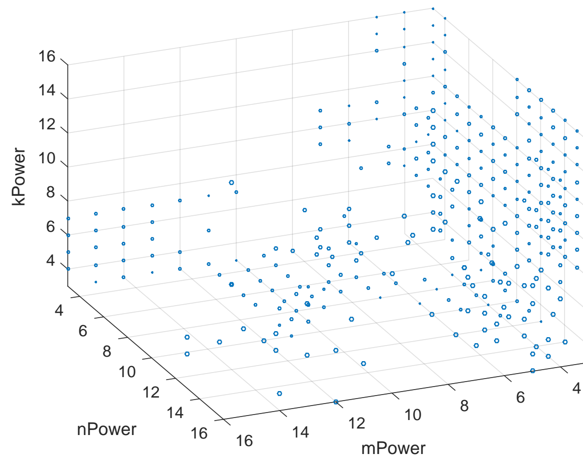


Figure 2.4 Non-MKL Data Locations 2

### 2.5.2 The C5.0 Classifier

Due to the computation constraints of generating the data set in $S^3$, the classifier will only see each training example $(\vec{x}_i, y_i)$ once. This is a problem for splitting the data set into distinct

training and test sets. Data splitting in this case will lower classification accuracy, especially in regions that classify as different algorithms. For our application, having a high accuracy classifier is more important than other considerations like over-fitting or bias, since picking the wrong GEMM algorithm can lead to poor performance. This will be done by training and testing with the entire data set.

The package we choose to use with Caret was C5.0, which learns a decision tree based on the data given (Kuhn, 2015). The C5.0 decision tree algorithm is an optimized version of the popular C4.5 algorithm. It has been shown to be orders of magnitude faster than C4.5 while producing equivalent or smaller trees RuleQuest (2018). A decision tree was generated via C5.0 on the data set, which is summarized in Table 2.3. The most important aspects of this tree is that it has an accuracy of 93% on the entire data set and can return a query in $2.99 \times 10^{-5}$ seconds on average. Thus, our classifier has a very small overhead. To choose a classifier that was accurately selecting the instances where non-MKL matrix multiplication methods were superior, the authors looked at the corresponding confusion matrix (Table 2.5). A confusion matrix shows the number correct classifications along the diagonal, while the number of wrong classifications are on off-diagonal entries for each column. From the Confusion matrix, the decision tree does a good job classifying Algorithms 1 through 4, but does poor job classifying Algorithms 5 and 7. While this isn't ideal, other classifiers tested from Carrot were unable to correctly classify non-MKL algorithms. This is because each non-MKL algorithms would be considered a "rare event" compared to MKL in the data set.

Table 2.3 C5.0 Decision Tree Properties

| Decision Tree Properties | Value |
|---|---|
| Decision Nodes | 44 |
| Min Depth | 2 |
| Max Depth | 8 |
| Avg Depth | 5.32 |
| Accuracy | 92.99% |
| Avg Query Time | 2.99e-5 s |

Table 2.4 C5.0 Decision Tree Class Accuracy

| Algorithm | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Accuracy | 78.9% | 66.7% | 80.3% | 58.1% | 80.0% | 95.4% | 68.6% |

Table 2.5 C5.0 Confusion Matrix. Rows indicate class predictions of the classifier $f(\vec{x}_i)$ and columns represent the true class values $y_i$.

| | $y_i$ | | | | | | |
|---|---|---|---|---|---|---|---|
| $f(\vec{x}_i)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 15 | 0 | 4 | 0 | 0 | 0 | 0 |
| 2 | 1 | 42 | 0 | 19 | 0 | 1 | 0 |
| 3 | 10 | 0 | 53 | 1 | 0 | 2 | 0 |
| 4 | 0 | 18 | 0 | 43 | 0 | 9 | 4 |
| 5 | 0 | 0 | 1 | 0 | 4 | 0 | 0 |
| 6 | 9 | 7 | 17 | 13 | 11 | 2287 | 53 |
| 7 | 0 | 2 | 1 | 0 | 0 | 2 | 11 |

### 2.5.3 ML_GEMM

Let ML_GEMM be the name of this algorithm selector implementing the C5.0 decision tree in Fortran. The authors tested the ML_GEMM against Intel's MKL using only a single time trial. The same grid points were used for this results phase. The results are given in Table 2.6. Since ML_GEMM can classify as MKL, it was important to know how many instances had near equal

execution times. This corresponds to the the 1.00-1.025x speedup bin for each method. Our library ML_GEMM was noticeably superior (>1.025x speedup) 10.6% of the time. The maximum speedups observed were 23.83x for ML_GEMM and 43.41x for MKL.

Listing 1: ML_GEMM Pseudo Code

```
1 SUBROUTINE ML_GEMM(m,n,k,alpha,beta,A,B,C)
2   Compute mPower,nPower,kPower
3   method=gemmClassifier(mPower,nPower,kPower)
4   SELECT CASE (method)
5     cases 1-7: Compute GEMM via Method
6     default: Compute GEMM via MKL
7   END SELECT
8 END SUBROUTINE ML_GEMM
```

Table 2.6 ML_GEMM vs MKL Speedup Analysis. Each row represents when the denominator algorithm was selected as fastest.

| Speedup Computation | Speedup Bins | | | | | Max Speedup |
|---|---|---|---|---|---|---|
| | 1.0-1.025 | 1.025-1.5 | 1.5-2.0 | 2.0-3.0 | >3.0 | |
| MKL / ML_GEMM | 265 | 225 | 32 | 14 | 7 | 23.83 |
| ML_GEMM / MKL | 710 | 1316 | 117 | 28 | 18 | 43.41 |

## 2.6 Related Work

Algorithm selection was a technique developed in 1976 to help define when a particular algorithm "works best" (Rice, 1976). In the past, developing a good selector required domain expertise to select subsets of the feature space for each algorithm used. Recently, developers are employing machine learning techniques to create algorithm selectors for various problems (such as the SAT problem (Lindauer et al., 2015)).

Parallel matrix-matrix multiplication algorithm development has been ongoing for decades. The SUMMA algorithm presented by van de Geijn and Watts is a well-known distributed algorithm that rivaled ScaLAPACK in 1995 (van de Geijn and Watts, 1995). More recent work uses recursion

and minimal communication, such as the CARMA algorithm created by Demmel (Demmel et al., 2013). CARMA uses divide-and-conquer techniques on blocks of the matrices to perform the matrix multiplication as opposed to traditional loops. This strategy was shown to have better performance than Intel's MKL in 2013.

To the author's knowledge, the first paper to use machine learning to aid matrix multiplication was by Spillinger et al. (Spillinger et al., 2015). Spillenger et al used the support vector machine (SVM) algorithm to choose a library to perform a dense matrix multiplication the fastest. The libraries that the authors choose to compare were Intel MKL and the CARMA algorithm.

Next, Shaohuai Shi et al. examined computing $C = AB^T$ on a GPU (Shi et al., 2017). Instead of choosing between two different libraries, the authors choose between using two versions of the cuBLAS GEMM algorithm. The first cuBLAS version transposes matrix $B$ inside GEMM, while the other version pre-transposed the matrix B, then used a standard GEMM matrix multiply. They then embedded their matrix multiply classifier within the Caffe deep learning framework to improve performance of image classification.

## 2.7 Conclusion

In this work, the authors have employed an algorithm selector developed via machine learning techniques to improve matrix-matrix multiplication. A multi-class classifier framework is provided to compute the General Matrix-matrix Multiplication (GEMM) in parallel on shared-memory computers. As a proof-of-concept, the authors selected six simple algorithms and compared them with Intel's MKL GEMM. These simple algorithms collectively outperformed MKL 12.8% of the time for matrix dimensions $m, n, k \in \{2^3, 2^4, \ldots, 2^{16}\}$. When non-MKL algorithms were selected, they exhibited an maximum speedup of 2.82-11.22x over MKL. Our algorithm selector, named ML_GEMM, was created with the C5.0 decision tree classification model to select the fastest algorithm for a given $m, n, k$. This algorithm selection was shown to be 93% accurate on the data set. ML_GEMM was shown to be superior to strictly using MKL 10.6% of the time, with a maximum speedup over MKL of 23.83x.

