Performance Optimization of Memory-Bound Programs on Data Parallel Accelerators

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Naseraddin Sedaghati Mokhtari, M.Sc.
Graduate Program in Computer Science and Engineering

The Ohio State University
2016

Dissertation Committee:
P. Sadayappan, Advisor
Louis-Noël Pouchet
Radu Teodorescu
Atanas Rountev
ABSTRACT

High performance applications depend on high utilization of memory bandwidth and computing resources, and data parallel accelerators have proven to be very effective in providing both, when needed. However, memory bound programs push the limits of system bandwidth, causing under-utilization in computing resources and thus energy inefficient executions. The objective of this research is to investigate opportunities on data parallel accelerators (i.e., SIMD units and GPUs) and design solutions for improving the performance of three classes of memory-bound applications: stencil computation, sparse matrix-vector multiplication (SpVM) and graph analytics.

This research first focuses on performance bottlenecks of stencil computations on short-vector SIMD ISAs and presents StVEC, a hardware-based solution for extending the vector ISA and improving data movement and bandwidth utilization. StVEC includes an extension to the standard addressing mode of vector floating-point instructions in contemporary vector ISAs (e.g. SSE, AVX, VMX). A code generation approach is designed and implemented to help a vectorizing compiler generate code for processors with StVEC extensions. Using an optimistic as well as a pessimistic emulation of the proposed StVEC instructions, it is shown that the proposed solution can be effective on top of SSE and AVX capable processors. To analyze hardware
overhead, parts of the proposed design are synthesized using a 45nm CMOS library and shown to have minimal impact on processor cycle time.

As the second class of memory-bound programs, this research has focused on sparse matrix-vector multiplications (SpMV) on GPUs and shown that no sparse matrix representation is consistently superior, with the best representation being dependent on the matrix sparsity patterns. This part focuses on four standard sparse representations (i.e. CSR, ELL, COO and a hybrid ELL-COO) and studies the correlations between SpMV performance and the sparsity features. The research then uses machine learning techniques to automatically select the best sparse representation for a given matrix. Extensive characterization of pertinent sparsity features is performed on around 700 sparse matrices and their SpMV performance with different sparse representations. Applying learning on such a rich dataset leads to developing a decision model to automatically select the best representation for a given sparse matrix on a given target GPU. Experimental results on three GPUs demonstrate that the approach is very effective in selecting the best representation.

The last part is dedicated to characterizing performance of graph processing systems on GPUs. It focuses on a vertex-centric graph programming framework (Virtual Warp Centric, VWC), and characterizes performance bottlenecks when running different graph primitives. The analysis shows how sensitive the VWC parameter is to the input graph and signifies the importance of selecting the correct warp size in order to avoid performance penalties. The study also applies machine learning techniques on the input dataset in order to predict the best VWC configuration for a given graph. It shows the applicability of simple machine learning models to improve performance and reduce the auto-tuning time for graph algorithms on GPUs.
To my parents, Mohammad Ali and Rabeah, for their love and sacrifices.
ACKNOWLEDGMENTS

This space is an opportunity to extend my sincere appreciation and gratitude to all those who made this PhD thesis possible.

First, I want to thank my PhD advisor Prof. Sadayappan, who provided me with amazing opportunities and great mentorship and support. My special words of thanks should go to Prof. Pouchet for giving tremendous help and advice and being the best co-advisor one could ask for. This dissertation would not have been possible without the support of Prof. Teodorescu and Prof. Rountev, who helped me broaden my knowledge and understanding of the world of computer architecture and compilers.

Special thanks go to my good friends at HPCRL lab, especially Martin Kong, Venmugil Elango, John Eisenlohr, Kevin Stock, Tom Henretty, Justin Holewinski, Mahesh Ravishankar, and Changwan Hong. Also many thanks to the friends in Arch Lab, Renji Thomas, Xiang Pan, Anys Bacha, and Tim Miller. Special thanks are due to one of my best friends, Abolfazl Javan, who has always been a mentor and a true supporter, on and off the field of computer science.

Last but not least, none of these would have been achievable without the love and support of my family and my dearest friend Masi Kimiaie. They have been the true supporters and the unsung heros in this epic journey; their love is what makes me feel strong, and I cherish it above all else.
The experiments in this research would not have been possible without the generous donations of Kepler GPUs by NVIDIA Corporation. The material presented in this dissertation is based upon work supported by the U.S. National Science Foundation under grants CCF-1240651, CCF-1321147, CCF-1418256, ACI-1404995, and in part by funding for the Center for Domain-Specific Computing (CDSC) through the NSF Expedition in Computing Award CCF-0926127. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Naser Sedaghati
Columbus, Ohio
January 8, 2016
1999 — 2004 ............................... B.Sc.,
Ferdowsi University of Mashhad,
Mashhad, Iran

2004 — 2007 ............................... M.Sc.,
University of Tehran,
Tehran, Iran

2005 — 2007 ............................... Hardware Engineer,
SiNA Microelectronics Inc.
Tehran, Iran

2007 — 2008 ............................... Systems Engineer,
UT e-Learning Center,
Tehran, Iran

2009 — 2010 ............................... Ph.D. Student,
North Carolina State University,
Raleigh, NC

2009 — 2010 ............................... Teaching Assistant,
North Carolina State University,
Raleigh, NC

2010 — 2015 ............................... Research Associate,
The Ohio State University,
Columbus, OH

Summer 2013 ............................... Intern,
NVIDIA,
Santa Clara, CA

Summer 2014 ............................... Intern,
Samsung Research America,
San Jose, CA

2015 ............................... Ph.D.,
The Ohio State University,
Columbus, OH
PUBLICATIONS

Research Publications

N. Sedaghati, L.N. Pouchet, and P. Sadayappan, Performance Characterization of Graph Programming Models on GPUs, Technical Report, Ohio State University, OSU-CISRC-10/15-TR19, October 2015

N. Sedaghati, T. Mu, L.N. Pouchet, S. Parthasarathy, and P. Sadayappan, Automatic Selection of Sparse Matrix Representation on GPUs, International Conference on Supercomputing (ICS), June 2015

N. Sedaghati, A. Ashari, L.N. Pouchet, S. Parthasarathy, and P. Sadayappan, Characterizing Dataset Dependence for Sparse Matrix-Vector Multiplication on GPUs, Workshop on Parallel Programming for Analytics Applications (held with PPoPP), February 2015


A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, An Efficient Two-Dimensional Blocking Mechanism for Sparse Matrix-Vector Multiplication on GPUs, International Conference on Supercomputing (ICS), June 2014


N. Sedaghati, R. Thomas, L.N. Pouchet, R. Teodorescu, and P. Sadayappan, **StVEC: a vector instruction extension for high-performance stencil computation**, *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2011

**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xvi</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xix</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xx</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Background</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Memory-bound Programs</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Stencil Computation</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 Sparse Matrix-Vector Multiplication</td>
<td>9</td>
</tr>
<tr>
<td>2.1.3 Graph Analytics</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Data-Parallel Architectures</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Short-Vector SIMD</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2 Graphics Processor Unit (GPU)</td>
<td>12</td>
</tr>
</tbody>
</table>
3. Optimizing Stencil Computation on Short-Vector SIMD ISAs 14
   3.1 StVEC: A Vector ISA Extension 17
      3.1.1 StVEC Functionality: An Example 17
      3.1.2 StVEC Execution Model 20
      3.1.3 StVEC Instruction Format 23
      3.1.4 Decoding StVEC Instructions 24
      3.1.5 Modified Vector Register File 25
      3.1.6 Generalizing StVEC 27
   3.2 Code Generation for StVEC 27
      3.2.1 Program Representation 28
      3.2.2 Auto-vectorization Using Intrinsics 29
      3.2.3 Integration of StVEC extension 33
      3.2.4 Avoiding All Unaligned Loads 37
   3.3 Evaluation Methodology 39
      3.3.1 Baseline Implementation 39
      3.3.2 StVEC Implementations 40
      3.3.3 Experimental Setup 41
      3.3.4 Stencil Benchmarks 42
   3.4 Experimental Results 43
      3.4.1 Performance Evaluation 43
      3.4.2 Hardware Overhead 47
   3.5 Related Work 48
   3.6 Conclusion 49

4. Sparse Matrix-Vector Multiplication on GPUs 50
   4.1 SpMV Representations on GPUs 51
   4.2 Sparse Matrices and Features 54
      4.2.1 Feature Analysis 55
   4.3 SpMV Performance Characterization 58
      4.3.1 Kernel Performance for CSR 60
      4.3.2 Kernel Performance for ELL 60
      4.3.3 Kernel Performance for COO 61
      4.3.4 Kernel Performance for HYB 62
      4.3.5 Conclusions 63
   4.4 The Sparse Triangle 63
      4.4.1 Selecting the Best Representation 63
      4.4.2 The Role of the Matrix Density 64
      4.4.3 Summary and Discussion 66
   4.5 Related Work 67
4.6 Conclusion .................................................. 68

5. Automatic Selection of Sparse Representation on GPUs ............... 69
   5.1 Sparse Matrices and Performance Variability ..................... 71
      5.1.1 GPUs and Compilation .................................. 73
   5.2 Feature Analysis ........................................... 73
      5.2.1 Feature Set ............................................. 74
      5.2.2 Feature Distribution .................................... 75
   5.3 Performance Analysis ......................................... 78
   5.4 Predicting The Best Representation .............................. 84
      5.4.1 Problem Learned ......................................... 84
      5.4.2 Training and Testing .................................... 86
   5.5 Selecting The Best Representation ................................ 87
      5.5.1 Experimental Protocol ................................... 87
      5.5.2 Evaluation of Specialized Models ......................... 88
      5.5.3 Discussion on The Validation Approach ..................... 92
   5.6 Related Work ................................................ 97
   5.7 Conclusion and Discussion .................................... 99

6. Graph Analytics on GPUs ........................................ 100
   6.1 Methodology ................................................ 102
      6.1.1 Graph Features ......................................... 102
      6.1.2 Graph Primitives and GPUs .............................. 103
      6.1.3 Graph Programming Models Overview .................... 104
   6.2 Feature Analysis ............................................ 105
      6.2.1 Feature Distribution .................................... 105
      6.2.2 Correlation Analysis .................................... 109
   6.3 Performance Analysis ......................................... 113
      6.3.1 Algorithm Performance .................................. 113
      6.3.2 Analysis of VWC Performance ........................... 115
   6.4 Automatic Selection of Warp Size ................................ 117
      6.4.1 Learning Problem and Quality Metric .................... 118
      6.4.2 Experimental Methodology ............................... 120
      6.4.3 Classification ........................................... 122
      6.4.4 Clustering ................................................. 125
   6.5 Discussion ................................................... 128
   6.6 Related Work ................................................ 129
   6.7 Conclusion ................................................... 130
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 A computing system viewed as a simple von Neumann architecture.</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Range of operational intensity [84].</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Illustration of VRF use and assembly code for code from Listing 3.1 (S3): (a) Using extra vector load and alignment instructions, (b) Using unaligned vector loads and (c) using StVEC.</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Operand encoding: (left) Two source vector registers, ( VR_x ) (( X_x ) elements) and ( VR_y ) (( Y_y ) elements), (right) Different permutations for second vector operand, ( VR_z ), based on values of the offset.</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Vector register file (VRA) augmented with adjustment logic.</td>
<td>26</td>
</tr>
<tr>
<td>3.4 StVEC code generation example: original loop (a), intrinsics translation (b), intrinsics plus software-pipeline (c) final StVEC code (d).</td>
<td>29</td>
</tr>
<tr>
<td>3.5 Methodology to emulate and evaluate StVEC performance on real machines</td>
<td>39</td>
</tr>
<tr>
<td>3.6 Average (geometric means) of relative speedup with StVEC for single precision across machines and compilers.</td>
<td>43</td>
</tr>
<tr>
<td>3.7 Average (geometric means) of relative speedup with StVEC for double precision across machines and compilers.</td>
<td>44</td>
</tr>
<tr>
<td>3.8 Summary of single-precision improvement by StVEC across machines.</td>
<td>46</td>
</tr>
<tr>
<td>3.9 Summary of double-precision improvement by StVEC across machines.</td>
<td>46</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 GFLOP/s for S1, S2 and S3 on different machines for N=1024 and T=500000</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Vector arithmetic instructions: Standard vs. StVEC (W: vector width)</td>
<td>23</td>
</tr>
<tr>
<td>3.3 StVEC new vector intrinsics</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Hardware platforms used for emulating StVEC instructions</td>
<td>41</td>
</tr>
<tr>
<td>3.5 Optimization options for ICC/GCC on different machines</td>
<td>42</td>
</tr>
<tr>
<td>3.6 Absolute performance numbers for baseline code (sp-intrin): ICC (top) and GCC (bottom)</td>
<td>45</td>
</tr>
<tr>
<td>3.7 Access time for baseline and StVEC vector register files (BVRF and StVRF) in 45nm CMOS technology</td>
<td>47</td>
</tr>
<tr>
<td>4.1 Space for every representation in single-precision (4-byte). Matrix A is $M \times N$ with total $nnz$ nonzeros. $k$ is the cut-off point for ELL/COO partitioning in HYB where $X$ nonzeros left for COO</td>
<td>52</td>
</tr>
<tr>
<td>4.2 Features of an $M \times N$ sparse matrix used in this study</td>
<td>54</td>
</tr>
<tr>
<td>4.3 Set of matrices used in this study ($M$: number of rows, $N$: number of columns, $nnz$: non-zero, $\mu$: mean, $\sigma$: standard deviation)</td>
<td>56</td>
</tr>
<tr>
<td>4.4 GPU hosted the experiments</td>
<td>58</td>
</tr>
<tr>
<td>4.5 Three outliers to %nnz model</td>
<td>65</td>
</tr>
</tbody>
</table>
5.1 Performance difference for sparse matrices from UFL matrix repository [18] (single-precision GFLOP/s on Tesla K40c GPU) .................................. 72

5.2 GPUs hosted the experiments. .................................................. 72

5.3 Features of an $M \times N$ sparse matrix used in this study. A "non-zero block" refers to a set of non-zeros in adjacent rows and columns (i.e. forming small rectangular dense blocks). .................................. 75

5.4 Statistics on main features. .................................................... 76

5.5 Fastest representation statistics .............................................. 79

5.6 Average slowdown over the 682 matrices when a fixed representation is used instead of the individual best ............................................. 80

5.7 Number of > 2x slowdown cases .............................................. 80

5.8 K40c using HYB ................................................................. 81

5.9 Feature sets evaluated .......................................................... 87

5.10 Best performing model found for K40c (i.e. training one model per library) ................................................................. 88

5.11 Best performing model found for K20c (i.e. training one model per library) ................................................................. 89

5.12 Best performing model found for Fermi (i.e. training one model per library) ................................................................. 89

5.13 BFTree minNum=2, numFold=5, Simple features ...................... 91

5.14 Performance of three-phase validation (cuSPARSE library on K40c GPU). $SD$ and $PID$ denote "Slowdown" and "Partition ID", respectively. ............................................. 95

6.1 Graph features and the cost of computing each on $G = (V, E)$. ........ 103

6.2 Group of studied graphs (Coefficient of Variation: $CV = \frac{\sigma}{\mu}$). ........ 107
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Graph feature subsets after removing the correlated features.</td>
<td>112</td>
</tr>
<tr>
<td>6.4</td>
<td>How often best warp size is the same for all algorithms.</td>
<td>114</td>
</tr>
<tr>
<td>6.5</td>
<td>Average (geometric mean) slowdown when a fixed warp size is chosen.</td>
<td>115</td>
</tr>
<tr>
<td>6.6</td>
<td>Average (geometric mean) slowdown across group of studied graphs.</td>
<td>116</td>
</tr>
<tr>
<td>6.7</td>
<td>Histogram of average (geometric mean) slowdown with a fixed-sized warp.</td>
<td>116</td>
</tr>
<tr>
<td>6.8</td>
<td>2D map of relative slowdown and performance differences.</td>
<td>118</td>
</tr>
<tr>
<td>6.9</td>
<td>2D map of distribution (%) of slowdown below 10% of fastest configuration.</td>
<td>119</td>
</tr>
<tr>
<td>6.10</td>
<td>Performance and quality of trained decision trees.</td>
<td>123</td>
</tr>
<tr>
<td>6.11</td>
<td>Performance and quality of KMeans clustering.</td>
<td>127</td>
</tr>
</tbody>
</table>
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>94</td>
</tr>
</tbody>
</table>

- 3.1 Generating basic SIMDized version of an innermost loop.
- 3.2 Generating software-pipelined version of each SIMDized loop.
- 3.3 Generating StVEC intrinsics.
- 5.1 Three-phase validation methodology.
## LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Example 2D stencil loop (Jacobi)</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Vector multiply-add loops with different multiply operands: aligned-constant (S1), unaligned-constant (S2), unaligned-aligned (S3)</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Retiming statement in a loop: before</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Retiming statement in a loop: after</td>
<td>38</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

In the high-performance computing world, programs are divided into categories, depending on the computation patterns. One metric to help identify a program is the amount of computation done per unit of data transferred, called Arithmetic Intensity. At the very extreme of the intensity spectrum, there exist memory-bound applications, whose execution time is limited by the memory bandwidth. Since transferring data from slower memory to the faster processing unit is fixed for a given architecture, the memory-bound programs tend to under-utilize compute resources due to having low arithmetic intensity. The objective of this research is to find opportunities for optimizing such programs on data parallel architectures (i.e. SIMD vector units and GPUs).

This work focuses on three different computation patterns (stencil computation, sparse matrix-vector multiplication and graph analytics) and attempts to address challenges and opportunities to optimize and characterize performance on data parallel architectures.

The first is stencil computation, an important class of scientific computations used in different domains (e.g. image processing, solution of PDEs, etc). Accelerating stencil computation on modern short-vector Single-Instruction Multiple-Data (SIMD)
architectures is tightly dependent on vectorizing inner-most loops where arrays are accessed in unit stride. When vectorizing the stencil loops, and due to the abstraction of stream alignment conflict \[33\], the vector operands share scalar words in multiple "conflicting" streams (i.e. one stream is a shifted version of another). Such an overlap leads to inefficient memory access for the vectorized stencil code, as loading vector registers cause redundant or unaligned memory accesses. Former is where a data element is moved with a different load for each distinct vector register position, while latter is possible when unaligned vector memory access is available by the architecture. Nevertheless, both cases lead to executing "overhead" instructions and reducing the execution rate for "actual" compute operations.

First part of this research presents StVEC, a hardware-based solution that extends SIMD ISA to optimize data movement and improve the performance by reducing (eliminating) the stream alignment effects. The key idea is to enhance the addressing modes for vector operands in vector arithmetic instructions, by allowing elements from a pair of registers to form one of the operands. This research proposes an architectural design along with a compiler algorithm for generating intrinsics-based code for the extended architecture. The effectiveness of StVEC is examined by emulating the new modified vector instructions on real processors. Using a number of stencil benchmarks, StVEC instructions are experimentally evaluated by an optimistic and pessimistic emulation on four modern multi-core CPU platforms. For instance, running stencil kernels using StVEC instructions could achieve up to 2.47x (optimistic) or 2.26x (pessimistic) performance speedup over an auto-vectorized code. StVEC \[66\] is the pioneer in designing hardware-based solution along with compiler support to address the performance limitations of current short-vector SIMD architectures for stencil
computations. The architecture design, code generation algorithms, and experiments results are described in Chapter 3.

The second class of memory-bound programs, studied in this research, is sparse matrix-vector multiplication (SpMV). Efficient execution of SpMV on modern data parallel accelerators (e.g. GPUs) is challenging, due to sparsity of memory accesses and low compute-per-byte ratio. Such properties make GPU-specific optimizations for high-performance SpMV very challenging. The most interesting solution is to design GPU-friendly sparse representation (i.e. storage format) for the input matrix in order to improve resource utilization, reduce load imbalance and thread/memory divergence (the three most important challenges of GPU programming). However, for a given sparse matrix, the choice of ”the best” sparse representation is very closely dependent on the sparsity of the matrix. As elaborated more in this study, no superior sparse representation exists that performs equally best among all sparse matrices.

The second part of this research focuses on four standard representations (from NVIDIA cuSPARSE library [14]): CSR [7], ELLPACK (ELL) [60], COO, and a hybrid ELL-COO (HYB) [8]. Using a modern NVIDIA K40c GPU, and for a set of 27 sparse matrices (covering a spectrum of application domains and sparsity features), Chapter 4 evaluates the SpMV kernel performance of each of the four representations to determine which representation performs best. By analyzing and correlating sparsity features to the best performing representation, the study makes insights on how to determine a priori the sparse representation delivering the highest SpMV kernel GFLOP/s. As a result, an empirical rule is developed to select a priori the best representation on the studied dataset, based on the density and standard deviation of the number of non-zero elements per row.
The next part of this research (Chapter 5) addresses the problem of automatically selecting, a priori, the best performing representation using only features of the input matrix such as the average number of non-zero entries per row (the average degree of graphs). This is achieved by developing a machine learning approach using decision tree classifiers. As a result of extensive characterization of the performance of the automatically generated models, it becomes feasible to achieve on average a slowdown of no more than 1.05x (compared to an ideal oracle approach systematically selecting the best representation for every matrix). The study focuses on standard representations (NVIDIA cuSPARSE and CUSP) on three high-end NVIDIA GPUs and develops a fully automated approach to select the best performing sparse representation, using only simple features of the input sparse matrices. The results suggest portable and simple-to-translate decision trees of 30 leaf nodes achieving 95% of the maximal possible performance.