## 2.8 Appendix

Algorithm 2.12 Basic

```
1  !$OMP PARALLEL DO SHARED(A,B,C) PRIVATE(i,j,l,sum) SCHEDULE(static)
2  do j=1,n
3    do i=1,m
4      sum=0.0
5      do l=1,k
6        sum= sum+A(i,l)*B(l,j)
7      enddo
8      C(i,j)=alpha*sum+beta*C(i,j)
9    end do
10 end do
```

Algorithm 2.13 Basic Transpose

```
1  ALLOCATE(Atr(k,m))
2  Atr = TRANSPOSE(A)
3  !$OMP PARALLEL DO SHARED(Atr,B,C) PRIVATE(i,j,l,sum) SCHEDULE(static)
4  do j=1,n
5    do i=1,m
6      sum=0.0
7      do l=1,k
8        sum=sum+Atr(l,i)*B(l,j)
9      end do
10     C(i,j)=alpha*sum+beta*C(i,j)
11   end do
12 end do
13 DEALLOCATE(Atr)
```

Algorithm 2.14 Dot Product

```
1  !$OMP PARALLEL DO SHARED(A,B,C) PRIVATE(i,j) SCHEDULE(static)
2  do j=1,n
```

```
3    do i=1,m
4      C(i,j)=alpha*DOT_PRODUCT(A(i,1:k), B(1:k,j))+beta*C(i,j)
5    end do
6  end do
```

Algorithm 2.15 Dot Product Transpose

```
1  ALLOCATE(Atr(k,m))
2  Atr=TRANSPOSE(A)
3  !$OMP PARALLEL DO SHARED(Atr,B,C) PRIVATE(i,j) SCHEDULE(static)
4  do j=1,n
5    do i=1,m
6      C(i,j)=alpha*DOT_PRODUCT(Atr(1:k,i), B(1:k,j))+beta*C(i,j)
7    end do
8  end do
9  DEALLOCATE(Atr)
```

Algorithm 2.16 Matmul

```
1  ! Uses -qopt-matmul and -parallel -qopenmp compiler options
2  C=alpha*MATMUL(A,B)+beta*C
```

Algorithm 2.17 SGEMM

```
1  ! Uses -mkl=parallel compiler option
2  CALL SGEMM('N','N',m,n,k,alpha,A,m,B,k,beta,C,m)
```

Algorithm 2.18 SGEMM Transpose

```
1  ALLOCATE(Atr(k,m))
2  Atr=TRANSPOSE(A)
3  ! Uses -mkl=parallel compiler option
4  CALL SGEMM('T','N',m,n,k,alpha,Atr,k,B,k,beta,C,m)
5  DEALLOCATE(Atr)
```

**Acknowledgments**

The research reported in this paper is partially supported by the HPC@ISU equipment at Iowa State University, some of which has been purchased through funding provided by NSF under MRI grant number CNS 1229081 and CRI grant number 1205413.

## 2.9 References

Bentley, J. L., Haken, D., and Saxe, J. B. (1980). A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA.

Center, T. A. C. (2017). *Stampede User Guide*.

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46.

Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6, New York, NY, USA. ACM.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., and Spillinger, O. (2013). Communication-optimal parallel recursive rectangular matrix multiplication. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 261–272, Washington, DC, USA. IEEE Computer Society.

Dongarra, J. J. (2002). Basic linear algebra subprograms technical (blast) forum standard (1). *IJHPCA*, 16:1–111.

Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition.

Ho, T. K. (1995). Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, pages 278–, Washington, DC, USA. IEEE Computer Society.

Huang, J., Smith, T. M., Henry, G. M., and van de Geijn, R. A. (2016). Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'16, pages 59:1–59:12, Piscataway, NJ, USA. IEEE Press.

Intel (2019a). *Intel C++ Compiler 19.0 Developer Guide and Reference.*

Intel (2019b). *Intel Fortran Compiler 19.0 Developer Guide and Reference.*

Keuper, J. and Preundt, F.-J. (2016). Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, pages 19–26, Piscataway, NJ, USA. IEEE Press.

Kiefer, J. and Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466.

Koza, J. R. (1990). Non-linear genetic algorithms for solving problems. United States Patent 4935877. filed may 20, 1988, issued june 19, 1990, 4,935,877. Australian patent 611,350 issued september 21, 1991. Canadian patent 1,311,561 issued december 15, 1992.

Kuhn, M. (2012). *caret: Classification and Regression Training.* R package version 5.15-044.

Kuhn, M. (2015). *C50: C5.0 Decision Trees and Rule-Based Models.* R package version 0.1.0-24.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323.

Le Gall, F. (2014). Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA. ACM.

Lindauer, M. T., Hutter, F., Hoos, H. H., and Schaub, T. (2015). Autofolio: An automatically configured algorithm selector. *J. Artif. Intell. Res.*, 53:745–778.

Platt, J. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rice, J. R. (1976). The algorithm selection problem. In Rubinoff, M. and Yovits, M. C., editors, *Advances in Computers*, volume 15 of *Advances in Computers*, pages 65 – 118. Elsevier.

RuleQuest (2018). Is see5/c5.0 better than c4.5? http://rulequest.com/see5-comparison.html. Accessed: 2018-03-06.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

Shi, S., Xu, P., and Chu, X. (2017). Supervised learning based algorithm selection for deep neural networks. *CoRR*, abs/1702.03192.

Spillinger, O., Eliahu, D., Fox, A., and Demmel, J. (2015). Matrix multiplication algorithm selection with support vector machines. Master's thesis, EECS Department, University of California, Berkeley.

Strassen, V. (1969). Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356.

van de Geijn, R. and Watts, J. (1995). Summa: Scalable universal matrix multiplication algorithm. Technical report, University of Texas at Austin, Austin, TX, USA.

van der Pas, R., Stotzer, E., and Terboven, C. (2017). *Using OpenMP - The Next Step*. The MIT Press.

# CHAPTER 3. CSR-DU ML: A MINIMAL DELTA UNITS LIBRARY USING MACHINE LEARNING

## 3.1 Sparse Matrix Formats

Sparse matrix-vector multiplication (SpMV) is a fundamental linear algebra operation used in many disciplines. Given a matrix $A \in \mathbb{R}^{m \times n}$ with many zero entries, and a dense vector $x \in \mathbb{R}^n$, SpMV computes $y = Ax \in \mathbb{R}^m$. The number of non-zero ($nnz$) entries of $A$ is used to classify how dense $A$ is, where density $d = \frac{nnz}{m*n} < .5$ is a classical cut-off for $A$ to be considered sparse. In contrast to dense matrices, sparse matrix operations are dominated by memory access as opposed to dense array floating-point operations (flops). Due to this memory bottleneck, many sparse matrix memory formats have been created to reduce memory bandwidth for SpMV. Instead of holding the entire $m \times n$ matrix, these matrix formats hold only the non-zero entries. Coordinate format (COO) is the most basic format, using three $nnz$ length arrays to explicitly hold the row, column, and value data where $A(row_i, column_i) = value_i$. The following is how a sparse matrix $A$ can be represented in COO format using zero-based indexing:

$$A = \begin{bmatrix} a & b & 0 & c \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & e \\ f & 0 & g & 0 \end{bmatrix} \underset{COO}{\Longrightarrow} \begin{aligned} row &= (0,0,0,1,2,3,3) \\ column &= (0,1,3,1,3,0,2) \\ values &= (a,b,c,d,e,f,g) \end{aligned} \tag{3.1}$$

In this case, the matrix-vector kernel is computed as

Algorithm 3.1 COO Matrix-Vector Multiplication

```
1 function COO_SpMV(row, column, value, x, y, nnz)
2 for(i=0; i<nnz; i++)
3     y[ row[i] ] += values[i] * x[ column[i] ];
```

Due to COOs simplicity, it is often used as a format to share matrices, such as from the Suite Sparse Matrix Collect (formally known as the University of Florida Sparse Matrix Collection) (Davis and Hu, 2011). However, the COO format has severe drawbacks in performance due to cache misses in arrays $x$ and $y$. While a matrix could be stored in COO in row-major or column-major order, there is no requirement to do so. Often, the matrix data for SpMV is streamed in a random order, making it to difficult access $x$ or $y$ efficiently without cache miss resolution.