The third class of memory-bound programs studied in this research is graph analytics. Executing a graph primitive efficiently on GPUs raise challenges similar to SpMV. Current graph programming models for GPUs mostly follow vertex-centric paradigm where a vertex is considered a unit of work, and is assigned to a GPU worker (e.g. thread, warp, or thread-block). However, performance of a graph analytics algorithm is very sensitive to the graph features and the graph system implementation. Selecting a configuration (of the system) that leads to best performance for a given algorithm is what Chapter 6 attempts to address.

As the last part, this research focuses on graph processing and optimization opportunities on the GPUs. As a case study for graph programming model, this part focuses on Virtual Warp Centric (VWC), and characterizes performance bottlenecks
when running different graph primitives. The analysis on this chapter showed the sensitivity of VWC (and the virtual warp size parameter) to the input graph and the importance of selecting the correct warp size in order to avoid huge performance penalties. This chapter provides machine learning techniques in order to predict the best warp size for a given graph. The quality of prediction is evaluated based on the performance different with respect to the best warp size, and also based on the distribution of slowdown. The main objective of this study is to address the applicability of simple machine learning models to reduce the auto-tuning time of graph algorithms on GPUs was the main objective for this chapter.

The rest of the dissertation is organized as follows. Chapter 2 provides background information on the programs, architectures, and performance optimization challenges. Chapter 3 presents the key idea of hardware-based solution for stencil computation, as well the the support for code generation. Optimizing SpMV program on GPUs, studying the bottlenecks and characterizing the correlations are presented in Chapter 4. The automatic selection of best representation, using the machine learning techniques is presented in Chapter 5. The analysis and modeling of graph primitives on GPUs are presented in Chapter 6. The future research directions are discussed in Chapter 7, and the thesis concludes in Chapter 8.
CHAPTER 2

Background

This chapter provides more insights about the memory-bound programs, data-parallel architectures, and the corresponding performance optimization challenges.

2.1 Memory-bound Programs

Memory boundedness refers to a situation where the time to complete a computation is decided primarily by the amount of data transferred from memory (i.e. memory speed). One way to visualize performance of a program and determine the boundedness (compute or memory) is the Roofline model [84]. Figure 2.1 shows a computing system viewed as a simple von Neumann architecture where the processing unit requires $Q$ memory transfers in order to complete $W$ arithmetic operations.

![Figure 2.1: A computing system viewed as a simple von Neumann architecture.](image-url)
In the Roofline model, $W$ is the total number of "useful" operations (e.g. in flops) that an algorithm performs and $Q$ is the total number of words it transfers (reads or writes). Then, the \textit{operational intensity} is measured by $I \equiv W/Q$. Intensity (in \textit{flops/byte} or \textit{flops/word}) defines scalability of an algorithm such that a higher value of $I$ implies that the algorithm has more work than memory operations (i.e. more likely to improve when compute throughput, e.g. number of processing units increases). Therefore, for a given algorithm running on a machine, the maximal attainable performance (i.e. $P$) is "bounded" by either the peak compute performance of the architecture (i.e. $\pi$ in flops/cycle) or the peak off-chip memory bandwidth (i.e. $\beta$ in byte/cycle), as shown by Equation 2.1:

$$P \leq \min(\pi, I\beta) \quad (2.1)$$

The former group (where $I \leq \pi/\beta$) defines "memory-bound" computations while the latter (where $I \geq \pi/\beta$) indicates "compute-bound" algorithms. The intersection of the two bounds (i.e. intensity of $I = \pi/\beta$) shows the \textit{balanced} algorithms where the time for computation and the time for memory traffic are the same.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{OperationalIntensity.png}
\caption{Range of operational intensity [84].}
\end{figure}
Depending on the core computation pattern, different applications could be mapped to different places of operational intensity spectrum, as shown by Figure 2.2. Among the memory-bound programs with lowest operational intensity (i.e. leftmost side of the spectrum), three classes of computations are of the most interest in this research: stencil computation, sparse matrix-vector multiplication (SpMV) and graph analytics.

2.1.1 Stencil Computation

Stencil computations typically appear in partial differential equations (PDE) solvers or in image processing. Considerable attention has been paid recently to optimizing stencil computations [3, 15, 19, 40, 44, 70, 78, 81, 33, 66, 73]

```
for (t = 0; t < T; t++)
    for (i = 1; i < N - 1; i++)
        for (j = 1; j < N - 1; j++)
```

**Listing 2.1:** Example 2D stencil loop (Jacobi).

Listing 2.1 shows an example of stencil computation (on 2D grids) where one point in the destination grid (B) is produced by combining (sum of product) of five neighboring points from the source grid (A). The two-dimensional spatial loops (i and j) are surrounded by an outer time loop (t). Each time iteration finishes one sweep over the grids and test for convergence (e.g. difference between current and
previous produced point values). The time loop exits when the convergence test is passed.

Stencil computation, when run on modern processors with memory hierarchy, does not achieve peak performance, mainly due to saturating the memory bandwidth. The problem becomes even worse for larger input sizes, as large stencil computations cause more cache misses (compulsory); increasing the processor idle time. Tiling in both space and time loops helps reduce the main memory traffic and increase the operand re-use; enabling higher performance.

2.1.2 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector (SpMV) multiplication is a Level-2 BLAS operations between sparse matrices and dense vectors, described by Equation 2.2.

\[ y = \alpha \times \text{op}(A) \times x + \beta \times y \]  

(2.2)

In our study, \( \alpha = \beta = 1 \), \( A \) is a two-dimensional sparse matrix, \( x \) and \( y \) are one-dimensional dense vectors, and \( \text{op]() \) is identity operation. Thus, the SpMV equation is simplified as \( y = A \times x + y \). Looking at the element-wise operation, where \( a_{i,j} \in A \), we have:

\[ \forall a_{i,j} \neq 0 : y_i = a_{i,j} \times x_j + y_i \]  

(2.3)

The SpMV algorithm by nature has low arithmetic intensity (i.e. \( AI \) which is defined as total flops per total DRAM bytes moved). For an \( n \times n \) sparse matrix with total of \( nnz \) non-zero values, the algorithm performs \( 2 \times nnz \) flops for which a total
of \( nnz + 2 \times n \) words are moved. When \( nnz >> n \) the \( AI \) moves towards \( \frac{1}{2} \) as shown by Equation 2.4.

\[
AI = \frac{\text{flops}}{\text{words}} = \frac{2}{4} \times \frac{nnz}{nnz + 2 \times n} \leq \frac{1}{2} \tag{2.4}
\]

Sparsity in the matrix \( A \) leads to irregularity (and thus lack of locality) when accessing \( a_{i,j} \) elements. Thus, even with optimal reuse for \( x \) and \( y \) elements, it is the accesses to matrix \( A \) (and thus the low arithmetic intensity) of SpMV that will significantly impact final execution time of a kernel.

In order to make efficient SpMV storage and processing possible, and unlike dense matrices, a sparse matrix is stored in a specific data structure (i.e. representation). Such representations (e.g. ACSR [1], BRC [2], CSR [14]) are designed to improve the memory access pattern of the sparse matrix and reduce processor’s idle cycles.

### 2.1.3 Graph Analytics

Numerous real-world problems, from social networks to language modeling, can be encoded as graphs. A graph is represented as \( G = (V, E) \) and can be seen as an abstract data structure that models structural relationships among objects. The definition of objects and relationships vary across different problem domain. Graph primitives normally are iterative algorithms working on (a subset of) vertices and edges, in order to compute a property of the graph.

A graph can be stored in memory as an adjacency matrix \( M \), where a non-zero value \( M[i][j] \) indicates an edge from \( v_i \) to \( v_j \). Sparse storage formats are also used to save time and memory space for the graphs. Depending on implementation, a graph may hold state information on an edge or a vertex, or both. Accessing memory
for vertex or edge could lead to "random accesses"; limiting the memory bandwidth utilization. Graph primitives are memory-bound as they tend to have a large number of memory operations (e.g. each vertex requires state information of all the incoming edges) and a small number of arithmetic operations (e.g. only increment by 1 when doing BFS). Such low computational intensity demands for certain optimizations on different architectures.

2.2 Data-Parallel Architectures

Data-parallel architectures are designed to handle executing same operations on multiple data elements; thus data-level parallelism (DLP). Such architectures are energy-efficient because they reduce the cost of executing an instruction by performing the "overhead" operations (e.g. fetch, decode) only once for multiple instructions. Unlike standard CPUs, the main design objective for data-parallel architectures/accelerators is throughput (i.e. operations per unit of time), as opposed to latency (i.e. cycles per operation).

From tightly-coupled SIMD extensions (e.g. SSE or AVX) on modern CPUs, to the more complex Graphics Processing Units (GPUs), the data parallel architectures can implement DLP (i.e. SIMD) as well as Thread-Level Parallelism (TLP); the combination of DLP and TLP is called Single Instruction Multiple Threads (SIMT).

2.2.1 Short-Vector SIMD

Short vector SIMD extensions are now included in all major high-performance CPU. While ubiquitous, the ISA and performance characteristics across vendors and hardware generations (e.g. Intel’s SSE2 vs AVX). With every processor, the latency and throughput numbers of instructions in these extensions change. These extensions
provide from 2-way adds and multiplies up to 16-way fused multiply-add operations, promising significant speed-up (over scalar execution).

2.2.2 Graphics Processor Unit (GPU)

A GPU is a high-throughput multi-core processor where each core is designed as a streaming multiprocessor (SM, in NVIDIA’s terminology) with variable number of functional units (e.g. 32 CUDA cores in NVIDIA Fermi GF100 [28] or 192 cores in NVIDIA Kepler GK110 [57]) implementing SIMT execution model. Number of SMs inside a GPU chip is determined based on the compute-power demand and other design constraints (e.g. area/power budget). In addition to the multiprocessors, a GPU chip consists of L2 cache banks, an on-chip interconnection network (e.g. a crossbar), on-chip DRAM controller and other I/O interfaces and scheduling/management logic.

An application kernel (e.g. written in CUDA) is formed as a multi-level hierarchy of threads in a form of multi-dimensional grids of blocks (i.e. cooperative thread arrays or CTAs). Depending on the availability of on-chip resources (e.g. number of registers), each CTA is capable of covering up to N (e.g 1024) threads in a multi-dimensional structure. Threads running a kernel (i.e. shader program) can use different available programmable-visible memory units: local (registers), shared-memory (scratch-pad), and global (device memory or DRAM). There are different types of caches as well to accelerate accesses to different address spaces (e.g. texture cache for caching texture/state data located in the frame-buffer on DRAM). The thread hierarchy is designed such that threads in a block run on one SM while using shared-memory to communicate and internal barriers to synchronize. Threads in different blocks do not have any means for direct communication (unless through global
memory and via atomic operations). As a high-throughput device, and to maximize occupancy of all the available compute power inside a GPGPU, an SM pipeline is designed to compensate for any long-latency operation (e.g. global load/store, specific ops) by having the ability to maintain and manage many threads (with cheap context switching cost) and to create a fair balance/overlap between time spent for computation and data-transfer. Note that higher number of threads per block can lead to higher occupancy while at the same time decrease the share of each thread for registers and thus increasing total cost of spilling to lower-level memory units.

When a compute kernel is launched (with the dimensionality set statically), a chip-wide CTA scheduling unit distribute CTAs across the SMs, in a fair and balanced round-robin fashion (e.g. CTA 0 to core0, CTA 1 to core , etc [5]. If number of CTAs is more than SMs, each SM can run more than one CTAs. Total number of CTAs that can be run on one SM is limited by the shared resources available on the SM (e.g. shared-memory size, number of registers, etc [5].

When running a CTA, SM divides threads into groups (called warps) in a transparent way such that: all the threads in a warp can proceed in the same rate (have the same PC), different warps can be at different execution points, and they can all be scheduled/issued for execution at once. Warp is a unit of scheduling and its size is determined by the number of SIMD lanes available in the pipeline (32 in our baseline). A second-level round-robin scheduler, inside the SM, searches among the ready warps and chooses one every few cycles (depending on the type of operation) to be issued/run on the SIMD lanes. An active warp will be moved to the idle-queue when it is blocked (e.g. due to memory latency) and another warp from the ready queue is chosen in the next cycle (if any).
Optimizing Stencil Computation on Short-Vector SIMD ISAs

Stencil codes are generally easily vectorized by compilers such as GCC and Intel’s ICC because they typically feature parallel innermost loops where array elements are accessed at unit stride. However, as we illustrate using a simple example below, the realized performance often falls far short of machine peak, even when all accessed data is resident in the L1 cache. The main reason for the loss of performance is that when compiling stencil codes for current vector instruction architectures, it is necessity to use either redundant and unaligned load operations or intra-register shuffle or other intra-register data reorganizing operations. We illustrate the challenges of vectorizing stencils, using an example below.

Consider the loops $S_1$, $S_2$, and $S_3$ (shown in Listing 3.1). The first loop ($S_1$) multiplies a scalar element $K$ with each element of vector $B$, and add the result to an element of vector $A$ with the same index, in order to produce vector result $A$. Assuming that the hardware vector size is 4 (as in SSE for float data), and that $A[0]$ and $B[0]$ are aligned to a boundary that is a multiple of the hardware vector size, the vector code generated by a compiler for $S_1$, in every iteration of the outer loop $t$, will use $\frac{N}{4} - 1$ aligned vector load and store operations to read/write the elements of $B$ and $A$ — compute $A[4i:4]$ by loading $B[4i:4]$ (the notation $A[i:V]$ denotes a vector
for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)
        A[i] += B[i] * K;    // S1

for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)
        A[i] += B[i-1] * K;   // S2

for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)
        A[i] += B[i-1] * B[i]; // S3

**Listing 3.1:** Vector multiply-add loops with different multiply operands: aligned-constant (S1), unaligned-constant (S2), unaligned-aligned (S3)

of \( V \) consecutive elements of \( A \) starting at index \( i \) and multiplying with a vector register containing four identical copies of the scalar \( K \).

<table>
<thead>
<tr>
<th>Code</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Core2 Quad</th>
<th>Phenom</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>4.10</td>
<td>11.36</td>
<td>3.70</td>
<td>3.84</td>
</tr>
<tr>
<td>S2</td>
<td>3.75</td>
<td>7.80</td>
<td>0.87</td>
<td>2.71</td>
</tr>
<tr>
<td>S3</td>
<td>2.83</td>
<td>6.51</td>
<td>0.83</td>
<td>2.27</td>
</tr>
</tbody>
</table>

**Table 3.1:** GFLOP/s for S1, S2 and S3 on different machines for \( N=1024 \) and \( T=500000 \).

With loop S2, the number of vector multiplication operations and the number of vector load/store operations is the same as for S1, but either \( A \) or \( B \) will require unaligned load/store operations. With loop S3, in addition to unaligned loads, redundant loads or intra-register data movement operations will be required since an overlapping and unaligned vector is involved in the multiplication of \( B[4*i-1:4] \) and \( B[4*i:4] \). Table 3.1 shows the performance of S1, S2, and S3 on four different processors (details of the hardware platforms are provided in Section 3.3). It may be seen
that on all platforms, the performance of $S2$ is worse than $S1$ and $S3$ is worse than $S2$. All three statements execute the same number of vector arithmetic operations and process the same number of distinct data elements. The difference in performance is due to the overheads incurred by one or more of the following: i) redundant load of data elements into different "slots" in different vector registers, ii) unaligned loads instead of aligned loads, and iii) shuffle or alignment operations to move data elements into a different position in a vector register. With current short-vector SIMD instruction set architectures, stencil computations will incur overheads similar to that of $S3$, limiting achievable performance.

In this chapter, we propose StVEC, an architectural solution with compiler support to address the problem. The key idea is to enhance the addressing modes for vector operands in vector arithmetic instructions, by allowing elements from a pair of registers to form one of the operands. We present an architectural design along with a compiler algorithm for generating intrinsics-based code for the extended architecture. Using a number of stencil benchmarks, we experimentally evaluate the effectiveness of the approach by using an optimistic and pessimistic emulation of the approach on four platforms. To the best of our knowledge this is the first hardware-based solution along with compiler support to address the performance limitations of current short-vector SIMD architectures for stencil computations. We note that in the following, we assume complementary loop transformations such as loop tiling [85] have been performed to control data cache misses, ensuring the loop nest to be considered accesses data that fits into the L1 cache. The selection and application of such transformations is orthogonal to the work presented here.
The chapter is organized as follows. Section 3.1 presents the instruction set enhancement and the hardware implementation of a register file to enable the new addressing modes. Section 3.2 develops the code generation algorithm through a sequence of examples of increasing complexity and performance. The approach to experimental evaluation is discussed in Section 3.3 and Section 3.4 reports on the experimental evaluation of the proposed approach. Conclusions are provided in Section 3.6.

3.1 StVEC: A Vector ISA Extension

Stencil computation typically involves access to adjacent array elements. As a result, vectorized stencil code often uses operands that span multiple vector registers. Aligning vector operands requires additional loads and/or shuffle operations even though the needed operands are already in the register file (albeit unaligned). In order to avoid unaligned loads or operand manipulation instructions, we propose StVEC, an ISA enhancement for stencil computation to change how vector operands are addressed and read from the physical vector register file and to allow automatic alignment of operands when needed.

3.1.1 StVEC Functionality: An Example

Considering the stencil example in Listing 3.1, we chose two different versions of the S3 loop generated and vectorized by Intel ICC compiler for two different x86-based machines. Figure 3.1 (-a and -b) show, for each version, the content of vector registers and the assembly code for each loop. It also shows the same loop when StVEC ISA is targeted. Note that only for demonstration purposes, we use Intel’s SSE instruction notations (i.e. xmm register names, etc).
Figure 3.1: Illustration of VRF use and assembly code for code from Listing 3.1 (S3): (a) Using extra vector load and alignment instructions, (b) Using unaligned vector loads and (c) using StVEC.

In order to vectorize the loop nest in Listing 3.1(S3), at every iteration of the innermost loop $i$, the vector multiplication ($B[i-1]*B[i]$) requires two operands whose memory alignments are different. Using stride 4 for vectorizing the loop, and assuming index $i$ is aligned in memory, first vector operand ($B[i-1]$) is an unaligned access (i.e. vector load) to memory while the second operand ($B[i]$) is aligned. Based on the cost estimation for underlying architectures, vectorizing compilers (e.g. Intel ICC) tend to generate different vector instructions in order to deal with such overlapping memory accesses. The following subsections describes two possible existing compiler solutions along with the approach proposed by StVEC. We assess code efficiency of the different approaches in terms of number of overhead instructions (i.e. unnecessary/redundant vector load, register alignment, register copy, etc) generated per stencil computation (multiplication, in this case).
Extra vector load with alignment  One way to generate code for building an unaligned operand is to load two consecutive vector slots from memory and combine them using data manipulation instructions (i.e. \textit{shuffle} or \textit{palignr}). This case (shown in Figure 3.1(a) as generated code for Intel Core2 Quad) is cost-efficient for architectures where unaligned loads are either expensive or not supported. As generated assembly code shows, \textit{palignr} takes two consecutive aligned vector operands in order to generate the unaligned multiplication operand. Note that this approach requires registers to hold copies due to limitation in a destructive instruction format where a source operand and the destination must be the same. In terms of code efficiency, this solution requires four overhead operations (two loads, one copy and one shuffle) for every single vector multiplication.

Unaligned vector load  For architectures where unaligned loads are successfully implemented with lower costs (i.e. Intel Nehalem), vectorizing compilers generate the code as shown in Figure 3.1(b). Neither multiplication operands requires extra permutations in this case. We only need two registers ($xmm_1$ and $xmm_2$) to hold unaligned (starting at $B[i - 1]$) and aligned (starting at $B[i]$) operands, respectively. However, for vector ISAs where unaligned vector load is not supported (i.e. in IBM Power), this approach is not practical. In terms of code efficiency, this solution executes two unaligned loads for every vector multiplication.

As described above, and can be observed from the $xmm$ register contents, the first solution suffers from unnecessary memory accesses, register copy and also the necessity for aligned memory access instructions. Second approach (Unaligned vector load), however, suffers from redundant memory accesses due to overlap between the
two vector operands (i.e. $B[i]$, $B[i+1]$, and $B[i+2]$ are being loaded twice, for $xmm1$ and $xmm2$). This cost is in addition to the necessity for architectures to support unaligned memory access.