### 3.1.1 Compressed Sparse Row

Compressed Sparse Row (CSR) is another matrix format using a row pointer to hold the start of a row in the column and value arrays. The use of a row pointer reduces memory of row array from $nnz$ to $m+1$ elements compared to COO. The CSR format can be summarized in the example below using zero-based indexing:

$$A = \begin{bmatrix} a & b & 0 & c \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & e \\ f & 0 & g & 0 \end{bmatrix} \underset{CSR}{\Longrightarrow} \begin{array}{rcl} rowPointer & = & (0,3,4,5,7) \\ columnIndex & = & (0,1,3,1,3,0,2) \\ values & = & (a,b,c,d,e,f,g) \end{array} \tag{3.2}$$

The SpMV kernel for CSR is given as follows:

```
1 function CSR_SpMV(rowPointer, columnIndex, value, x, y, m)
2 for(i=0; i<m; i++)
3   for(j=rowPointer[i]; j<rowPointer[i+1]; j++)
4     y[i] += values[j] * x[ columnIndex[j] ]
```

The last element $rowPointer[m] = nnz$ is a used as a ghost row pointer to allow access to $rowStart[i+1]$ with $i = m-1$. A major benefit of the CSR SpMV algorithm is that easy to parallelize in shared memory, where a single thread can operate on an entire row without synchronization. Due to CSR's inherent parallelism, it has been primary target for subsequent algorithms and optimization as seen in CSR5 (Liu and Vinter, 2015).

### 3.1.2 Blocked Compressed Sparse Row

Other successful formats exploit some structure within matrix $A$. The most common structures to be used are for blocked and banded matrices. For blocked matrices, Blocked Compressed Sparse Row (BCSR) is used to represent $r \times c$ blocks throughout $A$ (Im and Yelick, 2001). Like CSR, BCSR utilizes a block row pointer to store the start of a row. If $A$ is divided into row-order sequential nonzero blocks $A_i$ of size $r \times c$, then the block row pointer holds the index $i$ for the first nonzero submatrix $A_i$ in each blocked row. Similar to CSR, the last element of block row pointer holds the number of nonzero blocks in the matrix. In order to include possible leftover rows, the number of nonzero blocks is $\lceil m/r \rceil$. The blocked column array holds the starting column index for each nonzero submatrix $A_i$. The values array is sorted into the sequential blocks row-major ordering. For a simple implementation, any block starting index $(i, j)$ will satisfy $i \mod r = 0$ and $j \mod c = 0$. An example of BCSR is given in the following blocked matrix with $r \times c = 2 \times 2$ and zero padding for the values array inside each block:

$$A = \left[\begin{array}{cc:cc} a & b & 0 & c \\ 0 & d & 0 & 0 \\ \hdashline 0 & 0 & 0 & e \\ f & 0 & g & 0 \end{array}\right] = \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \underset{BCSR}{\Longrightarrow} \begin{array}{rcl} blockRowPointer & = & (0,2,4) \\ blockColumnIndex & = & (0,2,0,2) \\ values & = & (\mathbf{a}, b, 0, d, \mathbf{0}, c, 0, 0, \\ & & \mathbf{0}, 0, f, 0, \mathbf{0}, e, g, 0) \end{array} \tag{3.3}$$

The simplified BCSR SpMV kernel is given as the following

Algorithm 3.2 General $r \times c$ BCSR SpMV Kernel

```
1 function BCSR_SpMV(blockRowPointer, blockColumnIndex, values, x, y, m, r, c)
2 numBlockRows = ceiling(m/r);
3 for(i=0, row=0; i<numBlockRows; i++, row+=r)
4   for(j=blockRowPointer[i], valueOffset=0; j<blockRowPointer[i+1];
5       j++, valueOffset+=r*c)
6     for(iBlock=0; iBlock<r; iBlock++) // r×c block SpMV
7       for(jBlock=0; jBlock<c; jBlock++)
8       {
9         y_index = row + iBlock;
```

```
10          x_index = blockColIndex[j] + jBlock;
11          value_index = valueOffset + iBlock*c + jBlock;
12          y[y_index] += values[value_index] * x[x_index];
13        }
```

Given specific values for $r$ and $c$, the `iBlock` and `jBlock` loops can be unrolled into a single basic block using $r$ local accumulators for each row of $y$. Similar to the dense matrix multiplication counterparts, this loop unrolling can improve register/memory utilization and aid in vectorization for arrays $x$ and $y$. An example of a $2 \times 3$ BCSR SpMV kernel can be implemented as follows

Algorithm 3.3 Unrolled 2x3 BCSR SpMV Kernel

```
1  function BCSR2x3_SpMV(blockRowPointer, blockColumnIndex, values, x, y, m)
2  numBlockRows = ceiling(m/2);
3  for(i=0; i<numBlockRows; i++, y+=2)
4  { // y base address incremented
5    y0 = y[0]; y1 = y[1];
6    for(j=blockRowPointer[i]; j<blockRowPointer[i+1]; j++, values+=6)
7    { // values base address incremented
8      k = blockColumnIndex[j];
9      x0 = x[k]; x1 = x[k+1]; x2 = x[k+2];
10     y0 += values[0]*x0; y1 += values[3]*x0;
11     y0 += values[1]*x1; y1 += values[4]*x1;
12     y0 += values[2]*x2; y1 += values[5]*x2;
13   }
14   y[0] = y0; y[1] = y1;
15 }
```

In the unrolled BCSR implementation, the elements of $x$ are being reused. A drawback of BCSR is that picking good values of $r$ and $c$ are both matrix and machine dependent. Picking larger matrix blocks than necessary in $A$ will cause excessive zero padding in the values array, which wastes memory bandwidth and computation. CPU register and vector register sizes are also a consideration. Allowing the blocks $A_i$ to fit into cache or a single vector computation can improve throughput.

There also have been many works that investigate automatic tuning of these parameters (Vuduc, 2003) (Buttari et al., 2007). Auto-tuning is used because it is often too expensive computationally to do an optimal parameter search for$r$ and $c$ in a single instance of $A$.

### 3.1.3 Sparse Diagonal Format

For banded matrices, one of the leading storage format is called DIAG for storing matrix diagonals. The DIAG format assumes that the vast majority of of diagonals with nonzeros are dense. If this is not the case, then DIAG can have excessive zero padding to fill in empty diagonals. DIAG uses one array called the diagonal offset, which holds the band distance from the matrix diagonal, and a dense values where the columns of *values* hold the nonzeros associated with the matrix band. Let there be $d$ nonzero diagonals and $M = \min(m, n)$, then $values \in \mathbb{R}^{M\times d}$. If a diagonal offset $d < 0$, then the associated column of values is padded with $d$ zeros at the beginning, while $d > 0$ has $d$ zeros padded at the end of values. An example of the DIAG format can be summarized in the following where $0^*$ denotes a padded zero:

$$A = \begin{bmatrix} a & b & 0 & c \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & e \\ f & 0 & g & 0 \end{bmatrix} \underset{DIAG}{\Longrightarrow} \begin{array}{rcl} diagonal & = & (-3, -1, 0, 1, 3) \\ values & = & \begin{bmatrix} 0^* & 0^* & a & b & c \\ 0^* & 0 & d & 0 & 0^* \\ 0^* & 0 & 0 & e & 0^* \\ f & g & 0 & 0^* & 0^* \end{bmatrix} \end{array}. \tag{3.4}$$

For sparse matrices that are highly banded, DIAG is often one of the most compressed formats as it removes most data other than the values of $A$. However, for non-banded matrices, the values matrix can require more memory than the original matrix as seen in 3.4. For these reasons, the DIAG format is not a good candidate for general matrix storage. The DIAG SpMV kernel can be computed as

Algorithm 3.4 DIAG SpMV Kernel

```
1  function DIAG_SpMV(diagonal, values, x, y, numDiagonals, m, n)
2  M = min(m,n);
```

```
3 for(j=0; j<numDiagonals; j++)
4 {
5   d = diagonal[j];
6   rowStart = max(0,-d); rowEnd = M - max(0, d);
7   for(i=rowStart; i<rowEnd; i++)
8     y[i] += values[i][j] * x[d + i];
9 }
```

### 3.1.4 ELL Format

The last major format that is commonly used is called the ELL format from ELLPACK. In this format, only the column indices and values are stored in two 2D arrays in row order. If $R = \max_i RowNNZ_i$, then both the column and values arrays have dimensions $\mathbb{R}^{m \times R}$. While easier to convert to than CSR, ELL comes with a downside - each matrix row needs to approximately have the same number of nonzeros. In the case where matrix rows have irregular nonzero entries, additional zero padding in both the values and column arrays occur. However, a benefit of the zero padding is that SpMV computations can be done with cache-awareness and vectorization due to their regular array shapes. An example of the ELL format is given below.

$$A = \begin{bmatrix} a & b & 0 & c \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & e \\ f & 0 & g & 0 \end{bmatrix} \underset{ELL}{\Longrightarrow} columnIndex = \begin{bmatrix} 0 & 1 & 3 \\ 1 & 0^* & 0^* \\ 3 & 0^* & 0^* \\ 0 & 2 & 0^* \end{bmatrix}, \; values = \begin{bmatrix} a & b & c \\ d & 0^* & 0^* \\ e & 0^* & 0^* \\ f & g & 0^* \end{bmatrix} \tag{3.5}$$

Algorithm 3.5 ELL SpMV Kernel

```
1 function ELL_SpMV(columnIndex, values, x, y, m, R)
2 for(i=0; i<m; i++)
3   sum = 0.0
4   for(j=0; j<R; j++)
5     sum += values[i][j] * x[ columnIndex[i][j] ];
6   y[i] = sum;
```

To counteract the downside of the ELL format, Sliced ELL (SELL) was created where blocks of rows are converted to ELL individually (Kreutzer et al., 2014). SELL segregates variations of row nonzeros for the entire matrix into the individual blocks, where some blocks will have highly uniform row nonzeros, while other blocks do not. However, this improvement to ELL introduces the blocksize parameter $C$ (sometimes referred to as SELL-C), which is matrix and machine dependent.

### 3.1.5 Sparse Matrix Libraries

Similar to the dense BLAS standard, a sparse BLAS standard exists with three levels (Duff et al., 2002):

- Level 1: Sparse vector operations,
- Level 2: Sparse matrix, dense vector operations,
- Level 3: Sparse matrix-matrix operations.

Currently, there are many sparse libraries that implement the above formats for public use, such as the NIST library (public domain), Intel MKL, GNU Scientific Library (open source), cuSparse (NVIDIA), Eigen (open source), Librsb (open source), and SuiteSparse (open source). Many of these libraries also include sparse linear equation solvers and numerical algorithms using sparse matrix formats. Most libraries focus on allowing many different formats to exist within a single API, while others are optimized for a single format.

## 3.2 CSR Delta Units

CSR Delta Units (CSR-DU) can reduce memory requirements of CSR by exploiting that nonzeros in matrix rows are often close together (Kourtis et al., 2008). When the column array has been sorted in ascending order per matrix row, the "delta distance" of adjacent column indices ($delta = columnIndex[i] - columnIndex[i-1]$) is often small in comparison to the numerical ranges of 32-bit or 64-bit integers. When applicable, the compression of column index deltas into a 8-bit or 16-bit number yields a 2-4x memory savings over traditional 32-bit indices. With this in

mind, Kourtis et al. developed the CSR-DU format, which stores both the row pointer and column index arrays into a single structured called a *ctl*. The *ctl* object is composed of bit array data structures, which are numerical storage units, typically unsigned integers, used to encode data in bit form. Each bit array in the *ctl* has a header used for decoding, a column jump to increment the column index for new or continued matrix rows, and delta units to increment the column index. The header is comprised of a binary "new row" bit, the storage type for both the column jump and delta units, and the number of delta units stored. Kourtis et al. chose to use 8-bit, 16-bit, and 32-bit unsigned integers for the column jump and deltas. Going forward, we will denote unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers as U8, U16, U32, and U64 respectively. A list of the header data structure is given below.

- New Row (nr): Indicates if a delta unit is the start of the next matrix row.
- Unit Column Type (ucol): An integer code for a column offset for the first nonzero element in the delta unit.
- Unit Delta Type (uflag): An integer code for deltas stored in the delta unit.
- Unit Size (usize): The number of deltas stored in the delta unit.

An example of conversion process is given below:

$$A = \begin{bmatrix} a & b & 0 & c \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & e \\ f & 0 & g & 0 \end{bmatrix} \underset{CSR-DU}{\Longrightarrow} \quad ctl \quad = \tag{3.6}$$

| Header | | | | Payload | |
|---|---|---|---|---|---|
| nr | ucol | uflag | usize | col | deltas |
| 1 | U8 | U8 | 2 | 0 | $\{1,2\}$ |
| 1 | U8 | U8 | 0 | 1 | $\{\}$ |
| 1 | U8 | U8 | 0 | 3 | $\{\}$ |
| 1 | U8 | U8 | 1 | 0 | $\{2\}$ |

$$values \quad = \quad (a, b, c, d, e, f, g)$$

In cases where a delta unit has deltas that overflow the header bit array, bit arrays trailing the header are "pure" delta bit arrays without a header. Pure delta bit arrays keep the same delta type

that is encoded in the header. If $D_{header}$ is the number of deltas stored in header and $D_{bitarray}$ is the number of deltas that fit into a U64 bitarray, then the number of pure delta bitarrays required for a CSR-DU delta unit, denoted as $B_{pure}$, is given as

$$B_{pure} = \text{ceiling}\left[\left(usize - \min\left(usize, D_{header}\right)/D_{bitarray}\right].\tag{3.7}$$

The purpose of CSR-DU is to reduce memory bandwidth for the SpMV computation, since it is memory bound. However, decoding each bit array in the *ctl* adds computational time via control instructions for the new row, column jump, and delta units. Even with the additional computation, Kourtis et al. showed that the format gave a speedup of up to 10-20% over CSR for varying types of matrices. Furthermore, the conversion process from CSR into CSR-DU can be done online as it requires a single pass on the sparse matrix data. However, for delta units to be computed, the column indices must be sorted within each matrix row, which is not a requirement in traditional CSR. This column sort adds $\mathcal{O}\left(\sum_{i=1}^{m} nnz_i \ \ln nnz_i\right)$ operations to the conversion process, where $nnz_i$ is the number of non-zero elements in row $i$ of $A$.