**StVEC (no shuffle and no unaligned load)** As shown in Figure 3.1(c), using StVEC, one can load vectors one by one using *aligned* vector loads. Then, by a simple hardware support in the vector register file (VRF), vector elements can be read from two different vector registers (i.e. to build an unaligned operand). This solves the overlapping vectors because elements are loaded once but can be (re)used many times. As a result, once loaded into the VRF, different elements of each vector register can be indexed to construct different vector operands; thus eliminating the need for redundant memory accesses. In other words, StVEC postpone the operand manipulation to the register read stage of the pipeline, in order to avoid executing ”overhead” instructions to build operands. In fact, as shown in Section 3.2, using some code optimization techniques (e.g. ”software pipelining”), StVEC can improve the stencil computation performance by only executing one *aligned* vector load per stencil computation. Providing more details, the following is how StVEC’s execution model works in practice.

### 3.1.2 StVEC Execution Model

StVEC introduces a set of new arithmetic instructions, that can handle one unaligned operands without the need to execute overhead instructions. To simplify the hardware, and also due to the ”destructive” instruction format (i.e. first source operand is the destination as well), the proposed design for StVEC only supports unalignment for the second source operand. It the second operand is aligned, the StVEC
code generation uses a normal vector arithmetic instruction. Otherwise, and in order
to build such an unaligned operand, an StVEC instruction requires three pieces of in-
formation: a *base* register index, an *extension* register index, and an offset value. The
*base* register is used to locate the vector register in which the first group of elements
of the source operand is stored. The *extension*, however, determines which vector
register contains the second group of elements. The *offset* is used to identify where
the first element of the first group located in the *base* register. Thus, considering the
example in Figure 3.1(c), in order to build an operand containing \( B[i - 1 : 4] \), an
adequate indexing information would contain register specifiers 0x01 and 0x02 (for
\( xmm1 \) and \( xmm2 \) as the base and extension registers, respectively) and the offset
value of 0x03. As discussed in Section. 3.2, since vector elements in the *base*
register could be loaded in previous iteration(s) of the vectorized loop, it is required to
generate register-copy instruction(s) to circulate these temporary buffers. However,
using other compiler optimizations (i.e. "software pipelining"), we will remove the
extra register copies as well. Note that, due to the inherent stride-1 access pattern
for stencils (that are considered in this work), knowing one offset value is sufficient
to identify all the vector elements. Meaning, for offset value of \( i \), vector width of \( W \),
and the two given source operands, we can find the elements in two groups: first from
\([i:W - i]\) in *base* register and second from \([0:i]\) in the *extension* register. Also, in gen-
eral, one can assume that the two *base* and *extension* registers are distinct (and not
necessarily always consecutive registers such as \( xmm1 \) and \( xmm2 \) in Figure 3.1(c)).

From the functional perspective, for a second vector operand named *VOPR*\(_2\), *base*
and *extension* registers, \( VR_x \) and \( VR_y \), and different values of offset \( ofs \), the second
vector operand will be found as shown in Figure 3.2.
Figure 3.2: Operand encoding: (left) Two source vector registers, $VR_x$ ($X_x$ elements) and $VR_y$ ($Y_y$ elements), (right) Different permutations for second vector operand, $VR_z$, based on values of the offset.

According to the Figure 3.2, if $offset$ is zero, then second operand is the same as the base register, $VR_x$. If $offset$ is one, the decoder (i.e. at "Register Read") will select the row associated to $VR_x$ in columns (i.e. banks) 1, 2 and 3 and to $VR_y$ in bank 0, and so on. Therefore, other than offset value of zero, the first group of elements is in $VR_x$ and the second group is stored in $VR_y$. Also, number of elements in the first and second groups are $(W - offset)$ and $(offset)$, respectively, where $W$ refers to the vector width.

As suggested earlier in this section, StVEC only requires some changes to the "read-ports" of the vector register file. We stipulate a vector register file design that includes four banks, with each bank containing a single word (32-bits) of the multi-word register. Normally all four banks would be accessed with the same address to access a single register. StVEC changes this design to allow each bank to be accessed with a different address. In addition, each bank can provide an element associated to any position in the final output. No changes to the write port to the register file are
needed since all the realignment operations are done when reading from the register file.

For the rest of this chapter, a generic notation (i.e. \textit{stvadd} for vector-add in StVEC format) and 128-bit wide vector elements are considered as the reference cases to which the StVEC extension will be introduced in details. However, StVEC can be extended to support all the primitive vector arithmetic instructions (i.e. addition, subtraction, multiplication and division) in the existing vector ISAs.

### 3.1.3 StVEC Instruction Format

There are three source operands as vector registers (VR) and an additional 8-bit immediate value encoded in an StVEC instruction. First operand is the \textit{imm8} value for offset, \( k \). Second and third places are taken by \textit{base} and \textit{extension} registers, respectively. The last operand is used for destination vector register. Four primitive StVEC single-precision floating point vector operations are shown in Table 3.2 (where normal vector instructions are shown too, for side-by-side comparisons). The same pattern is used for double-precision instructions as well.

Table 3.2: Vector arithmetic instructions: Standard vs. StVEC (W: vector width).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{vadd} ( VR_x, VR_y )</td>
<td>( VR_x + VR_y )</td>
<td>\textit{stvadd} ( k, VR_x, VR_y, VR_z )</td>
<td>( VR_x + (VR_y[k : W-k], VR_y[0 : k]) )</td>
</tr>
<tr>
<td>\textit{vsub} ( VR_x, VR_y )</td>
<td>( VR_x - VR_y )</td>
<td>\textit{stvsub} ( k, VR_x, VR_y, VR_z )</td>
<td>( VR_x - (VR_y[k : W-k], VR_y[0 : k]) )</td>
</tr>
<tr>
<td>\textit{vmul} ( VR_x, VR_y )</td>
<td>( VR_x \times VR_y )</td>
<td>\textit{stvmul} ( k, VR_x, VR_y, VR_z )</td>
<td>( VR_x \times (VR_y[k : W-k], VR_y[0 : k]) )</td>
</tr>
<tr>
<td>\textit{vdiv} ( VR_x, VR_y )</td>
<td>( VR_x / VR_y )</td>
<td>\textit{stvdiv} ( k, VR_x, VR_y, VR_z )</td>
<td>( VR_x / (VR_y[k : W-k], VR_y[0 : k]) )</td>
</tr>
</tbody>
</table>

Note that StVEC instructions are designed as register-register type such that all source operands are vector registers. Register-memory or register-immediate formats have to be converted by the compiler to sequence of move-compute operations in
order to be mapped to register-register style. To illustrate how to use the StVEC format, suppose *stvadd* instruction is generated with the following operands, for a vector width \( W = 4 \):

\[
\text{stvadd} ~ \$3, VR_2, VR_0, VR_7
\]

According to the Table 3.2, the execution results in adding \( VR_7 \) with an operand whose first element is taken from \( VR_2 \) and the second three elements are taken from \( VR_0 \). So, we have:

\[
VR_7 = VR_7 + VR_2[3:1], VR_0[0:3]
\]

### 3.1.4 Decoding StVEC Instructions

Decoding StVEC instructions requires special handling of the *base* and *extension* registers and also the offset value. The Decoder generates distinct addresses for each of the vector register banks using the Bank Address Generator (BAG) logic. The BAG logic uses *base* and *extension* register specifiers (7-bit each, in this case) plus the offset value to compute the address for each register bank. The four bank addresses will be carried, along the other information, with the vector operand to the operand-read stage where they will be fed to the register file. Note that the BAG logic can be implemented anywhere in the pipeline, after the register renaming and before the register-read stages.

StVEC instructions read their second operand in two different registers. As a result, they are dependent on three registers rather than two in the case of regular vector operations. These dependencies have to be enforced by the out-of-order
scheduling logic. For instance, if the processor uses reservation stations to store pending instructions, reservation station entries need to have one additional pointer to the instruction generating the third register value. The same is true for a reorder buffer-based implementation. The scheduling logic has to enforce these dependencies and not allow the dispatch of an instruction until all dependent registers are available. In theory, adding an extra dependency could slow down execution. In practice this is not an issue because these are true dependencies for stencil codes and the input operands have to be in the register file anyway before execution can proceed.

3.1.5 Modified Vector Register File

The second change in the pipeline is re-structuring the vector register file (VRF). In general, a VRF is multi-bank memory structure where each bank $i$ stores the $i^{th}$ element of vector operands (i.e. words). Thus, number of banks is equal to the vector width and number of such physical registers is the same as number of rows in the VRF. In order to demonstrate the architectural changes, we use an example VRF design containing 128 registers of 128-bit wide each (4 32-bit banks). Note that only read ports of the VRF banks have to be modified, and write ports are not subject to any hardware changes.

To read a 4-wide vector (128-bit), a 7-bit register specifier is fed into the VRF. The corresponding read-port decoder activates word-select lines of the banks accordingly which causes the same row to be selected in all the four banks. When words are read from the banks, i.e. at the end of the register-read stage, slot 0 of the output vector operand contains a word from bank 0, slot 1 from bank 1 and so on. In this normal
vector read operation, words are placed in the appropriate output slots such that no adjustment operation (i.e. shift or rotate) will be required.

However, in order to support StVEC execution model, VRF has to be modified in the following way. Each bank is provided with its own 7-bit address. Instead of having a single decoder feeding the signals (i.e. bit/line select) into all four banks, each bank is outfitted with its own decoder. This is designed to facilitate read accesses to arbitrary registers of each bank. In addition, each bank can provide elements associated to any position in the final output. To support this, we add Vector Register Adjustment (VRA) logic to the output of the VRF. The VRA shifts the register elements to the appropriate positions in the operand (e.g. a block from bank 2 is moved to position 0 in the output when the offset is 0x02). The new VRF design is shown in Figure 3.3.

Figure 3.3: Vector register file (VRA) augmented with adjustment logic.
VRA logic is similar to a shifter except for the offset value which does not directly imply the "shift amount". Figure 3.3 also shows the mapping between the offset value, output elements of the banks ($B_0$, $B_1$, $B_2$, and $B_3$) and also the final adjusted elements ($W_0$, $W_1$, $W_2$ and $W_3$).

For aligned operands (offset zero), $B$ elements will be assigned to $W$s, with no shifting involved. But, in cases where offset is not zero (i.e. an unaligned operand), the VRA connects $B$ elements to appropriate output $W$ slots in order to build the final "aligned" output.

### 3.1.6 Generalizing StVEC

The requirement for an architecture to support StVEC execution model is to be able to decode the proposed instruction format and feature the vector register file such that arbitrary elements spread across different rows can be obtained by one register read operation. Such a general extension can be implemented on top of the existing SIMD ISAs, such as Intel’s SSE or AVX families and IBM-Freescale-Motorola’s AltiVec (known as VMX) family. Note that performance improvement achieved by StVEC extension (as will be shown in Section 3.4) substantially depends on the underlying architecture and on the penalty paid by executing unaligned memory accesses and data manipulation instructions.

### 3.2 Code Generation for StVEC

In this section, we describe the compiler algorithm for code generation. We first discuss how to create vectorized code for stencil loops using standard vector intrinsics. Then we show how to generate code to use StVEC instructions.
3.2.1 Program Representation

The code generation algorithm operates on an abstract syntax tree (AST) representation of the input program, suitable for detection of innermost loops as well as complex loop transformations such as peeling, unrolling and software pipelining. We assume the code is in three-address form, in order to simplify the process of copying and/or moving specific operations in the loop. The focus of our algorithm is innermost loops that are vectorizable; we assume the required transformations have been done beforehand to expose such loops [42].

We assume candidate innermost vectorizable loops have the following properties: 1) loop bounds are expressions that do not change during an execution of the loop, 2) the loop induction variable increments by steps of 1, 3) dependence analysis ensures the absence of loop-carried dependences in the loop, and 4) the loop has a single entry and single exit point.

We require all memory references in the innermost loop to be of stride-0 or stride-1. That is, for all memory references, two consecutive iterations of the loop must either access two consecutive data elements in memory (stride 1) or the same element in memory (stride 0). Note that stride-1 implies that the innermost loop iterator appears only in the right-most dimension of an array reference for row-major implementation of arrays in languages like C/C++.

Without loss of generality, for the context where the StVEC-based code generation is performed, the expression $expr$ used to dereference a memory address (e.g. in $A[\ldots][expr]$) is of the form

$$expr = liexpr + iterator + c$$
where \( liexpr \) is an arbitrary expression of program symbols whose value is loop invariant during loop execution, \( iterator \) is the loop iterator and \( c \) is an arbitrary scalar constant. For instance, in the reference \( A[i][42*N + j + 3] \) with \( j \) as the innermost loop iterator, \( 42*N \) is a loop invariant expression if \( N \) is never assigned in the loop \( j \), and \( c = 3 \) is the scalar constant for the reference.

### 3.2.2 Auto-vectorization Using Intrinsics

In the following, we present a target-independent algorithm to generate vector intrinsics using vectors of size \( W \), with abstract intrinsics such as \( vadd \), etc. to represent vector operations. Our experiments (discussed in Section 3.4) are based on the SSE and AVX vector instruction sets, but the code generation approach can be used with other vector instruction sets such as Altivec, LRBni, etc.

![Figure 3.4: StVEC code generation example: original loop (a), intrinsics translation (b), intrinsics plus software-pipeline (c) final StVEC code (d).](image-url)
Basic Vector Intrinsics Generation

The input to this algorithm is a representation of an innermost vectorizable loop that conforms to the conditions stated above. We now describe a systematic vector code synthesis scheme to translate this loop into a SIMDized loop, using standard vector intrinsics (such as SSE intrinsics; but we use an abstract notation instead of a target specific notation).

The first stage of the algorithm is to create a basic SIMDized version of the loop, where each stride-1 read memory reference in the scalar code is translated to a vector load in the generated code. Note that to preserve clarity, we outline a general algorithm operating on abstract vector operations, which does not distinguish between aligned and unaligned data. Later in this section, we explain how code can be generated using only aligned loads exclusively, thereby avoiding any overheads of unaligned loads.

Algorithm 3.1 shows the first stage of code generation for StVEC instructions. On each candidate innermost loop $L$, we first peel a suitable number of iterations at the end of the loop, so that the number of vectorized iterations is a perfect multiple of the vector length. Next step is to change the loop to increment by the vector length, $W$. Then, for all variables $V_0$ with stride-0 access in $L$, we splat $V_0$ into a vector temporary $V_{tmp}$ and substitute all the references to $V_0$ in $L$ with $V_{tmp}$. After that, for all read references $R_1$ with stride-1 access in $L$, a vector load $VL_{tmp}$ is created, which then substitutes the reference $R_1$. Also, for all write references $W_1$ with stride-1 access in $L$, we create a vector store $VS_{tmp}$ to substitute the reference $W_1$. After all these changes, we remove unnecessary loads and store, which typically come from multiple reads to the same address. We also compute use-def chains to remove loads
Algorithm 3.1: Generating basic SIMDized version of an innermost loop.

Input: \( L \): innermost loop, \( W \): vector width

Output: basic SIMDized \( L \)

begin

/* 1. peel the loop and update the trip count */
if number of iterations of \( L \) not divisible by \( W \) then
  peel \( k \) iterations, where \( k = N \mod W \)
end

change \( L \) to increment by \( W \)

/* 2. generate and insert vsplat */
forall the stride-0 variables \( V_0 \) in \( L \) do
  \( V_{\text{tmp}} \leftarrow \text{vsplat}(V_0) \)
  insert \( V_{\text{tmp}} \) before the loop \( L \)
  forall the references \( RW_0 \) to \( V_0 \) do
    \( RW_0 \leftarrow V_{\text{tmp}} \)
  end
end

/* 3. generate and insert vload */
forall the stride-1 read references \( R_1 \) in \( L \) do
  \( VL_{\text{tmp}} \leftarrow \text{vload}(R_1) \)
  \( R_1 \leftarrow VL_{\text{tmp}} \)
end

/* 4. generate and insert vstore */
forall the stride-1 write references \( W_1 \) in \( L \) do
  \( VS_{\text{tmp}} \leftarrow \text{vstore}(W_1) \)
  \( W_1 \leftarrow VS_{\text{tmp}} \)
end

/* 5. clean-up */
remove unnecessary vload and vstore instances in \( L \)

/* 6. insert vector ops */
forall the scalar arithmetic operations \( op \) in \( L \) do
  \( op \leftarrow \text{vectorIntrinsic}(op) \)
end
end

and stores to temporaries. Finally, we substitute all arithmetic operations with their vector equivalent.

We illustrate the application of the algorithm using the simple example shown in Figure 3.4(a). The translation code with vector intrinsics is shown in Figure 3.4(b).
Note that for simplicity the lower boundary of the loop \((lba)\) in Figure 3.4 is assumed to be an aligned version of the original one \((lb)\).

**Software Pipelining**

Algorithm 3.1 presents a simple translation of a loop into its vector equivalent, by using multiple vector loads for adjacent memory accesses. In order to improve the efficiency of the generated code, we perform software pipelining [47]. The objective is to overlap computation and data movement, benefiting from instruction-level parallelism. We illustrate this with 2-stage pipelining, where data is fetched one iteration ahead of its use.

Algorithm 3.2 demonstrates our software pipelining approach. We first make a copy of the relevant \(\text{vload}\) operations before the loop. Then, we rename the associated vector variables from \(VX\) to \(VX_1\) in the copy created. The \(\text{vload}\) is then re-timed by one iteration in the loop body, and the associated vector variables are renamed from \(VX\) to \(VX_2\). We then change references to \(VX\) into \(VX_1\) in the arithmetic vector operations in the loop body. After that, a copy of the relevant vector operations (\(\text{vstore}\) and arithmetic vector operations) is made after the loop. We then rename the associated vector variables from \(VX\) to \(VX_2\) in the copy created. Afterwards, we peel the last iteration of the loop and unroll it by two, as we use a two-stage software pipelining, to avoid the need for variable swap. In the part of the loop body corresponding to the second loop iteration unrolled, we substitute all references to \(VX_1\) by \(VX_2\), and conversely. Returning to the above example, the software-pipelined version is shown in Figure 3.4(c).
Algorithm 3.2: Generating software-pipelined version of each SIMDized loop.

Input: \( L \): SIMDized innermost loop, \( W \): vector width
Output: \( L \): software-pipelined innermost loop

begin
/* 1. replicate and re-time vloads, e.g. \( Vx = \text{vload}(\text{addr}) \) */
forall the \( \text{vload} \) instances \( Vx \) in \( L \) do
    \( Vx_1 \leftarrow \) copy of \( Vx \) with loop count = 0
    \( Vx_2 \leftarrow \) re-timed version of \( Vx \) by one iteration
    insert \( Vx_1 \) before the loop \( L \)
    forall the \( \text{vector arithmetic ops using variable} \ Vx \ \text{in} \ L \) do
        replace the reference \( Vx \) with \( Vx_1 \)
    end
end
/* 2. change vstors (\( \text{vstor}(\text{addr}, Vx) \)) and vops (\( Vz = \text{vadd}(Vx,Vy) \)) */
forall the \( \text{vstore or vector arithmetic instances} \ \text{OP} \) do
    if \( \text{OP} \) uses variable \( Vx \) referenced by a vload in \( L \) then
        \( \text{OP}_x \leftarrow \) copy of \( \text{OP} \) with references \( Vx \) replaced with \( Vx_1 \)
        insert \( \text{OP}_x \) after the loop \( L \)
    end
end
/* 3. peel and unroll (twice, for two-stage software pipelining) */
peel the last iteration of the loop \( L \) and unroll by two
forall the \( \text{references} \ Vx_1 \ \text{and} \ Vx_2 \ \text{in unrolled iteration of} \ L \) do
    substitute \( Vx_1 \) with \( Vx_2 \) and conversely
end
end

3.2.3 Integration of StVEC extension

We now present the code generation technique to use the StVEC ISA extension we have proposed. Our approach is based on the introduction of four new vector intrinsics, and the required modification to our vector code synthesis algorithm to use them.
<table>
<thead>
<tr>
<th>Standard</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 = \text{vadd} (V_2, V_3)$</td>
<td>$V_1 = \text{stvadd} (V_2, V_3, V_4, \text{offset})$</td>
</tr>
<tr>
<td>$V_1 = \text{vsub} (V_2, V_3)$</td>
<td>$V_1 = \text{stvsub} (V_2, V_3, V_4, \text{offset})$</td>
</tr>
<tr>
<td>$V_1 = \text{vmul} (V_2, V_3)$</td>
<td>$V_1 = \text{stvmul} (V_2, V_3, V_4, \text{offset})$</td>
</tr>
<tr>
<td>$V_1 = \text{vdiv} (V_2, V_3)$</td>
<td>$V_1 = \text{stvdiv} (V_2, V_3, V_4, \text{offset})$</td>
</tr>
</tbody>
</table>

Table 3.3: StVEC new vector intrinsics.