### 3.2.1 Compressed Sparse eXtended

The next iteration of CSR-DU, called Compressed Sparse eXtended (CSX), added a variety of improvements over CSR-DU (Kourtis et al., 2011). The first improvement is run-length encoding (RLE) for the delta units, which can significantly reduce memory requirements for sequentially repeated deltas. For example, a delta of 3 repeated forty times in sequence can be expressed in RLE as (3,40). Furthermore, CSX attempts to look for additional delta units vertically, diagonally, anti-diagonally, and in block matrix form. To do this, Kourtis et al. used index transformations to convert the non-horizontal structures into horizontal form. These transformations allowed the use of a single horizontal structure detector to be used to probe for many delta unit patterns. The transformations can be shown in table 3.1 (Kourtis et al., 2011).

Table 3.1 CSX Horizontal Structure Transformations

| Matrix Structure | Index Transformation $(i', j')$ |
| --- | --- |
| Horizontal | $(i,\ j)$ |
| Vertical | $(j,\ i)$ |
| Diagonal | $(m+i-j,\ \min(i,j))$ |
| Anti-Diagonal | $\begin{cases} (m+j-i,\ j) & i < m \\ (j,\ i+j-m) & i \geq m \end{cases}$ |
| Block (row-aligned) | $\left(\left\lfloor \frac{i-1}{r} \right\rfloor,\ mod(i-1,r)+r(j-1)+1\right)$ |
| Block (column-aligned) | $\left(\left\lfloor \frac{j-1}{c} \right\rfloor+1,\ c(i-1)+mod(j-1,c)\right)$ |

In order to perform an optimal sparse conversion from CSR, CSX has uses a horizontal detector on a subsets of the full matrix data. From this, CSX uses a selection criteria based on the number of patterns found and the number of encoded non-zeros for each pattern. The conversion algorithm is given below:

1. For each matrix substructure to encode $t \in Pool_t$:
   - Convert $A$: $(i', j') = T_t(i, j)$ + sort row-wise
   - Label each structure as $(i', j', t)$
   - Label $(i, j, t) = T_t^{-1}(i', j')$ for start of substructure
   - Score substructure: $score_t = nnz_t - \text{DeltaUnits}_t$
2. Remove bad substructures from $Pool_t$ $(nnz_t < nnz/10)$
3. For max score substructure $t_{max}$:
   - Convert $A$ to $t_{max}$ again + sort
   - Encode each $(i, j, t_{max})$ in CSX and remove elements from $A$
   - Remove $t_{max}$ from $Pool_t$

4. Continue steps 1-3 until $score_{max} = 0$

5. Use Just-In-Time compiler to dynamically generate SpMV.

The entire CSX workflow can be summarized below (Meyer et al., 2013):

1. Load sparse matrix from file (serial)

2. Matrix structure detection (multithreaded)

3. Matrix structure encoding (multithreaded)

4. Just-In-Time code generation (serial)

5. SpMV kernel execution (multithreaded)

The matrix structure search for horizontal structure was similar to CSR-DU, but searching for block matrix structures adds an overhead equivalent to thousands of serial CSR SpMV computations. Due to the block matrix detection overhead, the authors suggest that the full conversion of CSR to CSX should be done offline, while the linear delta unit detection is suitable for online conversion.

An updated version of CSX has been developed called SparseX (Elafrou et al., 2018). It features a reworked C and C++ API, as well as improvements for multithreading with symmetric matrices. This library is open-source, however has several dependencies which may hinder its widespread use.

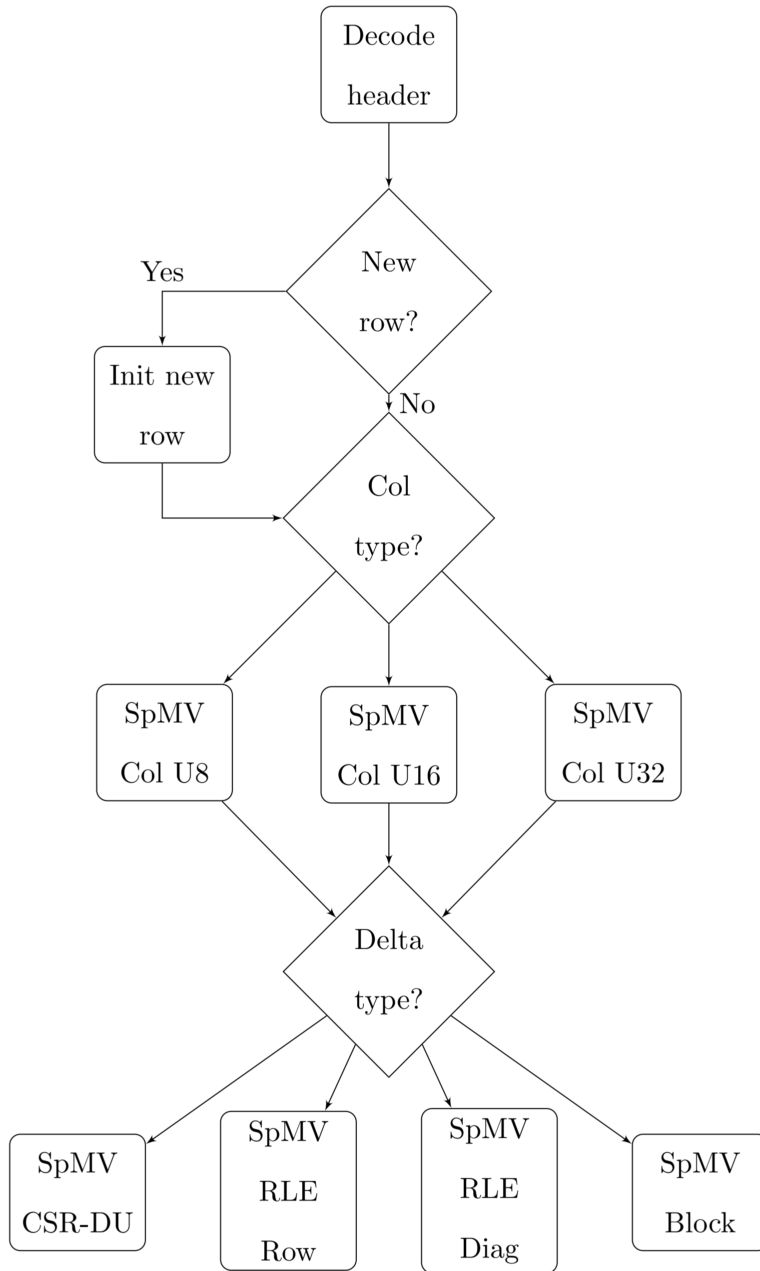


Figure 3.1 CSX SpMV Delta Unit Flow Chart

## 3.3 CSR-DU and CSX Implementation

### 3.3.1 Implementation

The implementation of CSX-ML is based on a bit array data structure using U64 as bit storage container. In the C language, the way to read and write a bit, byte, half-word, or word of data using the shift operators (`<<` and `>>`) in conjunction with a bit-wise and operator (`&`) or bit-wise or operator (|). A following code demonstrates how to write to bit array given an offset.

Algorithm 3.6 Bitarray Read and Write Operations

```
1 function readData(bitArray, offset)
2   return (bitArray >> offset) & readConstant
3
4 function writeData(bitArray, offset, data)
5   bitArray |= data << offset;
```

Table 3.2 shows the values for each unsigned integer type in the read and write bitarray operations, where non-bit data types are forced to be byte-aligned. In Table 3.3, we demonstrate the byte

Table 3.2 Bitarray Read and Write Implementation with Byte-aligned Offsets

| Size | Offset | Read Constant (hexadecimal) |
|---|---|---|
| bit | [0,63] | 0x1 |
| byte | [0,7]*8 | 0xFF |
| halfword | [0,6]*8 | 0xFFFF |
| word | [0,4]*8 | 0xFFFFFFFF |

layout of the delta unit header (U64 has 8 bytes shown in little endian) based on the type of column storage needed where H is a header byte, C is a column jump byte, and D is a delta. An expanded version of the table is given in Figure 3.2, where the exact number of bits/bytes are shown for the header bitarray.

Table 3.3 CSR-DU Header Byte Format.

| ucol | Header Byte Position | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| U8 | H | H | C | D | D | D | D | D |
| U16 | H | H | C | C | D | D | D | D |
| U32 | H | H | C | C | C | C | D | D |

| nr<br>1 bit | ucol<br>3 bits | uflag<br>4 bits | usize<br>1 byte | col<br>1-4 bytes | deltas<br>rest of bytes |
|---|---|---|---|---|---|

Figure 3.2 CSR-DU U64 Header Bitarray

Since it is impossible to predetermine the number of bitarrays needed to encode a matrix in CSR-DU or CSX, a re-sizable array of bitarrays is used as the base implementation of the ctl data structure. The resize capability is possible through the `realloc` C function, which copies the contents of a pointer and adds memory beyond the current bound into a new memory address. However, the `realloc` operation is expensive and should be avoided whenever possible during the file conversion process.

In order to encode CSR-DU or CSX from CSR, we used Algorithm 3.7 where it is assumed that CSR has already sorted each matrix row's columns in ascending order. If CSR is row-sorted, then the values array for CSR, CSR-DU, and the non-block formats of CSX are equivalent and no work needs to be done. Also, when converting from COO to CSR, the number of nonzeros in each row (`rowNNZ`) has to be computed for the `rowPointer` array, so this data can be made available when converting to from CSR to CSR-DU and CSX. A `state` data storage struct is used for the conversion process, as the format encoding is passed through several supporting functions which requires different variables. The supporting functions for this conversion were `deltaUnitEncoder()`, which is responsible for creating a new delta unit from the `deltas` array and state struct, while `resetData()` zeros the `deltas` array and most state variables. Furthermore, the column jump (col)

variable in serial CSR-DU can be encoded as a delta instead of a column index when a delta unit is continuing a matrix row.

Algorithm 3.7 CSR to CSR-DU Encoding Algorithm

```
1  function CSR_to_CSRDU(rowPointer, columnIndex, ctl, rowNNZ, state)
2  uint32_t deltas[255];
3
4  for(i=0; i<m; i++)
5    bitArray = 0;
6    state.isNewRow = 1;
7
8    if( rowNNZ[i] < 1 ) // If matrix row is empty
9    {
10     state.isEmptyRow = 1;
11     deltaUnitEncoder(ctl, state);
12   }
13   else if( rowNNZ[i] < 2 ) // Only single element in row
14   {
15     j = rowPointer[i]; state.column = columnIndex[j];
16     deltaUnitEncoder(ctl, state);
17   }
18   // At least 2 elements in row - can compute a delta
19   j = rowPointer[i]; state.column = columnIndex[j]; k = 0;
20
21   for(j=rowPointer[i]+1; j<rowPointer[i+1]; j++)
22   {
23     delta = columnIndex[j] - columnIndex[j-1];
24
25     // Check to see if deltas array overflowed in previous delta
26     if( deltasOverflowed ){ state.column = delta OR columnIndex[j]; continue; }
27
28     // If first delta in delta unit, set uflag
29     if(k == 0){ state.uflag = setUflag(bitArray, state); }
```

```
30
31      // Check to see if delta is too large to fit into current delta unit
32      mismatchU16 = (U8_MAX < delta < U16_MAX+1) && (state.uflag == U8);
33      mismatchU32 = (U16_MAX < delta) && (state.uflag == U8 || state.uflag == U16);
34      if( mismatchU16 || mismatchU32 )
35      {
36        state.usize = k;
37        deltaUnitEncoder(ctl, state);
38        resetData(deltas, state);
39        k = 0;
40        state.column = columnIndex[j]; // store mismatched delta as next column jump
41      }
42
43      // Now safe to add delta to delta unit
44      deltas[k++] = delta;
45      if(k == 255) // Check if deltas array full
46      {
47        state.usize = k;
48        deltaUnitEncoder(bitArray, state);
49        resetData(deltas, state
50      });
51        k = 0; deltasOverflowed = 1;
52    } // end for j
53
54    // Matrix row ended -> dump remaining deltas
55    // possible that last element in matrix row was a mismatch
56    if( k > 0 || state.column > 0)
57    {
58      state.usize = k;
59      deltaUnitEncoder(bitArray, state);
60      resetData(deltas, state);
61    }
62  } // end for i
```

Algorithm 3.8 CSR-DU SpMV Kernel

```
1  function CSRDU_SpMV(ctl, values, x, y, ctlSize)
2  for(ctlIndex=0; ctlIndex<ctlSize; ctlIndex++)
3  {
4    sum = 0.0; ucount = 0;
5    bitArray = ctl[ctlIndex];
6    usize = getUsize(bitArray);
7    if( isNewRow(bitArray) ) { rowIndex++; colIndex = 0; }
8
9    switch( getUcol(bitArray) )
10   {
11     case U8:  colIndex = getCol8(bitArray);  sum += *(values++)*x[colIndex]; break;
12     case U16: colIndex = getCol16(bitArray); sum += *(values++)*x[colIndex]; break;
13     case U32: colIndex = getCol32(bitArray); sum += *(values++)*x[colIndex]; break;
14     // ZeroCol is special case to enable matrix row w/ 1 element at colIndex=0
15     case ZeroCol: colIndex = 0; sum += *(values++) * x[0]; break;
16     case RowEmpty: continue;
17   }
18
19   switch( getUflag(bitArray) )
20   {
21     case U8:
22       // Process deltas in header bitarray
23       for(deltaByte = 0; ucount < usize && deltaByte < maxHeader8;
24           ucount++, deltaByte++)
25       {
26         colIndex += getDelta8(bitArray,deltaByte);
27         sum += *(values++)*x[colIndex];
28       }
29
30       // Process pure deltas bitarrays in unit
31       while(ucount < usize)
32       {
33         bitArray = ctl[ctlIndex++]; // Advance ctl
```

```
34          for(deltaByte = 0; ucount < usize && deltaByte < 8;
35              ucount++, deltaByte++)
36          {
37            colIndex += getDelta8(bitArray, deltaByte);
38            sum += *(values++) * x[colIndex];
39          }
40        }
41        break;
42
43      case U16:
44        // Process U16 deltas in header bitarray
45        for(deltaByte = 0; ucount < usize && deltaByte < maxHeader16;
46            ucount++, deltaByte+=2)
47        {
48          colIndex += getDelta16(bitArray,deltaByte);
49          sum += *(values++) * x[colIndex];
50        }
51
52        // Process U16 pure deltas bitarrays in unit
53        while(ucount < usize)
54        {
55          bitArray = ctl[ctlIndex++]; // Advance ctl
56          for(deltaByte = 0; count < usize && deltaByte < 8;
57              ucount++, deltaByte+=2)
58          {
59            colIndex += getDelta16(bitArray, deltaByte);
60            sum += *(values++) * x[colIndex];
61          }
62        }
63        break;
64
65      case U32:
66        // Process single U32 delta in header
```

```
67          if(ucount < usize && 0 < maxDeltasHeader32)
68          {
69            colIndex += getDelta32(bitArray,deltaByte);
70            sum += *(values++) * x[colIndex]; count++;
71          }
72
73          // Process U32 pure deltas bitarrays in unit
74          while(ucount < usize)
75          {
76            bitArray = ctl[ctlIndex++]; // Advance ctl
77            for(deltaByte = 0; count < usize && deltaByte < 8;
78                ucount++, deltaByte+=4)
79            {
80              colIndex += getDelta32(bitArray, deltaByte);
81              sum += *(values++) * x[colIndex];
82            }
83          }
84          break;
85
86    } // end switch uflag
87    y[rowIndex] += sum;
88  }
```

### 3.3.2 CSR-DU Parallel Implementation

For a shared-memory parallel version using OpenMP tasks, several small changes needed to be made. First, the following arrays are classified as shared to limit memory replication of large arrays: `ctl`, `columnIndex`, `values`, `x`, and `y`. Since the `values` array is shared, incrementing the base address of the array within a parallel task region is no longer feasible. To fix this, a values index is used to allow each task to access an independent section of the array. Also, writing to the shared array `y` without locks creates a race condition when multiple tasks are concurrently writing to the same vector row. To guard against a race condition, an OpenMP atomic update is used.