New Intrinsics Proposed

Standard vector intrinsics such as \texttt{vadd}, \texttt{vmul}, \texttt{vsub} and \texttt{vdiv} operate on two vector variables and perform the arithmetic operation on these two vectors. Our extension consists in four new intrinsics that each use three vectors and an offset. They are shown in Table 3.3.

For the StVEC arithmetic vector operation, the second vector operand is built using parts of $V_3$ and $V_4$. The \texttt{offset} argument is used to specify how many elements come from $V_3$ and how many from $V_4$, as discussed in Section 3.1.

Modified Code Generation Algorithm

In the basic code generation algorithm, there is one vector load per memory reference. When accessing $A[i:W]$ ($i:W$ represents the $W$ consecutive elements of $A$ starting from the address $i$) and $A[i+1:W]$ in the same loop iteration, two distinct loads are performed at each iteration. Neither of the vectors $A[i:W]$ or $A[i+1:W]$ is reused at the next iteration. Our ISA extension allows the formation of $A[i+1:W]$ from $A[i:W]$ and $A[i+W:W]$. An immediate benefit is the ability to reuse $A[i+W:W]$ at the next iteration.

The modified code generation algorithm works by detecting the set(s) of \texttt{vload} operations such that there are common scalar elements loaded from memory. Given
two vector loads \( \text{vload}(\text{addr}_1) \) and \( \text{vload}(\text{addr}_2) \), they load common scalar elements iff:

\[
|\text{addr}_1 - \text{addr}_2| < W
\]

If Equation (3.1) is true, then the vector arithmetic operations using these two loads can be converted into their \text{stvxx} equivalent. More precisely, the code generation algorithm constrains one of the two operands to be memory aligned. For the promotion to actually occur, we either have "\( \text{addr}_1 \% W = 0 \)" or "\( \text{addr}_2 \% W = 0 \)".

Final stage of our proposed code generation algorithm is sketched in Algorithm 3.3. The algorithm proceeds by analyzing the various vector loads generated by the previous basic vector intrinsics generation scheme (i.e. Algorithms 3.1, 3.2. We first peel the \( x \) first loop iteration(s) if \( (\text{lb} + \text{liexpr}_{\text{arrays}})\%W \neq 0, x < W \). The actual value of \( x \) is determined at run-time, so that the loop lower bound \( \text{lb}_{\text{a}} \) maximizes the number of aligned memory references in the loop. We also perform additional statement retiming to minimize the number of unaligned loads (details are provided in Section 3.2.4). Then, a basic intrinsics version, according to the Algorithm 3.1, is generated (without software pipelining). Given a set of vector loads to the same array, of the form \( \text{A}[\text{cst} + i + c : W] \), for all \( k \geq 0 \) and until unprocessed loads remain, we create additional vector loads and promote the StVEC vector arithmetic candidates to equivalent \text{stvxxx} intrinsics. A dead-code elimination step is then performed to remove vector loads made useless through the \text{stvxxx} promotion. We also perform a 3-stage software pipelining, to maximize the reuse of the vector loads for the \text{stvxxx} operations. Three stages are required to avoid register copy since the extended intrinsics address 3 vectors operands.
Algorithm 3.3: Generating StVEC intrinsics.

**Input**: $L$: software-pipelined SIMDized innermost loop, $W$: vector width  
**Output**: $L$: final loop with StVEC intrinsics

```
begin

/* 1. peel, re-time (if needed), and generate basic intrinsics */
if $(lb + \text{lexpr arrays}) \% W \neq 0$ then
  peel $x$ iterations, where $x$ is determined at run-time
end

perform statement retiming and generate a basic intrinsics version

/* 2. promote candidates to stvec format */
$VL \leftarrow$ set of all vector loads $A[cst + i + c : W]$

while $VL \neq \emptyset$ do
  $VL_p \leftarrow$ set of all vloads such that $k.W \leq |c_p| < (k + 1).W$ and $k \geq 0$
  if $\forall p, c_p \% W = 0$ or $|VL_p| = 1$ then
    continue
  end
  insert a vload $VL_{new}$ which loads from $A[cst + i + (k + \text{sign}(c_p)).W]$
  create a vload $VL_{align}$ which loads from $A[cst + i + k.W]$
  forall the vector ops $OP$ that consume a $V$ such that $V \in VL_p$ do
    if $c_p > 0$ then
      replace $V$ with $VL_{align}, VL_{new}, offset$ where $offset = c_p \% W$
    end
    if $c_p < 0$ then
      replace $V$ with $VL_{new}, VL_{align}, offset$ where $offset = W - c_p \% W$
    end
    replace the $OP$ with the equivalent $\text{stvxxx}$ intrinsic
  end
  $VL \leftarrow VL - VL_p$
end

/* 3. clean-up and optimization */
perform dead-code elimination and 3-state software pipelining

end
```

To illustrate the algorithm, we show its application on the running example in Figure 3.4(d).
3.2.4 Avoiding All Unaligned Loads

The \texttt{stvxxx} operations allow the formation of one of the operands from two registers that contain consecutive data elements. We have discussed how loading only aligned data in these two registers is enough for the \texttt{stvxxx} second operand. Further, our design requires the first operand to also be aligned. In other words, for the promotion of a vector arithmetic operation into its \texttt{stvxxx} equivalent, three registers formed with aligned data are required. This implies the equivalence of the problem of maximizing the number of promotions to \texttt{stvxxx} operations with the problem of minimizing the number of unaligned loads.

Consider $cst + i + c$, the index expression of an array used as an operand in an arithmetic operation in the original program, with $lb$ the value of the first iteration of the loop $i$. It requires only aligned vector loads if $(cst + lb + c) \% W = 0$. That is, if this property is verified for one of the two operands of each of the operations that are the first to consume data elements from the main memory, the promotion to the \texttt{stvxxx} equivalent operation is possible and no unaligned load is needed for this operation.

Each operation can be retimed freely (that is, iteration shifting is applied to this specific operation to modify which specific instance of the statements is executed in the same iteration of the loop $i$) provided all dependent operations are retimed by the same factor. Retiming changes which data element is accessed at a given iteration, i.e., affects whether or not the data elements accessed at the first iteration are aligned in memory. Consider the example in Listing 3.2:

Retiming $S$ by $+2$ leads to the code shown in Listing 3.3.
After retiming, A[i] is an operand of both operations. As soon as (lb+2)%W = 0, the vector loads required for these operations only loads aligned data. Since we can always dynamically peel iterations of the loop such that (lba)%W = 0, with x being the number of peeled iterations and lba = lb+2+x, the loop lower bound in the above example, only aligned loads are required. In general, the retiming factor \( \sigma_R \) is chosen such that, for the first consumed operand, we have \( c_R = \sigma_R \). This allows the use of only aligned loads for this operand. We generalize this reasoning by considering the only existing retiming constraint \textit{between} operations: all dependent operations are to be retimed with the same \( \sigma \) factor to ensure that semantics is preserved. Since we focus on synchronization-free inner loops, we note that there is no loop-carried dependence. This implies that for the operations \( S1, S2, \ldots, Sn \) which are
in dependence, the array index function that causes the dependence is identical in
the chain of dependent operations, i.e., $c_{S_1} = c_{S_2} = \ldots = c_{S_n}$. This implies that the
re timing factors required to align the operand’s data access are $\sigma_{S_1} = \sigma_{S_2} = \ldots = \sigma_{S_n}$,
which will preserve the semantics. By computing individual $\sigma$ factors for each set of
dependent operations in our 3-address representation, it is thus possible to eliminate
all unaligned loads on arithmetic operations.

3.3 Evaluation Methodology

The effectiveness of our design was assessed by using a number of stencil bench-
marks, using a combination of optimistic and pessimistic emulation on four different
processors, as explained below.

![Figure 3.5: Methodology to emulate and evaluate StVEC performance on real machines](image)

3.3.1 Baseline Implementation

Figure 3.5 shows the overall methodology to emulate and evaluate StVEC perform-
ance on real machines. The baseline for comparison (named `baseline` in the Figure)
was an implementation of the kernels using standard SSE intrinsics, as described in
the first part of Section 3.2. The generated codes use two-way unrolling and software pipelining to perform register loads in the loop iteration prior to use.

### 3.3.2 StVEC Implementations

Code using StVEC intrinsics was generated, as explained in Section 3.2. As the first StVEC-specific variant, and in order to check the semantic of the generated code, each StVEC intrinsic was replaced by a sequence of standard SSE intrinsics that implement the new intrinsic’s functionality. This variant was named \textit{st-func} and was only used to verify functional correctness of the generated StVEC code.

The \textit{st-func} code is then used to generate two other variants, as shown in Figure 3.5. The \textit{st-pes} variant is a \textit{pessimistic} emulation of the extended instructions in that each \texttt{stvxxx} instruction in the generated code was replaced by a sequence of two dependent vector arithmetic operations. For instance, the instruction \texttt{"stvmul 1, VR\textsubscript{0}, VR\textsubscript{4}, VR\textsubscript{7}"} would be replaced by the following two vector multiplications:

\begin{align*}
vmul & VR\textsubscript{4}, VR\textsubscript{0} \\
vmul & VR\textsubscript{7}, VR\textsubscript{4}
\end{align*}

This version is intended to mimic all data dependences of the StVEC instruction and an execution upper bound on the time required for the StVEC instruction by executing a sequence of two vector arithmetic operations available on existing processors.

The \textit{st-opt} variant is an \textit{optimistic} version that was generated by replacing the StVEC intrinsics simply with a standard SSE intrinsic for that arithmetic operation, using only one of the two paired registers in the StVEC intrinsic. This version serves as a basis for measuring a lower bound for the execution time of the StVEC based program. For the previously considered example, the \textit{st-opt} version of the \texttt{stvmul}
intrinsic would execute only one vector multiplication, "$\text{vmul VR}_7, VR_4$". Note that for any $stvxxx$, among all the three source registers (first operand, second-base and second-extension), the extension register (i.e. $VR_4$) is the latest one which is defined in the program sequence, according to our code generation algorithm.

In our evaluation, we considered the auto-vectorization performance by compiling a C version of the kernel, using the highest levels of compiler optimization. We used both ICC and GCC for our evaluation. This variant was named $\text{autovec}$, as shown in Figure 3.5.

3.3.3 Experimental Setup

The hardware platforms used for our experiments are four x86-64 based machines: Intel Sandy Bridge, Intel Core i7-920 (Nehalem microarchitecture), Intel Core2 Quad Q6600, and AMD Phenom 9850BE (K10h microarchitecture). We use the following labels to refer to the four machines: $i7$-$sb$, $i7$-$n$, $core2$ and $phenom$. Machine characteristics are provided in Table 3.4.

<table>
<thead>
<tr>
<th>Machine</th>
<th>GHz</th>
<th>Cores</th>
<th>SIMD ISA</th>
<th>Peak (GFlop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-sb</td>
<td>3.4</td>
<td>4</td>
<td>SSE4.2 + AVX</td>
<td>$\sim 56$</td>
</tr>
<tr>
<td>i7-n</td>
<td>2.66</td>
<td>8</td>
<td>SSE4.2</td>
<td>$\sim 21$</td>
</tr>
<tr>
<td>core2</td>
<td>2.4</td>
<td>4</td>
<td>SSSE3</td>
<td>$\sim 19$</td>
</tr>
<tr>
<td>phenom</td>
<td>2.5</td>
<td>4</td>
<td>SSE4</td>
<td>$\sim 20$</td>
</tr>
</tbody>
</table>

Table 3.4: Hardware platforms used for emulating StVEC instructions.
The peak throughput for the machines is shown for single-precision. The double-precision peak performance is around half that for single-precision. Vector data movement and manipulation instructions perform differently on the four platforms, even though all are x86-64 architectures.

Two compilers, GCC (version 4.4.4) and ICC (version 12.0) were used for the experimental study. Table 3.5 lists different compiler optimization options used for enabling auto-vectorization on different machines.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Common</th>
<th>i7-sb</th>
<th>i7-n</th>
<th>core2</th>
<th>phenom</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC</td>
<td>-fast</td>
<td>xavx</td>
<td>-msse4.2</td>
<td>-msse3</td>
<td>-msse4</td>
</tr>
<tr>
<td>GCC</td>
<td>-O3</td>
<td>mavx</td>
<td>-msse4.2</td>
<td>-msse3</td>
<td>-msse4</td>
</tr>
</tbody>
</table>

Table 3.5: Optimization options for ICC/GCC on different machines.

3.3.4 Stencil Benchmarks

A set of twelve stencil kernels was used to evaluate this work. The Jacobi kernels form a symmetric stencil pattern been used in many scientific computations, including image processing as well as explicit PDE solvers. We experimented with Jacobi stencils in one-dimension (2, 3, 5 and 7 points), 2D (5 and 9 points) and also 3D (27 points) with different weights for different points. We label the Jacobi kernels in the results section using dimensionality and number of points (i.e. j2d5p represents 2-D 5-Point Jacobi). The Parallel Ocean Program (POP) is an ocean circulation model that solves the three-dimensional primitive equations and computes finite-difference discretizations. The two most compute-intensive loop nests of the POP code (as labeled pop1 and pop2) differ from the Jacobi stencils in that the weights (coefficients)
are different at each grid point. The \textit{fdtd} 2D kernel represents the core computation in the Finite Difference Time Domain method, widely used in computational electromagnetics. The \textit{rician} 2D denoising kernel is used to remove noise from MRI images by repeatedly executing a stencil computation. Finally, the \textit{heattut} 3D is a kernel from the Berkeley stencil probe [40] based on a discretization of the heat equation PDE.

3.4 Experimental Results

![Figure 3.6: Average (geometric means) of relative speedup with StVEC for single precision across machines and compilers.](image)

3.4.1 Performance Evaluation

We evaluate StVEC’s performance on multiple benchmarks, across different machines and with two different compilers, and four different SIMD architectures. The results are summarized in Figure 3.6 and Figure 3.7. We show the geometric mean of execution time relative to the baseline (code variants shown in Figure 3.5). To represent StVEC, we use two code variants: \textit{st-pes} and \textit{st-opt}. To represent output of a vectorizing compiler for the same kernel, we include performance reported for
"autovectorized" (autovec) code variant. Performance is measured for operations in both single (Figure 3.6) and double precision (Figure 3.7).

StVEC demonstrates consistent performance improvement across the two different compilers (icc and gcc), on all the machines. With gcc, the StVEC performance improvement ranges from 20% on the phenom to 2.47× on the core2 for st-opt and 7% to 2.26× for st-pes. The icc improvements are very similar. Also note that both st-opt and st-pes cases are consistently higher than autovec.

StVEC performance improvement is, on average, much higher for core2 that for the other machines. This is because unaligned memory instructions are very expensive on the Core 2 machine. By eliminating these accesses, StVEC achieves a dramatic reduction in execution time.

StVEC performance improvements also scale well to double precision operations. Figure 3.7 shows average performance improvements ranging from 30 to 65% with gcc and 32 to 53% with icc for st-opt. For double precision, StVEC does not achieve as large a speedup as for single precision on the Core2 system. This is because double...
precision code uses fewer unaligned memory operations that can be eliminated by StVEC.

<table>
<thead>
<tr>
<th>Abs. GFlop/s</th>
<th>i7-sb</th>
<th>SP</th>
<th>DP</th>
<th>i7-n</th>
<th>SP</th>
<th>DP</th>
<th>core2</th>
<th>SP</th>
<th>DP</th>
<th>phenom</th>
<th>SP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>j1d2p</td>
<td>25.3</td>
<td>12.4</td>
<td>10.4</td>
<td>5.2</td>
<td>3.3</td>
<td>3.3</td>
<td>7.9</td>
<td>3.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20.9</td>
<td>10.6</td>
<td>12.3</td>
<td>4.5</td>
<td>2.3</td>
<td>3.6</td>
<td>8.9</td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j1d3p</td>
<td>22.9</td>
<td>11.5</td>
<td>12.6</td>
<td>5.5</td>
<td>3.3</td>
<td>4.3</td>
<td>9.9</td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>18.5</td>
<td>9.2</td>
<td>14.2</td>
<td>6.0</td>
<td>3.3</td>
<td>3.4</td>
<td>9.3</td>
<td>4.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j1d5p</td>
<td>21.2</td>
<td>16.6</td>
<td>13.2</td>
<td>7.8</td>
<td>5.0</td>
<td>4.2</td>
<td>11.7</td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19.5</td>
<td>8.7</td>
<td>13.9</td>
<td>7.9</td>
<td>4.6</td>
<td>3.7</td>
<td>11.9</td>
<td>6.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j1d7p</td>
<td>18.9</td>
<td>11.8</td>
<td>11.1</td>
<td>6.9</td>
<td>4.3</td>
<td>4.5</td>
<td>11.9</td>
<td>4.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.6</td>
<td>3.7</td>
<td>9.3</td>
<td>5.1</td>
<td>3.9</td>
<td>3.6</td>
<td>9.9</td>
<td>5.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j2d5p</td>
<td>31.4</td>
<td>16.6</td>
<td>13.0</td>
<td>5.8</td>
<td>5.4</td>
<td>4.9</td>
<td>10.3</td>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27.2</td>
<td>14.7</td>
<td>10.6</td>
<td>4.6</td>
<td>5.0</td>
<td>3.1</td>
<td>11.6</td>
<td>6.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j2d9p</td>
<td>24.2</td>
<td>9.1</td>
<td>13.2</td>
<td>5.0</td>
<td>3.3</td>
<td>3.3</td>
<td>8.3</td>
<td>3.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19.4</td>
<td>7.6</td>
<td>10.7</td>
<td>3.2</td>
<td>3.1</td>
<td>2.4</td>
<td>8.0</td>
<td>2.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j3d27p</td>
<td>13.0</td>
<td>8.0</td>
<td>8.4</td>
<td>4.8</td>
<td>3.5</td>
<td>3.2</td>
<td>5.8</td>
<td>3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.1</td>
<td>3.6</td>
<td>3.5</td>
<td>1.9</td>
<td>2.1</td>
<td>1.4</td>
<td>2.0</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop1</td>
<td>12.3</td>
<td>6.5</td>
<td>6.5</td>
<td>3.4</td>
<td>2.6</td>
<td>2.3</td>
<td>4.3</td>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.6</td>
<td>4.2</td>
<td>3.6</td>
<td>1.9</td>
<td>1.8</td>
<td>1.4</td>
<td>2.6</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop2</td>
<td>12.3</td>
<td>6.7</td>
<td>8.0</td>
<td>4.9</td>
<td>2.8</td>
<td>2.5</td>
<td>3.6</td>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5</td>
<td>4.2</td>
<td>3.8</td>
<td>1.9</td>
<td>2.0</td>
<td>1.4</td>
<td>2.6</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ftd</td>
<td>18.0</td>
<td>9.5</td>
<td>9.1</td>
<td>4.6</td>
<td>4.0</td>
<td>3.6</td>
<td>6.0</td>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.4</td>
<td>7.4</td>
<td>6.0</td>
<td>3.0</td>
<td>3.4</td>
<td>2.2</td>
<td>5.4</td>
<td>2.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heatut</td>
<td>17.0</td>
<td>9.3</td>
<td>8.5</td>
<td>3.3</td>
<td>4.2</td>
<td>3.4</td>
<td>6.7</td>
<td>3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14.4</td>
<td>7.8</td>
<td>6.1</td>
<td>3.2</td>
<td>3.7</td>
<td>2.4</td>
<td>5.7</td>
<td>3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rician</td>
<td>14.7</td>
<td>3.3</td>
<td>11.2</td>
<td>2.1</td>
<td>5.5</td>
<td>1.2</td>
<td>5.4</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.4</td>
<td>3.0</td>
<td>8.6</td>
<td>2.1</td>
<td>4.8</td>
<td>1.1</td>
<td>4.4</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: Absolute performance numbers for baseline code (sp-intrin): ICC (top) and GCC (bottom).

For reference, absolute performance numbers (in GFLOP/s) of the baseline kernel on different machines and compilers, and for all the benchmarks, are shown in Table 3.6.

We also take a closer look at StVEC’s performance across the benchmarks we test. Figure 3.8 and Figure 3.9 show relative speedup for StVEC for single and double precision benchmarks, respectively. Note that both st-opt and st-pes kernels show significant improvement over most the stencil kernels. The only exceptions are ftd which sees a performance degradation and rician which sees virtually no performance.
Figure 3.8: Summary of single-precision improvement by StVEC across machines.

Figure 3.9: Summary of double-precision improvement by StVEC across machines.

gain. The *fdtd* benchmark uses 2-point stencil pattern in the outer loop which is not beneficial with the StVEC execution model, where stencils in the innermost loop can only benefit the ISA enhancement.

*rician* is another case where there is little benefit from StVEC. Although *rician* uses stencil code, its execution time is mostly dominated by vector division operations which are very expensive across all the hardware platforms. Consequently, eliminating unaligned memory operations and shuffle operations has only marginal benefits.