Next, column jumps between consecutive delta units in a matrix row can no longer hold deltas. Instead, column jumps hold the column index of the first nonzero for the delta unit to remove any dependencies from previous tasks in the same row. If there are dependencies across a matrix row, then tasks would have to be scheduled sequentially across the row. While the column index change will add memory to the encoding scheme for matrices with many columns, it allows for independent execution of each task.

In terms of spawning tasks and executing a parallel region, an OpenMP single region is used to start the SpMV computation for each delta unit. The main thread that enters the OpenMP single region is responsible for the new row increment, column jump SpMV computation, and fetching the unit size and unit delta type from every delta unit header bitarray. Afterward, a single OpenMP task is spawned per delta unit to finish the SpMV computation containing `usize` deltas. After each task is spawned, the main thread increment both the values and ctl indices to the start of the next delta unit.

Algorithm 3.9 CSR-DU OpenMP SpMV Kernel

```
1  function CSRDU_SpMV(ctl, values, x, y, numDeltaUnits)
2  rowIndex = −1; colIndex = 0; valIndex = 0; ctlIndex = 0;
3  #pragma omp parallel shared(ctl, values, x, y)
4  {
5    #pragma omp single nowait
6    {
7      for(deltaUnit=0; deltaUnit<numDeltaUnits; deltaUnit++)
8      {
9        sum = 0.0;
10       bitArray = ctl[ctlIndex];
11       usize = getUsize(bitArray);
12       uflag = getUflag(bitArray);
13       if( isNewRow(bitArray) ) { rowIndex++; }
14
15       switch( getUcol(bitArray) )
16       {
```

```
17          case U8:   colIndex = getCol8(bitArray);
18                     sum += values[valIndex++]*x[colIndex]; break;
19          case U16:  colIndex = getCol16(bitArray);
20                     sum += values[valIndex++]*x[colIndex]; break;
21          case U32:  colIndex = getCol32(bitArray);
22                     sum += values[valIndex++]*x[colIndex]; break;
23          // ZeroCol is special case to enable matrix row w/ 1 element at colIndex=0
24          case ZeroCol: colIndex = 0; sum += values[valIndex++] * x[0]; break;
25          case RowEmpty: continue;
26        }
27
28        if(usize == 0) // if delta unit is empty, don't spawn task
29        {
30          #pragma omp atomic update
31          y[rowIndex] += sum;
32          continue;
33        }
34
35        #pragma omp task firstprivate(rowIndex, colIndex, valIndex, ctlIndex, sum,
36                                      usize, uflag, bitArray) private(ucount, deltaByte)
37      ` {
38            ucount = 0;
39            switch( uflag )
40            {
41              case U8:
42                // Process deltas in header bitarray
43                for(deltaByte = 0; ucount < usize && deltaByte < maxHeader8;
44                    ucount++, deltaByte++)
45                {
46                  colIndex += getDelta8(bitArray,deltaByte);
47                  sum += values[valIndex++]*x[colIndex];
48                }
49
```

```
50                // Process pure deltas bitarrays in unit
51                while(ucount < usize)
52                {
53                  bitArray = ctl[ctlIndex++]; // Advance ctl
54                  for(deltaByte = 0; count < usize && deltaByte < 8;
55                      ucount++, deltaByte++)
56                  {
57                    colIndex += getDelta8(bitArray, deltaByte);
58                    sum += values[valIndex++]*x[colIndex];
59                  }
60                }
61                break;
62
63              case U16:
64                // Process U16 deltas in header bitarray
65                for(deltaByte = 0; ucount < usize && deltaByte < maxHeader16;
66                    ucount++, deltaByte+=2)
67                {
68                  colIndex += getDelta16(bitArray,deltaByte);
69                  sum += values[valIndex++] * x[colIndex];
70                }
71
72                // Process U16 pure deltas bitarrays in unit
73                while(count < usize)
74                {
75                  bitArray = ctl[ctlIndex++]; // Advance ctl
76                  for(deltaByte = 0; ucount < usize && deltaByte < 8;
77                      ucount++, deltaByte+=2)
78                  {
79                    colIndex += getDelta16(bitArray, deltaByte);
80                    sum += values[valIndex++] * x[colIndex];
81                  }
82                }
```

```
83                  break;
84
85                case U32:
86                  // Process single U32 delta in header
87                  if(ucount < usize && 0 < maxHeader32)
88                  {
89                    colIndex += getDelta32(bitArray,deltaByte);
90                    sum += values[valIndex++] * x[colIndex]; ucount++;
91                  }
92
93                  // Process U32 pure deltas bitarrays in unit
94                  while(ucount < usize)
95                  {
96                    bitArray = ctl[ctlIndex++]; // Advance ctl
97                    for(deltaByte = 0; count < usize && deltaByte < 8;
98                        ucount++, deltaByte+=4)
99                    {
100                     colIndex += getDelta32(bitArray, deltaByte);
101                     sum += values[valIndex++] * x[colIndex];
102                   }
103                 }
104                 break;
105
106           } // end switch uflag
107           #pragma omp atomic update
108           y[rowIndex] += sum;
109         } // end omp task region
110         valIndex += usize; // Move to start of next delta unit in main thread
111         switch(uflag)  // Move ctlIndex based on number of pure delta bitArrays
112         {
113           case U8:  ctlIndex += ceil((usize - MIN(usize, maxHeader8))/8); break;
114           case U16: ctlIndex += ceil((usize - MIN(usize, maxHeader16))/4); break;
115           case U32: ctlIndex += ceil((usize - MIN(usize, maxHeader32))/2); break;
```

```
116         }
117       } // end for delta unit
118     } // end omp single region
119   } // end omp parallel region
```

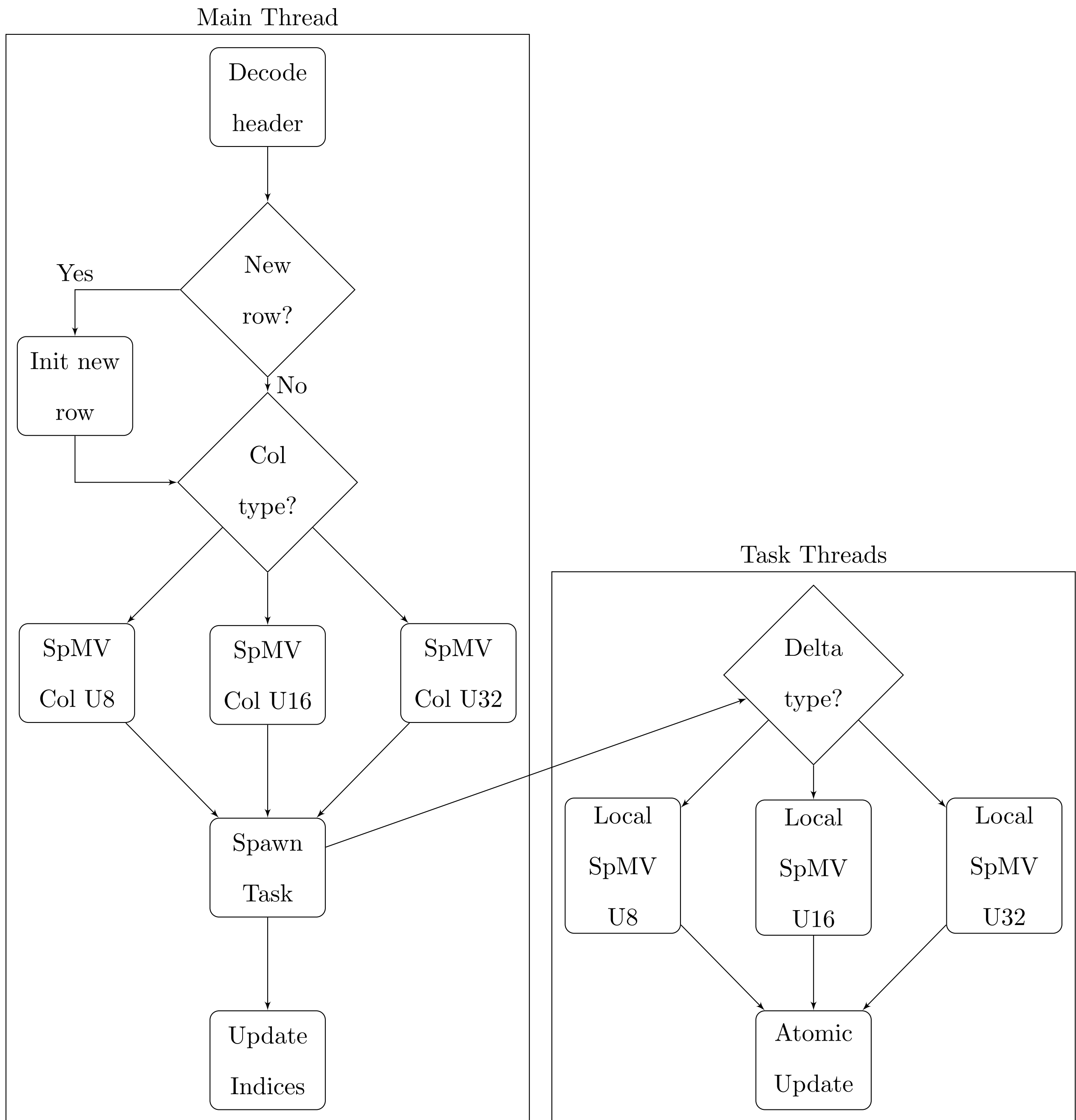


Figure 3.3 CSR-DU OpenMP Delta Unit SpMV Flow Chart

### 3.4 References

Buttari, A., Eijkhout, V., Langou, J., and Filippone, S. (2007). Performance optimization and modeling of blocked sparse kernels. *Int. J. High Perform. Comput. Appl.*, 21(4):467–484.

Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25.

Duff, I. S., Heroux, M. A., and Pozo, R. (2002). An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. https://www.nist.gov/publications/overview-sparse-basic-linear-algebra-subprograms-new-standard-blas-technical-forum.

Elafrou, A., Karakasis, V., Gkountouvas, T., Kourtis, K., Goumas, G., and Koziris, N. (2018). Sparsex: A library for high-performance sparse matrix-vector multiplication on multicore platforms. *ACM Trans. Math. Softw.*, 44(3):26:1–26:32.

Im, E.-J. and Yelick, K. (2001). Optimizing sparse matrix computations for register reuse in sparsity. In Alexandrov, V. N., Dongarra, J. J., Juliano, B. A., Renner, R. S., and Tan, C. J. K., editors, *Computational Science — ICCS 2001*, pages 127–136, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kourtis, K., Goumas, G., and Koziris, N. (2008). Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 87–96, New York, NY, USA. ACM.

Kourtis, K., Karakasis, V., Goumas, G., and Koziris, N. (2011). Csx: An extended compression format for spmv on shared memory systems. *SIGPLAN Not.*, 46(8):247–256.

Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. (2014). A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5):C401C423.

Liu, W. and Vinter, B. (2015). Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 339–350, New York, NY, USA. ACM.

Meyer, J. C., Cebrian, J. M., Natvig, L., Karakasis, V., Siakavaras, D., and Nikas, K. (2013). Energy-efficient sparse matrix autotuning with csx – a trade-off study. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 931–937.

Vuduc, R. W. (2003). *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley. AAI3121741.

# CHAPTER 4. CONCLUSION

## 4.1 Project Summaries

In the ML GEMM project, the authors have employed an algorithm selector developed via machine learning techniques to improve matrix-matrix multiplication. A multi-class classifier framework is provided to compute the General Matrix-matrix Multiplication (GEMM) in parallel on shared-memory computers. As a proof-of-concept, the authors selected six simple algorithms and compared them with Intel's MKL GEMM. These simple algorithms collectively outperformed MKL 12.8% of the time for matrix dimensions $m, n, k \in \{2^3, 2^4, \ldots, 2^{16}\}$. When non-MKL algorithms were selected, they exhibited an maximum speedup of 2.82-11.22x over MKL. Our algorithm selector, named ML_GEMM, was created with the C5.0 decision tree classification model to select the fastest algorithm for a given $m, n, k$. This algorithm selection was shown to be 93% accurate on the data set. ML_GEMM was shown to be superior to strictly using MKL 10.6% of the time, with a maximum speedup over MKL of 23.83x.

In the sparse encoding project, the authors demonstrated a new implementation of Compressed Sparse Row Delta Units library in serial and shared-memory via OpenMP. The CSR-DU format is a general sparse matrix encoding format based on CSR that assumes minimal structure of the a matrix $A$. If $A$ has a majority of rows with elements close together, then the distance between row elements can be encoded with a smaller number of bits compared to regular column indices in CSR. Furthermore, CSX is an extension of CSR-DU that enables additional matrix structures to be encoding via delta units. However, the current implementation of the CSX library comes with several shortcomings: software dependencies and a long format conversion process. The framework provided from CSR-DU and CSX is designed alleviate those restrictions.

## 4.2 Future Work

With the ML GEMM project, there are several avenues of potential improvement. The first being the use of more advanced optimization methods within the current loop-based algorithms such as Basic, Basic Transpose, Dot, and Dot Transpose. Additional low-level optimizations, including cache-aware memory management, can be implemented to further improve performance. To improve the algorithm pool, first the `Matmul` Fortran intrinsic function could be removed without much loss in perceived performance. Out of all the non-MKL algorithms, `Matmul` had the lowest representation within the data set. Next, recursive matrix multiplication algorithms can be explored, such as the CARMA algorithm. The original shared-memory CARMA algorithm was implemented with the Intel CILK Plus threading library, which has been deprecated in 2018. Within the matrix multiplication recursion, an end case is can be defined to be when a matrix dimension because small enough to use a loop-based multiplication algorithm. Another layer of algorithm selection can be used at to the recursion base case to select the fastest small matrix multiplication algorithm, where the matrix dimensions are restricted to a much smaller space.

As for the CSR-DU and CSX library, many improvements and features are planned. Continued optimizations for CSR-DU, such as decreasing memory requirements for the algorithm and exploring increasing the delta unit size to make tasking more efficient. Furthermore, a second shared-memory implementation that uses a ctl pointer (ie. an array storing the ctl index of the start of each delta unit) could be used in lieu of OpenMP tasking. A ctl pointer would reduce threading overhead managing each delta unit, but would add a significant increase in memory bandwidth to the SpMV algorithm. Any improvements to CSR-DU will carry over to CSX, as they share the same algorithm structure for the SpMV computation. Beyond the library implementation, the machine learning algorithm selection engine will have to be implemented on top of the of the CSX algorithm.