Overall, StVEC achieves very significant performance improvements which are consistent across most benchmarks, different compilers, architectures and computation precision. This shows that eliminating unaligned loads and efficiently re-using
already loaded elements is beneficial for cases where unaligned memory access and data manipulation instructions are very expensive (or for architectures that do not support unaligned operations such as such as the IBM Power).

### 3.4.2 Hardware Overhead

To estimate the overhead of the additional hardware required by StVEC, we build a model of the StVEC Register File using CACTI [54]. We augment this model with delay information for the Vector Register Adjustment logic (Figure 3.3) required by StVEC. The Vector Register Adjustment logic design was synthesized using the Synopsys Design Compiler [75] for 45nm technology using Nangate’s Open Cell Library [55]. The synthesized logic is used to determine the additional delay caused by VRA.

<table>
<thead>
<tr>
<th># Regs.</th>
<th># Banks</th>
<th>Reg. size (bits)</th>
<th>BVRF (ns)</th>
<th>StVRF (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>4</td>
<td>128</td>
<td>0.24</td>
<td>0.30</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>128</td>
<td>0.26</td>
<td>0.32</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>256</td>
<td>0.34</td>
<td>0.47</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>0.37</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 3.7: Access time for baseline and StVEC vector register files (BVRF and StVRF) in 45nm CMOS technology.

Table 3.7 shows the access time for the StVEC Vector Register File (StVRF) compared to a baseline Vector Register File (BVRF). We show access time for 128 and 256-entry register files with 128 bit (4 word) and 256 bit (8 word) registers. The VRA overhead ranges from 25% to 37% of the total VRF access time. Note that the standard cell library used in the synthesis is not a production library. As a result, the delay measurements are conservative. This overhead can be reduced by using high-speed custom logic. Even with the additional overhead, the StVRF can still
be accessed in a single cycle at 3GHz (128 bit configuration) or at 2GHz (256 bit configuration).

3.5 Related Work

Several recent studies [4, 20, 16, 45, 71, 82] have reported on different aspects of optimizing stencil computations, including tiling, effective vectorization, and parallelization on shared-memory and distributed-memory systems, as well as GPUs. However, we are unaware of any work that has proposed a architecture/compiler approach to optimizing stencil computations.

Vectorization for short-vector SIMD architectures has also been a subject of much research [21, 23, 35, 56]. The majority of work on this topic addresses compiler algorithms for generating efficient code for existing SIMD architectures. In contrast, in this research, we propose a hardware/compiler approach to enhancing the performance of stencil computations on short-vector SIMD architectures.

Previous work has examined the benefits of flexible access and addressing of the register file. For instance, row-wise and column-wise access has been proposed for speeding up matrix operations [38, 68, 31]. These designs are generally more complex than StVEC because they require concurrent addressing and access to arbitrary words in the register file, including accessing the same word in different registers, needed for column-wise access. This requires a complete redesign of the register file, with word level address decoding. In [37], a flexible permutation of arbitrary-sized data blocks within SIMD registers is proposed. But unlike StVEC, it does not allow operands to span multiple registers.
Henretty et al. have proposed a software-based method to address the stream-alignment conflict of stencils [34]. Their technique requires either a program-wide data-layout transformation or a data layout conversion before and after stencil computations. In contrast, our approach imposes no global layout constraints or data layout conversion overhead.

3.6 Conclusion

This chapter has addressed a fundamental performance limiting factor with implementation of stencil computations using current short-vector SIMD instruction sets such as SSE — due to the unavoidable overhead of performing multiple loads of data elements from memory or inter-register shuffle operations. An enhanced addressing mode was introduced that allows data elements from two different vector registers to be combined to form operands for vector instructions. A hardware implementation of register files was developed to implement the enhanced addressing mode and a compiler code generation scheme was described for the enhanced vector instruction set architecture. The effectiveness of the new architecture and code generation strategy were demonstrated by using a combination of optimistic and pessimistic emulation on four different x86 CPUs.
Sparse Matrix-Vector Multiplication on GPUs

Sparse Matrix-Vector Multiplication (SpMV) is a Level-2 BLAS operation between a sparse matrix and a dense vector \((y = A \times x + y)\), described (element-wise) by Equation 4.1.

\[
\forall a_{i,j} \neq 0 : y_i = a_{i,j} \times x_j + y_i
\]  

(4.1)

As a low arithmetic-intensity algorithm, SpMV is typically bandwidth-bound, and due to the excellent GPU memory bandwidth relative to CPUs, it has become a good candidate for GPU implementations [17]. Sparsity in the matrix \(A\) leads to irregularity (and thus lack of locality) when accessing the matrix elements. Thus, even with optimal reuse for the elements of vectors \(x\) and \(y\), it is the accesses to matrix \(A\) that significantly impacts the execution time of a kernel. In particular, accessing the matrix \(A\) leads to irregular and non-coalesced memory accesses and divergence among threads in a warp. A number of efforts [7, 8, 88, 50, 87, 2] have sought to address these challenges. The NVIDIA libraries, cuSPARSE [14] and CUSP [13, 7, 8], are two of the most widely-used CUDA libraries that support different sparse representations (e.g., \(COO, ELL, CSR, HYB\)). The libraries also provide a ”conversion” API to convert one representation to another.
This chapter focuses on four representations implemented in the NVIDIA cuSPARSE library [14]: CSR [7], ELLPACK (ELL) [60], COO, and a hybrid ELL-COO (HYB) [8], on an NVIDIA Kepler GPU. A set of 27 sparse matrices, covering a spectrum of application domains and sparsity features, is selected for this study. The main purpose in this chapter is to evaluate the SpMV kernel performance of each of the four representations, for each matrix in our dataset, to determine which representation performs best. By analyzing and correlating sparsity features to the best performing representation, it is attempted to gain insights on how to determine a priori the sparse representation delivering the highest SpMV kernel GFLOP/s. We make the following contributions:

- a performance evaluation of 4 SpMV representations implemented in the NVIDIA cuSPARSE library, on a high-end GPU;

- a study of the correlation between sparse matrix features and the best kernel GFLOP/s for each representation;

- an empirical rule to select a priori the best representation on the studied dataset, based on the density and standard deviation of the number of non-zeros per row.

4.1 SpMV Representations on GPUs

To improve space utilization as well as temporal locality, many representations for storing sparse matrices have been proposed. For a sparse matrix $A$ (with $M$ rows and $N$ columns), the cost of transforming its standard adjacency matrix into a more complex representation (i.e. via an extra "pre-processing" phase) is normally amortized for iterative SpMV operations (e.g. iterative linear solvers). Some of the
well-established representations, for iterative SpMV, are implemented and maintained in the NVIDIA cuSPARSE library [14] which we explain in further details next.

Figure 4.1: Sparse representations for matrix $A$ ($M \times N$) in COO, CSR, ELL and HYB.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Memory Space Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>COO</td>
<td>$3 \times \text{nnz}$</td>
</tr>
<tr>
<td>CSR</td>
<td>$2 \times \text{nnz} + M + 1$</td>
</tr>
<tr>
<td>ELL</td>
<td>$2 \times M \times \text{max_nnz}$</td>
</tr>
<tr>
<td>HYB</td>
<td>$2 \times M \times k + 3 \times (\text{nnz} - X)$</td>
</tr>
</tbody>
</table>

Table 4.1: Space for every representation in single-precision (4-byte). Matrix $A$ is $M \times N$ with total $\text{nnz}$ nonzeros. $k$ is the cut-off point for ELL/COO partitioning in HYB where $X$ nonzeros left for COO.
Coordinate Representation (COO)

COO is most natural way of storing a sparse matrix by using three dense vectors: one to store the non-zero values, and two auxiliary vectors for storing column and row indexes of every non-zero elements. Figure 4.1 demonstrates how a given matrix $A$ can be transformed into COO representation. For a given matrix with $nnz$ non-zero values, the total memory space required for COO is $3 \times nnz$ (as listed in Table 4.1).

Compressed Sparse Row Representation (CSR)

Depending on the sparsity of the $A$ matrix, it is possible to have very few rows with many data points (e.g. power-law matrices). For such cases, storing a row index for every element will be inefficient. Instead, one can only store the number of data points in each row. Inspired by this, CSR compresses the row index array such that non-zeros of row $i$ as well as their column indexes can be found respectively at $values$ and $col_index$ vectors, and in index $r$ where $row_offset[i] \leq r \leq row_offset[i + 1]$. As one of the simplest and most widely used representation, CSR only requires $2 \times nnz + M + 1$ of space to store values and row/col indexes (as shown in Figure 4.1). CSR-Vector [7, 8] is the currently available implementation of CSR in the cuSPARSE library [14].

ELLPACK Representation (ELL)

For the sparse matrices with similar $nnz$ non-zeros per row, ELL [60] was proposed to convert the matrix into a rectangular shape by shifting the non-zero values in each row to the left. Then, each row is zero-padded such that all the rows have the same width as the row with the largest number of non-zeros. In addition to the padded $values$ matrix, the representation requires an index matrix to store the
corresponding row/col index for every non-zero element (as shown in Figure 4.1). As listed in Table 4.1, the ELL representation could be space inefficient, especially when there are very few wide rows and many very narrow ones.

**Hybrid COO-ELL Representation (HYB)**

HYB is a combination of ELL and COO representations that partitions the matrix into two parts: a dense part to be processed with ELL and a sparse part where COO could be most effective. Depending on the non-zero distribution of the matrix, a ”cut-off” point is defined (i.e. \( k \) parameter) such that first \( k \) non-zero from each row are dedicated to the ELL part. If a row has less than \( k \) non-zeros, then it is zero-padded (shown in Figure 4.1). Note that the choice of the cut-off point is matrix-dependent and could impact the overall performance.

### 4.2 Sparse Matrices and Features

In this chapter, 27 sparse matrices, from the UFL repository [18], are studied (mainly from William and LAW groups). These matrices have been used in previous studies [8, 12, 87, 2, 1]. The matrices represent a vast range of sparsity (i.e. diagonal, blocked, banded-diagonal, etc) from different application domains.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M )</td>
<td>Number of rows</td>
</tr>
<tr>
<td>( N )</td>
<td>Number of columns</td>
</tr>
<tr>
<td>( nnz_tot )</td>
<td>Number of non-zeros</td>
</tr>
<tr>
<td>( nnz_den )</td>
<td>Matrix density (%) ( = \frac{nnz_tot}{M \times N} \times 100 )</td>
</tr>
<tr>
<td>( nnz_{min,max,\mu,\sigma} )</td>
<td>Min, max, mean ((\mu)) and std-dev ((\sigma)) of ( nnz ) per row</td>
</tr>
</tbody>
</table>

**Table 4.2:** Features of an \( M \times N \) sparse matrix used in this study.
Each matrix is scanned to compute a set of features (shown in Table 4.2), on which the characterizations and analysis of this chapter are based. Note that, depending on the matrix representation, some of the features can be computed directly (without the need for the matrix to be scanned).

Table 4.3 shows key characteristics of the evaluated sparse matrices including name, number of rows ($M$) and columns ($N$), non-zero distribution parameters: total number of non-zeros ($nnz_{tot}$), min/max/mean and standard deviation of non-zeros per row ($nnz_{min}$, $nnz_{max}$, $nnz_{mu}$, and $nnz_{sig}$ respectively). The table also adds a snapshot to graphically demonstrate the spatial distribution of non-zeros in the 2D map of each sparse matrix.

4.2.1 Feature Analysis

![Figure 4.2: Matrix features: matrix density, $\mu$ and $\sigma$ (sorted by increasing $\mu$, logscale)](image)

Figure 4.2 plots for 26 of the matrices the values of three features: $nnz_{mu}$ (i.e. $\mu$, the arithmetic mean of non-zero entries in a row), $nnz_{sig}$ (i.e. $\sigma$, the standard deviation of non-zeros per row), and $nnz_{den}$ the matrix density. We have removed
Matrix (Abbrev.) | $M \times N$ | Non-Zeros (NZ) Distribution | Spyplot
---|---|---|---
uk-2002 (UK2) | 18.5M x 18.5M | 298.1M | 0.0001 | 1 | 2450 | 16.1 | 26.7 |
ljournal-2008 (LJ8) | 5.4M x 5.4M | 79.0M | 0.0003 | 1 | 2469 | 14.7 | 37 |
webbase-1M (WEB) | 1M x 1M | 3.1M | 0.0003 | 1 | 4700 | 3.1 | 25.3 |
indochina-2004 (IND) | 7.4M x 7.4M | 194.1M | 0.0004 | 1 | 6985 | 26.2 | 215.6 |
youtube (YOU) | 1.2M x 1.2M | 4.9M | 0.0004 | 1 | 2864 | 4.3 | 48.3 |
flickr (FLI) | 1.9M x 1.9M | 22.6M | 0.0007 | 1 | 26185 | 12.2 | 101.1 |
mc2depi (EPI) | 526K x 526K | 2.1M | 0.0008 | 2 | 2469 | 14.7 | 37 |
internet (INT) | 66K x 66K | 105K | 0.0024 | 1 | 7753 | 12.2 | 37.2 |
amazon-2008 (AMZ) | 735K x 735K | 5.2M | 0.0010 | 1 | 10 | 7 | 3.9 |
wiki (WIK) | 1.9M x 1.9M | 40.0M | 0.0011 | 1 | 6975 | 21.1 | 41.6 |
dblp-2010 (DBL) | 326K x 326K | 1.6M | 0.0015 | 1 | 238 | 5 | 7.7 |
internet (INT) | 66K x 66K | 105K | 0.0024 | 1 | 2639 | 1.6 | 21.1 |
eu-2005 (EU5) | 86K x 86K | 19.2M | 0.0026 | 1 | 6985 | 22.3 | 29.3 |
mac_econ fwd500 (ECO) | 207K x 207K | 1.2M | 0.0030 | 1 | 44 | 6.2 | 4.4 |
cnr-2000 (CNR) | 326K x 326K | 3.2M | 0.0030 | 1 | 2716 | 9.9 | 20.5 |
scircuit (CIR) | 171K x 171K | 959K | 0.0033 | 1 | 353 | 5.6 | 4.4 |
enron (ENR) | 69K x 69K | 276K | 0.0058 | 1 | 1392 | 4 | 28.3 |
hollywood-2009 (HLW) | 1.1M x 1.1M | 113.9M | 0.0088 | 1 | 11468 | 99.9 | 271.7 |
cop20k_A (ACC) | 121K x 121K | 2.6M | 0.0179 | 8 | 81 | 21.7 | 13.8 |
pwtk (WIN) | 218K x 218K | 11.6M | 0.0245 | 2 | 180 | 53.4 | 4.7 |
shipsec1 (SHI) | 141K x 141K | 7.8M | 0.0394 | 24 | 102 | 55.5 | 11.1 |
qcd5_d (QCD) | 49K x 49K | 1.9M | 0.0793 | 39 | 39 | 0 |
conoph (SPH) | 83K x 83K | 6.0M | 0.0865 | 1 | 81 | 72.1 | 19.1 |
cant (CAN) | 62K x 62K | 4.0M | 0.1028 | 1 | 78 | 64.2 | 14.1 |
rma10 (HAR) | 46K x 46K | 2.3M | 0.1082 | 4 | 145 | 50.7 | 27.8 |
rail4284 (LP) | 4K x 1.1M | 11.3M | 0.2410 | 1 | 56181 | 2633 | 4209.3 |
pdfHY5 (PRO) | 36K x 36K | 4.3M | 0.3276 | 18 | 204 | 119.3 | 31.9 |

Table 4.3: Set of matrices used in this study ($M$: number of rows, $N$: number of columns, $nnz$: non-zero, $\mu$: mean, $\sigma$: standard deviation)
LP for easier plotting due to its extreme value for $\mu$. The matrices are sorted by ascending value of $\mu$, and features are shown using a log-scale.

First to observe is the relatively even distribution of the dataset along the $\mu$ feature, covering well the range $3 - 100$. There are however clusters of matrices with similar $\mu$ values, as shown with matrices having almost the same $\mu$ value on this plot. On the other hand, the $\sigma$ distribution is not even relative to $\mu$: for large values of $\mu$, $\sigma$ is smaller than $\mu$, and for smaller values of $\mu$, $\sigma$ is mostly greater than $\mu$. It remains to be determined if these are fundamental properties of the applications modeled by those matrices, or if our dataset should be extended to cover more $\sigma$ cases.

It is also observed that there exists a partial decorrelation between the value of $\mu$ and $nnz\_den$ in this dataset: $\mu$ is not an accurate predictor of matrix density, as shown by the lack of ordering of the $nnz\_den$ points. However, a trend is that for the highest values of $\mu$ (the 6 matrices on the right) a higher value of $nnz\_den$ is observed: in these cases, $nnz\_den$ is above 0.01% (i.e. at least one every 10,000 matrix element is non-zero), while this fraction is generally an order of magnitude smaller for smaller values of $\mu$ in the dataset.

![Figure 4.3: Matrix density ($nnz\_den$)](image-url)
Figure 4.3 plots only the matrix density feature, sorting the matrices in increasing value of this feature. One can observe a relatively even distribution of the dataset with respect to the density feature. In later sections, more studies will show that \textit{nnz\_den} is actually a discriminant to determine which of CSR, ELL, COO or HYB should be used to get best performance.

### 4.3 SpMV Performance Characterization

Experiments in this study are conducted using a NVIDIA Tesla GPU, K40c from the Kepler generation, as described in Table 4.4. This GPU is characterized by its peak global memory bandwidth of 288 GB/s (which helps memory-bound operations such as SpMV) as well as its peak compute capability of 4.3 TFLOP/s (single-precision fused multiply-add).

<table>
<thead>
<tr>
<th>GPU Model</th>
<th>Tesla K40c (GK110B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute capability</td>
<td>3.5</td>
</tr>
<tr>
<td>#SMXs, cores per SMX</td>
<td>15, 192</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Max threads / CTA,SMX</td>
<td>1024:2048</td>
</tr>
<tr>
<td>Sh-mem (KB) per CTA</td>
<td>48</td>
</tr>
<tr>
<td>L2 / Global memory size</td>
<td>1536 KB / 12288 MB</td>
</tr>
<tr>
<td>Peak off-chip BW (GB/s)</td>
<td>288</td>
</tr>
<tr>
<td>Peak GFLOP/s (DP FMA)</td>
<td>1430</td>
</tr>
</tbody>
</table>

**Table 4.4:** GPU hosted the experiments.

The kernels are compiled by the NVIDIA C Compiler (\textit{nvcc}) version 6.5, with the maximum compute capability supported (i.e. 3.5 for the Kepler GPUs). We have also enabled standard optimizations using –O3 switch.
Four major GPU-specific sparse representations are selected from the NVIDIA cuSPARSE library [14]. Each matrix is first read in Matrix Market format [58] and stored in COO. Then it is converted into CSR from which every other representations used in this study has been generated. In these experiments, time is recorded for different steps: transforming CSR into a target representation, transferring data between CPU and GPU, and also executing the kernel. In this study, the focus is only on the kernel performance (GFLOP/s), for each representation. The question this study attempts to answer is: given the sparsity features of a matrix, what is the most compute-efficient representation to perform SpMV on GPUs. Note that the matrix size is constrained by the available global memory (i.e. DRAM) on the device.

Performance is reported in terms of computation rate (as number of floating point operations per second in GFLOP/s). Each SpMV experiment was repeated 100 times and the average (arithmetic mean) was recorded for the kernel. The analysis starts below, with the kernel performance for each representation considered.

Figure 4.4: Performance: matrices sorted by increasing $\mu$ (left) and by increasing $\sigma$ (right).
4.3.1 Kernel Performance for CSR

Figure 4.4-A plots the GFLOP/s achieved by the SpMV computation for all matrices in our dataset, where matrices are sorted by increasing $\mu$ value. Figure 4.4-B plots the same data, but sorted by increasing $\sigma$ value.

First observation to be made is that CSR performs above 10 GFLOP/s for more than 16 of the matrices, while it also goes below 6 GFLOP/s range 7 times. Although there are clear outliers, the GFLOP/s trend mainly follows the $\mu$ value: the higher the mean number of non-zeros per row, the higher the GFLOP/s achieved by CSR. This trend however is not verified for $\sigma$: there is no particular ordering of performance shown in Figure 4.4-B.

A key concern about the implementation of CSR [7, 8] is the work distribution (i.e. how nonzeros are assigned to threads). In the CSR representation, a row is assigned to a warp of threads (i.e. group of 32). Thus, a balanced execution across threads would require matrix rows to have large-enough nonzeros, in order to avoid divergence between the threads in a warp. A high $\mu$ with a relatively low $\sigma$ is expected to provide a balanced execution.

4.3.2 Kernel Performance for ELL

Figures 4.4-C:D plot similar data for the ELL representation. Numerous matrices are reported as 0 GFLOP/s as attempting to convert the input CSR matrices to this representation exhausted memory in our setup. This is typical for cases with at least one row with a very large number of non-zero (e.g., YOU), due to how ELL is generated.
It is also observed that for most of the matrices (and when applicable), ELL significantly outperforms CSR. This is particularly visible for high values of the \( \mu \) spectrum (QCD to SPH), however not for the highest values. For lower values of \( \mu \), CSR generally outperforms ELL. Figure 4.4-D provides interesting possible correlation between the ability to perform ELL conversion in our setup and/or get good performance, and the value of \( \sigma \): matrices for which ELL provides good GFLOP/s are almost all for small values of \( \sigma \). The ELL representation requires building rectangular matrices (from the input sparse matrix) by padding all the rows to be equivalent in size to the one with maximum \( \text{nnz} \). This leads to waste of cycles for useless computation and and transfer (for padded zeros). It also significantly increases the storage space (i.e. when nonzero distribution in rows is non-uniform). Interestingly however, GFLOP/s do not correlate fully with the \( \max - \mu \) metric (which gives the average padding needed), as shown with the higher performance for WIN than for AMZ.

### 4.3.3 Kernel Performance for COO

Figures 4.4-E:F plot similar data for the stand-alone COO representation. There seems to exist a fairly consistent GFLOP/s achieved by COO, but the absolute performance remains limited: it is essentially between 6 and 12 GFLOP/s (except three lower performing matrices). There is a partial correlation between \( \mu \) and the GFLOP/s, as for more than \( \frac{2}{3} \) of the matrices the higher the \( \mu \) value the higher the GFLOP/s. However, in our experiments, COO never outperformed both CSR, ELL and HYB for any single matrix, as shown below with the HYB results.

The nonzero-to-thread mapping in COO is done such that every GPU thread works on a nonzero element. As a result, it is required to perform atomic operations
in order to collect contributions of those threads working on elements from the same row. Additionally, an uneven distribution of nonzeros per row (i.e. high value of $\sigma$) may significantly decrease the performance because of unbalanced executions across threads. Solutions as segmented reduction [8, 67] have been proposed to decrease the atomic overhead, but the performance is still not completely invariant to the distribution of the nonzeros per row [8].

4.3.4 Kernel Performance for HYB

Figures 4.4-G:H plot similar data for the HYB representation. HYB can significantly outperform CSR, especially for the matrices where CSR was unable to achieve good performance, e.g. ENR and YOU. It is expected as HYB is a hybrid combining the best of ELL and COO. Cases where CSR fails to deliver good performance are often very well handled by either ELL or COO. However, HYB is not systematically best either: it is (significantly) outperformed by CSR for 4 matrices, and outperformed by ELL in 5 other cases. Figure 4.4-G shows a somewhat general trend of higher GFLOP/s for higher value of $\mu$, as opposed to a partial trend of lower GFLOP/s for higher values of $\sigma$ (in Figure 4.4-H).

HYB automatically selects a cut-off point (i.e. $k$) for separation of dense (i.e. ELL) and sparse (i.e. COO) parts of a matrix. Such a parameter is dependent to the target GPU as well as the input matrix. However, it has been observed that for most of the matrices, using the $\mu$ value as the cut-off point achieves a better performance than what found by the library. Carefully selecting such a parameter is key to success of using the HYB for a given matrix.
4.3.5 Conclusions

This study allowed to conclude on several aspects. First, no representation consistently dominates the other ones. In other words, a selection of the best representation done on a per-matrix case can lead to significantly better performance. Second, there is a general (but not systematic) trend of higher GFLOP/s being achieved for higher values of $\mu$, especially when discarding the low-end and high-end of the $\mu$ range. While this is an important finding, it however does not facilitate the actual selection of the best representation for a particular matrix, as all four representations studied have a similar trend. Next section discusses the various representations and shows that how density (i.e. $\text{nnz}_{\text{den}}$) is a good first-order metric to determine the best.

4.4 The Sparse Triangle

![Figure 4.5: Comparison of performance, matrices sorted by increasing %nnz](image)

4.4.1 Selecting the Best Representation

It has been observed in several previous works that no single representation achieves the best performance on all matrices. Hence an open question is: how
to select the representation that performs the best, given feature information about a matrix? The characterization conducted in the previous section illustrates such differences in representations, focusing exclusively on the kernel performance. Explained below is how to correlate the matrix density feature (i.e. \texttt{nnz\_den}) with the best representation and how to choose the best, for a given matrix.

### 4.4.2 The Role of the Matrix Density

Figure 4.5 plots kernel performance for the three representations (CSR, ELL and HYB), on different matrices. Note that COO was dropped from the list since, as previous section showed, it never outperforms any of the other three representations.

While $\mu$ had a loose correlation with the GFLOP/s achieved for all representations and $\sigma$ showed no clear correlation, the matrix density shows interesting correlations with the kernel performance. A key observation is that, on the studied dataset, one can partition the \texttt{nnz\_den} range into three sets:

- **S1** where $\texttt{nnz\_den} \in [0 - 0.00035]$: contains UK2 and LJ8 matrices.
- **S2** where $\texttt{nnz\_den} \in [0.00036 - 0.009]$: contains WEB, IND, YOU, FLI, EPI, IN4, AMZ, WIK, DBL, INT, EU5, ECI, CNR, CIR, ENR and HLW matrices.
- **S3** where $\texttt{nnz\_den} \in [0.009 - 0.35]$: contains ACC, WIN, SHI, QCD, SPH, CAN, HAR, LP and PRO.

From this partitioning into three sets, the following rules hold broadly: **use CSR for S1, use HYB for S2, use ELL for S3.** This extremely simple model successfully selects the best representation for all matrices except LP, WIN, HLW, and IND (i.e., correct prediction of the best representation for 23 out of 27 matrices).
Looking further into the four incorrectly classified matrices, Table 4.5 recalls their specific features, which representation was predicted from the $nnz_{\text{den}}$ model, which representation performs best and how much performance improvement the best representation achieves over the predicted one.

An interesting observation is that the three matrices with highest loss in performance have the highest $\sigma$ values of the entire dataset (LP, HLW and IND), and LP has the highest $\mu$ value of the entire dataset; an order of magnitude higher than the one in second position. On the dataset studied in this chapter, it appears that if $\sigma > 200$ then the $nnz_{\text{den}}$ rule does not work. It is also important to note that for IND, although CSR outperforms the ”predicted” HYB, it is only by a marginal fraction. In fact, IND is at the boundary of the range between CSR and HYB. A more interesting case is HLW, where, even though it is at the boundary between HYB and ELL, but CSR performs better (i.e. 1.45x faster) than HYB. It shows that although CSR seems most appropriate for small values of $nnz_{\text{den}}$, it may also be best for higher values of $nnz_{\text{den}}$ when $\sigma$ is large. WIN is at the boundary between HYB and ELL, and only a marginal improvement is obtained by using HYB. However one may note that, apart from ACC, there is only at best a small advantage to ELL compared to HYB, in this dataset.
LP, on the other hand, is a case of dramatic failure for the $nnz\_den$ model: ELL gives the worst performance of the three, and HYB is 22x faster than ELL on this code. LP has the highest $\sigma$ of all, and it is interesting to note that CSR is competitive on LP: it is only 1.3x slower than HYB.

### 4.4.3 Summary and Discussion

In this chapter, it has been shown that how a very simple model, based on the matrix density, can be used to determine which of the three representations (CSR, ELL and hybrid COO-ELL) should be used to achieve the best kernel GFLOP/s. However, precision of such a surprisingly simple model in predicting the best has shown to be very low as was shown for the clear outliers. This suggests considering a more sophisticated model, which could take into account other matrix features such as the mean and standard deviations of the number of non-zeros per row.

This chapter provides basics and foundations for further characterization and analysis (listed below), that will be further studied in the next chapter:

1. The role of $\sigma$ in the selection of the best representation needs to be further studied. In particular, considering more matrices with higher values of $\sigma$ can help better find possible correlations with the best representation.

2. The working ranges for the three representations has been manually derived, and is obviously very dependent on the dataset. For the next study, machine learning techniques (i.e. classifiers) are employed to systematically compute a decision tree and select the best representation. Such a decision will need to take into account $nnz\_den$ and $\sigma$ in an integrated way. But an open question
is whether such a decision tree is unique with regard to the features of the matrices, or if it also depends on the GPU architectural parameters.

4.5 Related Work

Sparse storage representations have been studied extensively for CPUs [79, 83] as well as GPUs [8, 14, 13] and accelerators [50]. For the GPUs, the optimizations have been targeted at the features of the architecture. GPU implementation of sparse representations are significantly impacted by the work distribution strategy (i.e. the way nonzeros are mapped to the thread-blocks and threads).

In addition to the standard representations (e.g. from NVIDIA cuSPARSE [14]) studied in this chapter (i.e. COO, ELL, CSR and HYB), other alternatives have been proposed. In general, and from the application perspective, sparse representations have moved towards more complex data structures for which a heavy pre-processing is required [29, 12, 87, 88, 2]. This is desirable when SpMV is run for several iterations (as opposed to where the input matrix changes frequently, e.g. dynamic graphs [1]). However, in addition to the data-structure and work distribution, GPU implementations may rely heavily on auto-tuning algorithms to find the best kernel configuration as well as runtime parameters [59, 87].

The objective in this study is not to introduce any new representation but to seek to understand how performance is related to sparsity features of the input dataset. Although this study was limited to the standard representations (implemented in NVIDIA cuSPARSE [14]), such analysis and observations can be expanded for other available representations (as future work).
4.6 Conclusion

Libraries implementing efficient SpMV operations on GPUs have been developed with a focus on efficient representation and exploitation of matrix sparsity. NVIDIA cuSPARSE implements several such representations, such as CSR, ELLPACK, COO and a hybrid scheme ELL-COO.

In this chapter, 27 different sparse matrices were evaluated on each of the four representations, characterizing the SpMV kernel performance. By reasoning on matrix features, such as the mean and standard deviation of the number of non-zeros per row, and the matrix density, a good correlation between density and the best performing representation was found. This enables choosing, a priori, the best sparse representation for iterative SpMV on a given sparse matrix. However, existence of very few outliers showed that, for such a simple to become more effective requires taking into account several matrix features, as well as covering a wider range of matrices to gain statistical confidence in the classification heuristics.
CHAPTER 5

Automatic Selection of Sparse Representation on GPUs

One of the challenging aspect of optimizing SpMV on GPUs is that the performance profile depends not only on the characteristics of the target platform, but also on the sparsity structure of the matrix. There is a growing interest in characterizing many graph analytics applications in terms of a dataset-algorithm-platform triangle (e.g. [64]). This is a complex relationship that is not yet well characterized even for much studied core kernels like SpMV. This chapter makes a first attempt at understanding the dataset-algorithm dependences for SpMV on GPUs.

The optimization of SpMV for GPUs has been a much researched topic over the last few years, including auto-tuning approaches to deliver very high performance [62, 17] by tuning the block size to the sparsity features of the computation. For CPUs, the compressed sparse row (CSR) – or the dual compressed sparse column – is the dominant representation, effective across domains from which the sparse matrices arise. In contrast, for GPUs no single representation has been found to be effective across a range of sparse matrices. The multiple objectives of maximizing coalesced global memory access, minimizing thread divergence, and maximizing warp occupancy often conflict with each other. Different sparse matrix representations, such as
COO, ELLPACK, HYB, CSR (these representations are elaborated later in the chapter), are provided by optimized libraries like SPARSKIT [62], CUSP and cuSPARSE [14] from NVIDIA because no single representation is superior in performance across different sparse matrices. But despite all the significant advances in performance of SpMV on GPUs, it remains unclear how these results apply to different kinds of sparse matrices.

The question this chapter seeks to answer is the following: *Is it feasible to develop a characterization of SpMV performance corresponding to different matrix representations, as a function of simply computable metrics of sparse matrices, and further use such a characterization to effectively predict the best representation for a given sparse matrix?*

This chapter focuses on four sparse representations implemented in the cuSPARSE and CUSP libraries [14] from NVIDIA: CSR [7], ELLPACK (ELL) [60], COO, and a hybrid scheme ELL-COO (HYB) [8], as well as a variant of the HYB scheme. To study the impact of sparsity features on SpMV performance, total of 682 sparse matrices were gathered from the UFL repository [18] covering nearly all available matrices that fit in GPU global memory. The performance distribution of each considered representation was extensively analyzed on three high-end NVIDIA GPUs: Tesla K20c and K40c, and a Fermi GTX 580, demonstrating the need to choose different representations based on the matrix features, library and GPU used.

This chapter then addresses the problem of automatically selecting, a priori, the best performing representation using only features of the input matrix such as the average number of non-zero entries per row (the average degree of graphs). This is achieved by developing a machine learning approach using decision tree classifiers.
Extensive characterization shows that the automatically generated model can achieve an average slowdown of no more than 1.05x compared to an ideal oracle selector (i.e. that systematically selects the best representation for every matrix). This work makes the following contributions:

- an extensive performance evaluation of popular SpMV schemes implemented in NVIDIA cuSPARSE and CUSP, on three high-end NVIDIA GPUs;
- the development of a fully automated approach to select the best performing sparse representation, using only simple features of the input sparse matrices;
- an extensive evaluation of numerous machine learning models to select sparse representations, leading to portable and simple-to-translate decision trees of 30 leaf nodes achieving 95% of the maximal possible performance.

5.1 Sparse Matrices and Performance Variability

From a performance perspective, for a given matrix $A$ with a certain sparsity structure, the representation is selected based on the required memory space (nonzero + meta-data), the conversion time, and the kernel execution time. As shown in Table 4.1, the required space for a matrix $A$ is a factor of the total number of nonzeros ($nnz$) in $A$, dimensions of matrix $A$ ($m \times n$), and other internal (matrix-specific) parameters (e.g., cut-off point $k$ in the case of HYB).

The representation of the sparse matrix affects SpMV performance on GPUs and none of the representations is consistently superior as shown later in Section 5.3. Table 5.1 shows four sparse matrices and performance of SpMV in GFLOP/s on a
single Tesla K40c GPU. It can be seen that each of the four representations is the best for one of the four matrices.

The choice of "best representation" for a given matrix depends on the sparsity structure (i.e. matrix features) as well as the GPU architecture (e.g. number of SMs, size/band-width of global memory, etc.). However, using representative parameters from both matrix and architecture, one can develop a model to predict the best performing representation for a given input matrix, as demonstrated in this chapter.

<table>
<thead>
<tr>
<th>Group</th>
<th>Matrix</th>
<th>COO</th>
<th>ELL</th>
<th>CSR</th>
<th>HYB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meszaros</td>
<td>dbir1</td>
<td>9.9</td>
<td>0.9</td>
<td>4.4</td>
<td>9.4</td>
</tr>
<tr>
<td>Boeing</td>
<td>bcsstk39</td>
<td>9.9</td>
<td>34.3</td>
<td>18.5</td>
<td>25.0</td>
</tr>
<tr>
<td>GHS_indef</td>
<td>exdata_1</td>
<td>9.5</td>
<td>4.3</td>
<td>31.5</td>
<td>11.3</td>
</tr>
<tr>
<td>GHS_psdef</td>
<td>bmwcra_1</td>
<td>11.5</td>
<td>26.7</td>
<td>24.8</td>
<td>35.4</td>
</tr>
</tbody>
</table>

**Table 5.1:** Performance difference for sparse matrices from UFL matrix repository [18] (single-precision GFLOP/s on Tesla K40c GPU)

<table>
<thead>
<tr>
<th>GPU Model</th>
<th>Fermi</th>
<th>Tesla K20c</th>
<th>Tesla K40c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip</td>
<td>GTX580</td>
<td>GK110</td>
<td>GK110B</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>Num. of SMs</td>
<td>16</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>ALUs per SM</td>
<td>32</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Threads per CTA</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Threads per SM(X)</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Sh-mem /CTA(KB)</td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>L2 cache(KB)</td>
<td>768</td>
<td>1536</td>
<td>1536</td>
</tr>
<tr>
<td>Glob-mem (GB)</td>
<td>1.5</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>Glob-mem (GB/s)</td>
<td>192</td>
<td>208</td>
<td>288</td>
</tr>
<tr>
<td>Tera-FMA/s (SP/DP)</td>
<td>1.5/-</td>
<td>3.5/1.2</td>
<td>4.3/1.4</td>
</tr>
</tbody>
</table>

**Table 5.2:** GPUs hosted the experiments.
5.1.1 GPUs and Compilation

Two NVIDIA Kepler GPUs (K20c and K40c) and one Fermi GPU (GTX 580) were used for our experiments. Table 5.2 shows the details for each GPU. When selecting the GPUs, this study has considered variety in global memory bandwidth (i.e. 192 vs 288 GB/s) as well as the compute capacity (i.e. total number of CUDA cores). The kernels were compiled using the NVIDIA C Compiler (nvcc) version 6.5, with the maximum compute capability supported by each device. We enabled standard optimizations using the –O3 switch.

5.2 Feature Analysis

A very rich dataset of sparse matrices, representing different sparsity feature cases, is selected from the UFL repository [18] (all available in Matrix Market format [58]). From more than 2650 matrices (from 177 groups of applications) in the repository, 682 were selected from 114 groups. These matrices included all those used in numerous previous GPU studies [8, 12, 2, 1, 87]. While previous work using sparse matrices has usually tried to group matrices based on the domain where they arise, this work instead looks to obtain a statistically relevant coverage among several key quantitative features describing the sparsity structure of the matrix.

Our selection mechanism applied the following constraints to set bounds on two metrics: the total number of non-zeros in the matrix, and the number of rows:

- **C1**: the sparse matrix does not fit in the CPU LLC (8MB for our machines).

This is to focus on matrices where GPU execution is most beneficial.
• C2: the sparse matrix fits in the "effective" free space on the global memory of the device (i.e. single-GPU execution). This space is the entire DRAM memory on the GPU board minus what is normally taken by the driver toolset.

• C3: the number of rows is large enough to guarantee minimum GPU concurrency. This is achieved by assuming that a warp works on a row; thus the minimum number of rows equals the maximum warp-level concurrency on a given device.

The size of the matrix (for C1 and C2) is conservatively computed as \( S = 16 \times \text{nnz\_tot} \), where a nonzero in COO, double-precision, needs 16 bytes (8 bytes for data plus two 4-byte indexes). It is also assumed that up to 80\% of the global memory on the device is available to hold the sparse matrix data. In C3, the maximum warp-level concurrency is computed by dividing the maximum number of threads per GPU (i.e. \( \text{Threads}/\text{SMX} \times \#\text{SMXs} \)) by the warp size.

5.2.1 Feature Set

The objective is to gather relevant information about the sparsity structure of the matrix without utilizing any domain knowledge, i.e., without considering the domain from which the matrix originates. For this work, a set of (sparsity-related) features is computed, as described in Table 5.3.

The features were computed by traversing over each of the input sparse matrix. As shown later in Section 5.5, only a small subset of these features are actually needed for predicting the best representation. Nevertheless, the use of more features could possibly improve the quality of the prediction. Note that not all the features require full traversal of the matrix. For instance, density (\( \text{nnz\_den} \)) is computed as
Table 5.3: Features of an $M \times N$ sparse matrix used in this study. A "non-zero block" refers to a set of non-zeros in adjacent rows and columns (i.e. forming small rectangular dense blocks).

\[
\frac{\text{nnz}_{\text{tot}}}{M \times N} \times 100, \text{ or mean non-zeros per row (nnz\_mu) is simply computed as } \frac{\text{nnz}_{\text{tot}}}{M}. \text{ Also note that all the computed features are exact, and no approximation (e.g. sampling of matrix rows) was used for this study.}
\]

5.2.2 Feature Distribution

The selected matrices represent a wide range of structure (i.e. diagonal, blocked, banded-diagonal, etc.) from different application domains. However, the focus in this study is to learn the non-zero distribution of a matrix without any possible knowledge or assumptions from the domain. To achieve this, three critical (and group/domain-independent) features and their distribution and correlation are discussed next.

Feature statistics Table 5.4 summarizes, across the 682 matrices, the minimum, maximum, mean and standard deviation of the main features considered.

One can observe that the dataset covers a wide spectrum of matrix sizes, with an average of 4.5 million non-zeros. The average block size represents the average number of contiguous columns with non-zeros in the rows of a matrix. As shown by the
<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Std-Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>1301</td>
<td>12M</td>
<td>377K</td>
<td>118K</td>
</tr>
<tr>
<td>$N$</td>
<td>2339</td>
<td>12M</td>
<td>399K</td>
<td>126K</td>
</tr>
<tr>
<td>$nnz_{\text{tot}}$</td>
<td>350k</td>
<td>53M</td>
<td>4.5M</td>
<td>1.7M</td>
</tr>
<tr>
<td>$nnz_{\text{den}}$</td>
<td>0.000018</td>
<td>12.48</td>
<td>0.31</td>
<td>0.92</td>
</tr>
<tr>
<td>$nnz_{\text{min}}$</td>
<td>1</td>
<td>130</td>
<td>8.90</td>
<td>11.63</td>
</tr>
<tr>
<td>$nnz_{\text{max}}$</td>
<td>3</td>
<td>2.3M</td>
<td>13k</td>
<td>8K</td>
</tr>
<tr>
<td>$nnz_\mu$</td>
<td>1.60</td>
<td>3098</td>
<td>60.46</td>
<td>108.73</td>
</tr>
<tr>
<td>$nnz_\sigma$</td>
<td>0</td>
<td>6505</td>
<td>107.38</td>
<td>213.77</td>
</tr>
<tr>
<td>$nnzb_\text{tot}$</td>
<td>8256</td>
<td>27M</td>
<td>2.1M</td>
<td>696K</td>
</tr>
<tr>
<td>$nnzb_\text{min}$</td>
<td>1</td>
<td>49</td>
<td>2.17</td>
<td>2.43</td>
</tr>
<tr>
<td>$nnzb_\mu$</td>
<td>2</td>
<td>462K</td>
<td>2203</td>
<td>2580</td>
</tr>
<tr>
<td>$nnzb_\sigma$</td>
<td>1.5</td>
<td>1333</td>
<td>21.73</td>
<td>28.95</td>
</tr>
<tr>
<td>$snzb_\mu$</td>
<td>0</td>
<td>2720</td>
<td>29.21</td>
<td>71.69</td>
</tr>
<tr>
<td>$snzb_\sigma$</td>
<td>1</td>
<td>18</td>
<td>1.77</td>
<td>0.82</td>
</tr>
<tr>
<td>$snzb_max$</td>
<td>1</td>
<td>309K</td>
<td>1101</td>
<td>365.8</td>
</tr>
<tr>
<td>$snzb_\mu$</td>
<td>1</td>
<td>126.3</td>
<td>5.27</td>
<td>6.02</td>
</tr>
<tr>
<td>$snzb_\sigma$</td>
<td>0</td>
<td>737.4</td>
<td>8.45</td>
<td>11.09</td>
</tr>
</tbody>
</table>

| Table 5.4: Statistics on main features. |

...snzb\_mu metric of 5.27, the average block size is quite low. These statistics provide sensible information about the UFL database concerning GPU-friendly matrices, and confirm, via the standard deviation of these features (last column), that this study covers wide ranges of the various features and that the selection of matrices is effective.

Figures 5.1 shows the distribution for the three critical features considered.

**Matrix density ($nnz\_\text{den}$)** This feature represents the fractional sparsity of the matrix, in a manner independent of the matrix size. Figure 5.1-top plots its value for each matrix, sorted in ascending order. It can be observed that a near perfect coverage of this feature exists, but with a lower density at the extremes of the range. This feature seems to be an excellent discriminant between matrices in our database,
from the absence of horizontal segments. This motivates its inclusion in all feature sets for prediction of the best sparse representation.

**Average number of non-zeros per row (nnz\_mu)**  Distribution of this feature is plotted in Figure 5.1-middle, using the same order along the X axis as in Figure 5.1-top, i.e., it plots nnz\_mu in increasing order of nnz\_den. This feature also shows excellent coverage across the range. There exists a mild correlation between the two features, as seen by the overall increasing trend of nnz\_mu. Nevertheless, there are significant outliers to this trend, indicating that nnz\_mu carries complementary information to nnz\_den.
Standard deviation of number of non-zeros per row \((nnz\_sig)\) This feature is plotted in Figure 5.1, also in increasing order of \(nnz\_den\). The plot shows that there is a clear decorrelation between \(nnz\_sig\) and \(nnz\_den\). This is indicative of different information being carried by \(nnz\_sig\) than \(nnz\_den\), and motivates its inclusion in all feature sets.

5.3 Performance Analysis

In order to further characterize the performance of each representation, on all matrices in the dataset, many experiments were conducted by measuring the performance of the SpMV kernel in GFLOP/s, on three GPUs (K40c, K20c and Fermi, as described in Table 5.2). This study only considers total execution time of the SpMV kernel; the data transfer time between CPU and GPU is not included. This is appropriate since the target applications are iterative SpMV schemes, where the matrix is expected to be loaded once and the kernel runs repeatedly.

Best performing representations Table 5.5 reports how many matrices, out of the 682 considered, are best executed (i.e., highest GFLOP/s achieved), with a given representation.\(^1\) Note that, for cuSPARSE, a variation of the hybrid scheme is evaluated (named HYB-MU), where the partitioning cut-off point between ELL and COO is not chosen by the library, but instead set to the \(nnz\_mu\) value of the matrix. Such a user-specified control is not available with CUSP.

First observation is that HYB performs the best with the highest number of matrices for CUSP, while for cuSPARSE ELL dominates. There exists an apparent

\(^1\)Running the latest version of CUSP library did not succeed on the older Fermi GPU; so only cuSPARSE results are included for this GPU.
consistency between the Kepler GPUs: similar trends in the distribution of the best appears for both K20c and K40c. On the older Fermi architecture, CSR actually performs consistently best, but as shown below, the true performance differences with HYB are minimal. However, care must be taken with these numbers: a more precise analysis reveals that the matrices for which CSR is best (on K40c) are not strictly the same as the ones on K20c, and similarly between CUSP and cuSPARSE. While ELL, COO and HYB have correlation, but since HYB combines the first two, then CSR remains an important representation by getting the best performance for around 20% of the matrices.

**Performance differences** Table 5.5 only shows a fraction of the information. In practice, what matters is not only which representation performs best, but also the performance difference between different "possible" candidates. Table 5.6 shows the average slowdown in always using the same representation for the entire dataset, compared to using the "actual" best representation for each matrix individually. For instance, 1.97x for COO / K40c-cusparse means that about 50% of the performance is lost by always using COO for the 682 matrices, compared to using the "actual" best representation for each of them.

<table>
<thead>
<tr>
<th></th>
<th>COO</th>
<th>ELL</th>
<th>CSR</th>
<th>HYB</th>
<th>HYB-MU</th>
</tr>
</thead>
<tbody>
<tr>
<td>K40c-cusparse</td>
<td>10</td>
<td>278</td>
<td>104</td>
<td>103</td>
<td>185</td>
</tr>
<tr>
<td>K40c-cusp</td>
<td>0</td>
<td>168</td>
<td>172</td>
<td>339</td>
<td>N/A</td>
</tr>
<tr>
<td>K20c-cusparse</td>
<td>10</td>
<td>269</td>
<td>95</td>
<td>134</td>
<td>174</td>
</tr>
<tr>
<td>K20c-cusp</td>
<td>1</td>
<td>194</td>
<td>318</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Fermi-cusparse</td>
<td>5</td>
<td>200</td>
<td>302</td>
<td>86</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 5.5: Fastest representation statistics
Table 5.6: Average slowdown over the 682 matrices when a fixed representation is used instead of the individual best

The HYB representation performs best overall, and there is overall consistency across libraries and GPUs. However, it also clearly demonstrates that using CSR can lead to a significant average slowdown on the Kepler GPUs.

**Large slowdowns** While it seems from the above that HYB is a good overall representation in terms of performance, these average numbers mask very significant slowdowns on some matrices, due to the large number of matrices in the dataset. Table 5.7 reports the number of matrices (out of 682) for which use of a fixed representation across the entire dataset results in a slowdown greater than 2x over the best representation.

Table 5.7: Number of > 2x slowdown cases
This table carries critical information and dismisses the one-size-fits-all strategy that might be suggested from the previous data. If HYB is always used, between 20 and 50 matrices would achieve less than 50% of the possible performance. Slowdowns of up to 10x can actually arise by always using HYB. The numbers are even more staggering for the other representations, where such loss of performance can occur for half of the matrices. *This data clearly motivates the need for an effective approach to select the best representation for each matrix* in order to achieve consistently good performance. This metric of the number of cases with 2x slowdown is a particular point of focus in the quality of the machine learning approach described in Section 5.5, where it is shown that successful prediction can reduce this number to no more than 6 out of 682 matrices.

**Impact of ELL and CSR** Table 5.8 focuses on the K40c, considering the 2x-slowdown cases when using HYB, and reports which representation actually performing best, and the actual average slowdown for each of these. For example, out of the 47 cases where HYB has a 2x slowdown, for 18 of them CSR was the best representation, and a speedup of 2.36x on average can be achieved by using CSR instead of HYB for these 18 cases.

<table>
<thead>
<tr>
<th></th>
<th>COO</th>
<th>ELL</th>
<th>CSR</th>
<th>HYB-MU</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CS) &gt; 2× slowdown</td>
<td>0</td>
<td>29</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>(CS) Avg. slowdown</td>
<td>N/A</td>
<td>2.36x</td>
<td>2.54x</td>
<td>N/A</td>
</tr>
<tr>
<td>(CU) &gt; 2× slowdown</td>
<td>0</td>
<td>22</td>
<td>28</td>
<td>N/A</td>
</tr>
<tr>
<td>(CU) Avg. slowdown</td>
<td>N/A</td>
<td>2.28x</td>
<td>2.75x</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 5.8:** K40c using HYB
This shows that while the HYB-MU case does not lead to any improvement greater than 2x over HYB, both ELL and CSR are equally critical to address the cases with large slowdowns. In other words, ensuring good overall performance and avoiding significant performance loss is achievable only by limiting the selection to HYB, CSR or ELL.

**SpMV performance breakdown**  Finally Figures 5.2-5.4 plot the matrix distribution of performance (GFLOPS/s) achieved by the SpMV kernel, for all configurations. The range of GFLOP/s achieved for a configuration is split into 10 buckets, and the number of matrices achieving no more than the GFLOP/s label on the X axis (and no less than the value for the previous bucket) is reported. For example, 394 matrices achieve between 8.3 GFLOP/s and 12.4 GFLOP/s using COO with K40c/cuSPARSE in Figure 5.2.

**Figure 5.2:** Histogram of overall performance of cuSPARSE and CUSP on K40c GPU.
Figure 5.3: Histogram of overall performance of cuSPARSE and CUSP on K20c GPU.

Figure 5.4: Histogram of overall performance of cuSPARSE on GTX580 Fermi GPU.

It is observed that CSR is almost never in any of the three right-most buckets, and COO is mainly represented at the low end of the spectrum, while both HYB and ELL are spread across the entire spectrum. In other words, the GPU computing power may be better harnessed for matrices where HYB and ELL perform best. Note that for the cases where ELL failed to convert from MatrixMarket format on the GPU, the
performance is as 0 (instead of removing the matrix, and in order to be consistent) which artificially increases its representation in the first bucket. Such cases of failure of conversion occur typically because of very large $nnz_{max}$ values, where ELL is not appropriate to use.

Additional study was conducted to observe if there is any correlation between the matrix density and the GF/s achieved by each representation. The set of matrices was partitioned into 10 classes, by increasing range of $nnz_{den}$, and then the average GFLOP/s was computed for each representation in this class. It can be observed that while for COO (and to a more limited extent CSR) the higher the $nnz_{den}$ the higher the GFLOP/s on average, this trend is reverted for ELL and the hybrid: the lower the $nnz_{den}$ the better the average GFLOP/s. Note that there are numerous outliers to these trends: while previous results showed that, on a restricted dataset, $nnz_{den}$ was a possible predictor of the best representation [64] (also in Chapter 4), the more extensive study in this chapter showed the need for considering additional features to properly determine the best representation in general.

5.4 Predicting The Best Representation

After the extensive characterization, this section elaborates on the machine learning approach used to automatically predict the best representation, based only on the set of pre-computed features of the input matrices.

5.4.1 Problem Learned

The findings presented so far in this chapter prove the need to develop an approach which would predict, at the very least, which of CSR, ELL or HYB is the best representation for a matrix. While this is a simpler problem instance (3 categories),
in order to achieve the truly best overall performance, each possible representation must be considered for prediction. The predictor is built of the form:

$$\text{Predictor}(\vec{x}) = c$$

where $c \in \{ \text{COO, ELL, CSR, HYB, HYB-MU} \}$ for cuSPARSE (HYB-MU is removed for this study). $\vec{x}$ is the vector of features considered as input; these are metrics computed for each dataset.

**Learning Algorithms** A classification tree is a form of decision tree where the predicted value is a category. It takes the form of a series of conditions on the input feature values, leading to a leaf containing the predicted category when the conditions are met. Figure 5.5 shows an example of an excerpt of a tree built in our experiments, the number in comment (e.g., #(22)) shows the number of matrices correctly predicted by matching the input features against each conditional, until a leaf is reached. For example a matrix with $\text{nnz}_\text{sigma} = 2$, $\text{nnz}_\text{den} = 0.00005$ is predicted as HYB here.

```
nz\_sigma < 14.9
|   | nz\_mu < 5.3
|   | | nz\_sigma < 2.05
|   | | | nz\_den < 1.0E-4
|   | | | | nz\_den < 3.0E-5: HYB-MU #(3)
|   | | | | nz\_den >= 3.0E-5: HYB   #(4)
|   | | | nz\_den >= 1.0E-4: ELL   #(22)
```

**Figure 5.5:** Classification tree example
Tree models have numerous advantages over other algorithms, including (1) the ability for a human to read and understand them, favoring knowledge transfer; (2) their excellent evaluation speed and quick training speed, enabling use with large datasets; (3) their widespread availability in machine learning libraries and ease of implementation.

In this chapter, an extensive study was conducted using the Weka library version 3.6.12 [76] on most of the available decision tree algorithms implemented. This included RandomTree, J48, BFTree, and SimpleCart. The results determined that BFTree and SimpleCart outperformed the other algorithms and therefore focused particularly on those two. BFTree [24] is a best-first decision tree classifier, while SimpleCart [9] implements a hierarchical optimal discriminant analysis using minimal cost-complexity pruning [76]. Both these algorithms can be parameterized to control the size of the tree generated, trading off generalization and prediction quality.

### 5.4.2 Training and Testing

Our experimental protocol is as follows. First the GFLOP/s for all representations and all matrices is computed, and each matrix is given a class attribute corresponding to the actual best performing representation (c above). Then, the entire set of 682 matrices is split into two sets, following the 80%-20% rule. In this two-phase validation process, the training of the model is done on the largest bucket containing 80% of the dataset, while the evaluation is done on the remaining 20%. This process is repeated 5 times, each with another 80-20 split, such that each matrix is used for testing exactly once. A stratified cross-validation is used with randomization to create each case,
using the available Weka tools. This seeks to approximately retain the original class
(c) distribution in both sets.

As far as the time needed to finish the two-phase validation, it is important to
note that the training was completed in less than 2 seconds on a standard laptop,
and the testing in less than 0.1s, for one machine learning algorithm instance.

5.5 Selecting The Best Representation

This section presents the evaluation of BFTree and SimpleCart on the prediction
problem.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>nnz_den, nnz_mu, nnz_sig</td>
</tr>
<tr>
<td>Advanced1</td>
<td>nnz_den, nnz_mu, nnz_sig, nnzb_mu, nnzb_sig, snzb_sig</td>
</tr>
<tr>
<td>Advanced2</td>
<td>nnz_den, nnz_max, nnz_mu, nnz_sig, nnzb_mu, nnzb_sig, snzb_sig</td>
</tr>
</tbody>
</table>

Table 5.9: Feature sets evaluated

5.5.1 Experimental Protocol

**Feature sets** Several feature sets were evaluated, as described in Table 5.9. The
Simple feature set is meant to capture features that are extremely easy to compute,
only \textit{nnz\_sig} needs a scan of the matrix or sampling to be obtained. The Advanced1
and Advanced2 sets on the other hand consider statistics on the number and size of
blocks of non-zeros, and typically requires a scan of the matrix, or a good knowledge
of the regularity in the sparsity pattern.

87
Algorithm parameters  Both algorithms take two parameters as argument for the training stage, determining (1) \( \text{minNum} \) the minimal number of instances at the terminal node; and (2) \( \text{numFold} \) the number of folds used in the pruning. This study has explored the Cartesian product of combinations generated with the values \( \text{minNum} = \{2, 3, 4, 5\} \times \text{numFold} = \{5, 10, 15\} \).

Invalidating the random approach  A classical random classifier is evaluated to ensure the problem cannot be addressed using a simple random selection of the best representation. As expected this approach failed, giving an average slowdown compared to the best representation of 1.7x for cuSPARSE and 4.7x for CUSP — significantly worse than always using HYB.

<table>
<thead>
<tr>
<th>cuSPARSE</th>
<th>BFTree</th>
<th>Simple</th>
<th>2</th>
<th>5</th>
<th>34.8%</th>
<th>6</th>
<th>1.07x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced1</td>
<td>2</td>
<td>10</td>
<td>34.2%</td>
<td>7</td>
<td>1.08x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advanced2</td>
<td>2</td>
<td>5</td>
<td>20.4%</td>
<td>3</td>
<td>1.04x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SimpleCart</td>
<td>Simple</td>
<td>2</td>
<td>10</td>
<td>35.2%</td>
<td>6</td>
<td>1.08x</td>
<td></td>
</tr>
<tr>
<td>Advanced1</td>
<td>2</td>
<td>10</td>
<td>18.8%</td>
<td>3</td>
<td>1.04x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advanced2</td>
<td>2</td>
<td>5</td>
<td>19%</td>
<td>3</td>
<td>1.04x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUSP</td>
<td>BFTree</td>
<td>Simple</td>
<td>2</td>
<td>10</td>
<td>21%</td>
<td>4</td>
<td>1.06x</td>
</tr>
<tr>
<td>Advanced1</td>
<td>2</td>
<td>10</td>
<td>18.8%</td>
<td>3</td>
<td>1.04x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advanced2</td>
<td>2</td>
<td>5</td>
<td>16.2%</td>
<td>3</td>
<td>1.02x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SimpleCart</td>
<td>Simple</td>
<td>5</td>
<td>15</td>
<td>25%</td>
<td>3</td>
<td>1.05x</td>
<td></td>
</tr>
<tr>
<td>Advanced1</td>
<td>2</td>
<td>10</td>
<td>19%</td>
<td>3</td>
<td>1.04x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advanced2</td>
<td>2</td>
<td>5</td>
<td>16.2%</td>
<td>1</td>
<td>1.02x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.10: Best performing model found for K40c (i.e. training one model per library)

5.5.2 Evaluation of Specialized Models

Following comes the analysis results of a first approach of training a different model for each GPU and each library independently, before discussing the use of a single model for all cases.
Table 5.11: Best performing model found for K20c (i.e. training one model per library)

Table 5.12: Best performing model found for Fermi (i.e. training one model per library)

Tables 5.10-5.12 presents statistics about the best performing models found, after exploring all algorithm parameters. The criterion retained for optimality was the minimization of the number of matrices for which a 2x slowdown or more was experienced, compared to the best representation. A total of 12 configurations was tested per entry in these tables, for a grand total of 144 models trained and evaluated per GPU, from which only the 12 best are reported. The parameters used to train the best model are shown, along with the percentage of testing matrices that were mispredicted from the testing set, the number of matrices achieving a 2x slowdown or more compared to the best performing matrices, and the average slowdown compared
to the best per-matrix representation. These tables should be put in perspective with Tables 5.6-5.7.

**General observations**  First and foremost, it can be observed that there exits a very strong gain of our approach compared to using a single representation (HYB), both in terms of average slowdown and especially in terms of number of matrices for which the choice of representation was ineffective. Even limiting to a simple feature set, made of \( \text{nnz\_den}, \text{nnz\_mu} \) and \( \text{nnz\_sig} \), an average 92% or more of the maximal performance can be achieved. This limits the number of ineffective cases to 6 or less, for both libraries and GPUs. Considering a more complicated feature set, these numbers improve to 96% and 3.

Another interesting aspect is the somewhat marginal gain that increasing the feature set provides: considering features on the block size and distribution result in a decrease of the number of ineffective cases only from 6 to 3 or less. So it appears that simple features are overall good enough for a predictor, which enhances the simplicity of the approach: such features can be computed from the size of the data structures and provided by a domain expert.

Regarding mispredictions, it can be observed an overall high number of matrices being mispredicted, ranging from 16% to 38% of the testing set in the worst case. The misprediction rate was initially used as the criterion to select the best parameter configuration. However, reducing this metric only led to larger trees, with no improvement in average slowdown. The reason for the lack of apparent correlation between the misprediction rate and the average slowdown is found in the performance distribution, and the overall comparable performance of different representations in
numerous cases. For these, our models may not predict the absolute best representation, but at least predict a representation achieving very good performance, close to the best of the selected representations. Allowing for higher misprediction rate while preserving the average slowdown and number of ineffective cases enabled the building of simpler and smaller trees, improving generalization and knowledge transfer.

Finally, it is evident that while BFTree performs overall best for the K20c GPU, SimpleCart slightly outperforms BFTree with the Fermi and K40c for CUSP. Nevertheless, with differences being so marginal, one can use either model. A more challenging observation relates to the algorithm parameters. Clearly, different parameters are needed across GPUs, and even across libraries, to achieve the best performance.

**Single parameter set** Table 5.13 shows performance statistics for use of a single parameter configuration and classification algorithm. As no algorithm/parameter setup performs best across all configurations, the one that minimizes the total number of ineffective configurations is chosen.

<table>
<thead>
<tr>
<th></th>
<th>&gt; 2x slowdown</th>
<th>Avg. slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>K40c/cuSPARSE</td>
<td>6</td>
<td>1.07x</td>
</tr>
<tr>
<td>K40c/CUSP</td>
<td>5</td>
<td>1.06x</td>
</tr>
<tr>
<td>K20c/cuSPARSE</td>
<td>8</td>
<td>1.09x</td>
</tr>
<tr>
<td>K20c/CUSP</td>
<td>6</td>
<td>1.05x</td>
</tr>
<tr>
<td>Fermi/cuSPARSE</td>
<td>6</td>
<td>1.08x</td>
</tr>
</tbody>
</table>

*Table 5.13: BFTree minNum=2, numFold=5, Simple features*
These results show a slight degradation of the results when parameter selection is avoided, making it a good candidate for single-shot training (i.e., no auto-tuning on the parameters). However, it is notable that, as the training time is a matter of a few seconds per configuration, such training autotuning on the proposed range of parameter remains feasible at library installation time. Nevertheless, running all SpMV cases on all representations to collect the classified data for this supervised learning will remain a task-consuming task.

A trained tree for K40c  This study and the analysis conclude by showing the tree produced for K40c / cuSPARSE in Figure 5.6. This very simple tree was trained on the entire dataset and as such can be embedded as-is as a predictor in the cuSPARSE library, to provide the user guidance on the possible fastest representation. It is particularly interesting to note how 102 (122-18) ELL cases are correctly predicted using only conditions on \( nnz_{\mu} \) and \( nnz_{\sigma} \), and 34 more considering \( nnz_{\text{den}} \).

5.5.3 Discussion on The Validation Approach

A two-phase validation method was presented in Section 5.5.2, where training is done on 80% of the dataset and the model is cross-validated against the remaining 20%. A three-phase validation process could be performed as follows:

- **Training phase**: train the classification trees based on the expected outputs (e.g. on 60% of the dataset).

- **Validation/Test phase**: cross-validate and estimate what the classification errors are and how well the model has been trained (e.g. on 20% of the dataset)
Figure 5.6: Tree trained on all input data, K40c
• **Application phase**: apply the best validated model on “real world” data (e.g. on the second, and last, 20% of the dataset) and speculate the quality of the best validated model.

In order to verify the effectiveness of the two-phase validation process used in this chapter, three-phase validation methodology, shown in Algorithm 5.1, has been implemented. To match the previous experiments, number of folds, $N$, is set to 5.

---

**Algorithm 5.1**: Three-phase validation methodology.

**Input**: $M$: the set of 682 matrices, $N$: number of folds

**Output**: $R$: Best trained models

begin

$R \leftarrow \emptyset$

foreach $i \in \{1...N\}$ do

/* partition $M$ into 80% and 20% sub-sets */

$(A_i, B_i) \leftarrow \{ \text{a partition of } M \mid A_i \cup B_i = M \text{ and } A_i \cap B_i = \emptyset \}$

$T \leftarrow \emptyset$

forall the algorithms and parameters do

foreach $k \in \{1...N\}$ do

/* partition $A_i$ into 80% and 20% sub-sets */

$(C_k, D_k) \leftarrow \{ \text{a partition of } A_i \mid C_k \cup D_k = A_i \text{ and } C_k \cap D_k = \emptyset \}$

/* train on $C_k$, cross-validate against $D_k$ */

$TM \leftarrow$ trained classifier on $C_k$

$T = T + \text{result of cross-validation (test) of } TM \text{ against } D_k$

end

end

/* find the best trained model */

$M_i \leftarrow$ the best trained model found on $T$

$R = R + \text{result of cross-validation (test) of } M_i \text{ against } B_i$

end

end
Algorithm 5.1 works as following: the dataset of 682 matrices is split into 80%-20% subsets (repeated 5 times randomly, as Section 5.5.2 alluded to). The 80% is used for training, on different tree algorithms (i.e. "BFTree" and "SimpleCart") and for a very wide range of the learning algorithm parameters (i.e. \texttt{minNum} and \texttt{numFold}). Each of the trained models is tested against the remaining 20% and the best one is chosen at the end; based on 1) least number of $2x$ slowdown, 2) least average slowdown, and 3) least misprediction ration. The 80% is further split into two sub-partitions (80% and 20%). This step is similar to the previous one except that the process is performed on the 80% partition (as opposed to the whole set). For all the algorithms and respecting parameter ranges, the best trained model is selected and sent as output. The outcome of Algorithm 5.1 is a set of 5 best trained models (along with the corresponding algorithm and parameters).

<table>
<thead>
<tr>
<th>Feat-Set</th>
<th>PID</th>
<th>Algorithm</th>
<th>minNum</th>
<th>numFold</th>
<th>Mispred(%)</th>
<th>Avg. SD(x)</th>
<th># &gt; 2x SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>1</td>
<td>BFTree</td>
<td>3</td>
<td>10</td>
<td>40</td>
<td>1.13x</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>BFTree</td>
<td>2</td>
<td>10</td>
<td>27</td>
<td>1.06x</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>BFTree</td>
<td>5</td>
<td>5</td>
<td>32</td>
<td>1.12x</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>BFTree</td>
<td>4</td>
<td>15</td>
<td>40</td>
<td>1.11x</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>BFTree</td>
<td>2</td>
<td>15</td>
<td>35</td>
<td>1.08x</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Avg.</td>
<td></td>
<td>–</td>
<td>–</td>
<td>34.8</td>
<td>1.10x</td>
<td>8</td>
</tr>
<tr>
<td>Advanced1</td>
<td>1</td>
<td>BFTree</td>
<td>3</td>
<td>15</td>
<td>30</td>
<td>1.09x</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>SimpleCart</td>
<td>2</td>
<td>5</td>
<td>24</td>
<td>1.06x</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>BFTree</td>
<td>2</td>
<td>10</td>
<td>36</td>
<td>1.12x</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>BFTree</td>
<td>4</td>
<td>15</td>
<td>32</td>
<td>1.09x</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>SimpleCart</td>
<td>2</td>
<td>10</td>
<td>34</td>
<td>1.07x</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Avg.</td>
<td></td>
<td>–</td>
<td>–</td>
<td>30.6</td>
<td>1.09x</td>
<td>7</td>
</tr>
<tr>
<td>Advanced2</td>
<td>1</td>
<td>BFTree</td>
<td>2</td>
<td>10</td>
<td>24</td>
<td>1.03x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>SimpleCart</td>
<td>3</td>
<td>5</td>
<td>22</td>
<td>1.05x</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>BFTree</td>
<td>2</td>
<td>10</td>
<td>23</td>
<td>1.04x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>BFTree</td>
<td>2</td>
<td>10</td>
<td>29</td>
<td>1.04x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>BFTree</td>
<td>2</td>
<td>10</td>
<td>32</td>
<td>1.06x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Avg.</td>
<td></td>
<td>–</td>
<td>–</td>
<td>26.0</td>
<td>1.04x</td>
<td>3</td>
</tr>
</tbody>
</table>

\textbf{Table 5.14:} Performance of three-phase validation (cuSPARSE library on K40c GPU). \textit{SD} and \textit{PID} denote "Slowdown" and "Partition ID", respectively.
The three-phase method is evaluated from two separate angles. First, it is important to study the quality of partitioning and how good the training is for different dataset partitions. Table 5.14 shows how partitioning the dataset impacts the choice of "best" trained model (for cuSPARSE library and on K40c GPU). For this table, each partition (shown with a partition ID, i.e. \textit{PID}) corresponds to a pair \((A_i, B_i)\) in Algorithm 5.1. From the table, one can observe that the trained model is not the same for different partitions. This can be seen from different number of cases with slowdown more than two times (i.e. \(> 2x\)) and also from the average slowdown.

The second angle to look at three-phase validation is the difference between the best models trained and found (i.e. outcome of Algorithm 5.1) versus what was presented in Section 5.5.2. This requires comparison between the aggregate results in Table 5.14 (across different partitions, labeled as \textit{Avg}) versus what was summarized (as the outcome of two-step validation) earlier in this section (in Table 5.10). The results reveal a very close and similar quality metrics for the best trained models, for the two-phase (i.e. 80-20) versus the three-phase (i.e. 60-20-20) methods. For the large dataset used in this chapter with the very good feature distribution and coverage, and given the fact that such distribution is preserved after the stratified cross validation, training a model on 60\% of the data (i.e. 400+ points) versus 80\% (i.e. 500+ points) should not change much, if any, the quality of the predictor. The three-phase method works on 60\% of the points for learning, which appear to result in high quality predictors. Also, the additional parameter tuning (i.e. the second step) in the three-phase method shows only minimal differences between parameter values. So, even a sub-optimal decision in terms of the chosen machine-learning parameters should lead to only a minor degradation in the quality of results. This is why the
60-20-20 approach gives essentially similar results to the 80-20 approach, which is what is expected for the ML problem in hand, and the dataset used.

5.6 Related Work

There is an extensive amount of work in the literature on customizing sparse matrix representations and optimizing SpMV on different platforms. This research follows our previous work [64] and raises the question of how to choose between these representations in an automated, portable and systematic way [65].

In the CPU domain, Vuduc [79] studied SpMV on single-core CPUs and presented an automated system for generating efficient implementations of SpMV on these platforms. Williams et al. [83] moved toward multi-core platforms with the implementation of parallel SpMV kernels. SpMV kernels have been studied and optimized for GPUs as well. Bell and Garland implemented several sparse matrix representations in CUDA [8] and proposed HYB (hybrid of ELL and COO), that generally adapts well to a broad class of unstructured matrices.

Baskaran and Bordawekar [6] optimized CSR so that it performs about the same (modestly faster with more memory cost) as CSR-Vector [8] (in which a row is assigned to each warp, which then performs segmented reduction). Choi et al. [12] introduced blocking to CSR and ELL (BCSR and BELLPACK). These representations outperform HYB for matrices with dense block substructure but on average their auto-tuned version is not as good as HYB for general unstructured and sparse matrices.
Liu et al. [50] proposed ELLPACK Sparse Block (ESB) for the Intel Xeon Phi Co-processor, and showed that (on average) ESB can outperform the standard cuSPARSE on NVIDIA K20X.

Yang et al. [88] developed a representation suited for matrices with power-low characteristics, by combining Transposed Jagged Diagonal Storage (TJDS) [22] with COO and blocking. Ashari et al. [1] presented a CSR-based approach (ACSR) crafted for power-law matrices, using binning and dynamic parallelism on Kepler GPUs. Liu et al. [49] recently proposed CSR5 (a CSR-based representation) for SpMV on different target platforms (CPU, GPU and Xeon Phi), which is insensitive to the sparsity structure of the input sparse matrix. Reguly and Giles [59] also presented a CSR-based solution with an auto-tuning algorithm to select the number of rows for a thread to work on based on available resources on the device. Additionally, there have been developments towards having a runtime to select the best-performing representation for a given matrix (e.g. Cocktail Format [74]).

As briefly summarized above, sparse storage representations have been studied extensively for GPUs. Proposed optimizations have followed changes in the GPU architecture. GPU implementation of such representations are significantly impacted by the way nonzeros are distributed across the thread-blocks and threads [8, 29, 7, 12, 87, 88, 1].

While this work was limited to a set of standard representations available in NVIDIA cuSPARSE [14] and CUSP [13], the approach, taken in this study, does not depend on the selected representations. It can be effectively extended to cover additional sparse matrix representations, if needed.
5.7 Conclusion and Discussion

This chapter addressed the problem of automatically selecting the best sparse representation for effective SpMV execution on a GPU, by using input-dependent features on the sparse matrix to be operated on. The extensive characterization is done on the feature distribution of 682 sparse matrices, selected from the UFL repository. This work also analyzed the performance distribution of two popular Sparse libraries from NVIDIA, cuSPARSE and CUSP, on three high-end GPUs, GTX 580, K20c and K40c. The results showed that no representation performs best across this set, and that a one-representation-fits-all approach results in many cases of very poor performance. This study proposed a machine learning approach to automatically predict the best sparse representation, using classification trees. The experiments showed that, using only basic features, predicting the best performing representation (with the proposed approach) is very accurate and only fails for 6 out of 682 matrices. More complex features lead to a 98% efficiency, with only 1 matrix with poor performance using our predictor.
CHAPTER 6

Graph Analytics on GPUs

Vertex-centric programming models [52] have been developed to help users implement graph algorithms in a different (and less difficult) way. In this model, a user-defined program is executed on every vertex of the graph. The user-defined program often requires data from adjacent vertexes or incoming edges, and then produces updates to the outgoing edges, or adjacent vertexes. All the graph vertexes are worked on for a fixed number of iterations or until a convergence condition is met. Examples of vertex-centric models is Google’s Pregel [52] for distributed systems and GraphChi [46] for single-node multi-core CPU.

Graph processing on GPUs has also followed the vertex-centric programming model. Similar to the CPU counterparts, existing GPU programming models [36, 90, 91, 43, 80, 25, 11] employ Bulk-Synchronous Parallelism (BSP) [77], in which data parallelism is implemented in iterations (i.e. "super-steps") of the graph, that are themselves separated by barriers. In each super-step, the algorithm follows Gather-Apply-Scatter (GAS) model [30], where vertexes gather information from neighbors and apply computations. The computation is then followed by a global synchronization to send (i.e. scatter) updated information around and to the neighboring...
vertexes/edges. The vertex-centric programs continue running these super-steps until a termination condition happens (i.e. convergence).

Fast and efficient graph analytics on GPUs is possible when the following challenges are addressed properly:

- **Work distribution**: This is about how to distribute work across GPU threads in order to minimize execution divergence and maximize resource utilization (e.g. memory bandwidth or core clock cycles). Efficient work distribution leads to balanced execution across thread blocks and warps. Existing frameworks assign a vertex to a thread \[32\], one or group of vertexes to a warp \[36\], a group of vertexes to a thread block \[43\], or a vertex to thread/warp/block based on the vertex degree \[53\]. Thus, based on the graph structure (e.g. degree distribution of vertexes), each execution scenario could experience imbalance.

- **Synchronization**: BSP inspired execution of an algorithm requires synchronization and data movement. Even though BSP model is very well suited to the GPU architecture, but the cost of barrier-synchronization between super-steps remains expensive. Such global synchronization events have to be minimized, or be replaced by asynchronous solutions, that are at early stages \[69\].

- **Memory access**: Accessing memory, to read/write information associated to vertexes or edges, is not trivial on GPUs. Depends on the data structure where the graph is stored (e.g. CSR), accessing either incoming or outgoing edges could be sequential, leaving the others to be done in a random fashion. Sequential accesses are very well supported in GPUs due to the coalescing logic that can absorb accesses to consecutive addresses from threads in a block. Random
accesses, however, can greatly reduce bandwidth utilization of global memory and also cause warp-level memory divergence.

In this chapter, the goal is to characterize the design space of an example graph programming model, and to study and model the performance of different graph primitives based on graph features, as well as GPU parameters.

6.1 Methodology

The graphs used in our study are from University of Florida Sparse Matrix Collection [18], Stanford’s SNAP large network data collection [48], and synthetic R-MAT generated graphs [10]. The normal constraints, applied when selecting sparse matrices (namely, C1, C2 and C3 in Section 5.2), have been used to select candidate graphs. In addition, and due to lower arithmetic intensity of some of the graph primitives (e.g. BFS), we tighten the C3 condition to only select graphs that are large enough such that each GPU thread can work on at least one vertex.

6.1.1 Graph Features

Each graph is stored in a file as an un-ordered list of edges where each is a pair \((V_{src}, V_{dst})\), representing a connection from \(V_{src}\) to \(V_{dst}\). In order to build (i.e. load) a graph from the file, all the edges must be processed one-by-one. Table 6.1 lists all the features that are used in this study. Some features can be computed instantly, while others require a full scan over all the edges of the graph. The tables shows that degree distribution and graph density are the two pieces of information obtained from the graph. We distinguish between incoming and outgoing edges because of the way each are stored in memory. When a graph is stored in CSR format, the
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Scan Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>number of vertices</td>
<td>NO</td>
</tr>
<tr>
<td>E</td>
<td>number of edges</td>
<td>NO</td>
</tr>
<tr>
<td>D</td>
<td>graph density ( D = \frac{</td>
<td>E</td>
</tr>
<tr>
<td>min(_{in})</td>
<td>minimum degree of incoming edges</td>
<td>YES</td>
</tr>
<tr>
<td>max(_{in})</td>
<td>maximum degree of incoming edges</td>
<td>YES</td>
</tr>
<tr>
<td>(\mu_{in})</td>
<td>mean degree of incoming edges</td>
<td>YES</td>
</tr>
<tr>
<td>(\sigma_{in})</td>
<td>standard deviation of incoming edges</td>
<td>YES</td>
</tr>
<tr>
<td>min(_{out})</td>
<td>minimum degree of outgoing edges</td>
<td>YES</td>
</tr>
<tr>
<td>max(_{out})</td>
<td>maximum degree of outgoing edges</td>
<td>YES</td>
</tr>
<tr>
<td>(\mu_{out})</td>
<td>mean degree of outgoing edges</td>
<td>YES</td>
</tr>
<tr>
<td>(\sigma_{out})</td>
<td>standard deviation of outgoing edges</td>
<td>YES</td>
</tr>
<tr>
<td>min(_{e})</td>
<td>minimum degree of edges (in or out)</td>
<td>YES</td>
</tr>
<tr>
<td>max(_{e})</td>
<td>maximum degree of edges (in or out)</td>
<td>YES</td>
</tr>
<tr>
<td>(\mu)</td>
<td>mean degree of edges (in or out)</td>
<td>YES</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>standard deviation of edges (in or out)</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 6.1: Graph features and the cost of computing each on \( G = (V, E) \).

outgoing edges are located in consecutive places in memory, while incoming edges will be scattered around (vice versa if stored in CSC). Thus, depends on how graph is stored, and which one dominates for a given graph, a graph primitive could suffer or gain advantage from the data structure and memory layout.

Note that other graph features (e.g. topology-based eccentricity) can also be suitable for analysis and characterization. However, we found the density and degree distribution most relevant to the programming models studied in this chapter (details described in Section 6.1.3).

### 6.1.2 Graph Primitives and GPUs

In this study, three graph primitives are selected and analyzed. These primitives belong to the ”traversal-based” category (as described by Wu et al [86]) where only a subset of vertices (i.e. ”frontier”) are being worked on at any iteration. When such
a primitive finishes execution (i.e. full traversal), each edge has been visited once. Because of this property, we use Million Edges Traversed Per Second (MTEPS) as the performance metric. The selected primitives from this category are: Breath-First Search (BFS), Single-Source Shortest Path (SSSP), and Connectivity Component (CC). An alternative category of primitives, "dense-computation-based" [86], contains algorithms that require most/all the edges to be active at any iteration. Since the number of visits for a given edge depends on the graph structure, only the processing time can be used as the performance metric for this group. PageRank (PR) is a well-known representative of this category. Note that, depends on the implementation, CC could belong to either category. However, in this chapter, only candidates from the first category are chosen to be studied and experimented on further. Also note that an NVIDIA Kepler GPU has been used in this study (K40c, as described in Table 5.2).

6.1.3 Graph Programming Models Overview

For this study, Virtual Warp Centric (VWC) [36], a vertex-centric programming model, has been selected and worked on. Unlike the physical warp which is defined by the SIMD hardware (32, for NVIDIA GPUs), VWC introduces the concept of virtual warp (varies from 2 to 32). When processing a graph algorithm, and in the vertex-centric model, assigning a vertex to a warp could cause inter-warp load imbalance. Meaning, some warps wind up processing more edges than the others, because the assigned vertex has higher incoming or outgoing degree. With a virtual warp, a warp size is selected based on the degree of a vertex, and thus, multiple virtual warps can be run simultaneously on a single physical warp. This helps running more than one
vertex on a single warp; reducing the divergence and thread idle cycles. Thus, the most challenging part of the VWC is to select the best virtual warp size, for a given graph.

We elaborate on performance for different graphs and different GPUs, in the remaining of this chapter. However, "VWC" is also used for further modeling and prediction studies (presented in Section 6.4). As described in this chapter, the methodology of modeling and performance prediction is designed to be generic and applicable to other similar graph primitive systems.

6.2 Feature Analysis

This section focuses, on the graph dataset and elaborates more on the distribution of the features, how different features of the graphs are correlated, and what insights can be obtained and later used for performance modeling and prediction.

6.2.1 Feature Distribution

In general, performance of a graph primitive is closely dependent on the structure of the input graph. In other words, topology of the input graph greatly impacts how fast a graph algorithm runs on a given architecture. Two major aspects of the graph topology, "eccentricity" and "connectivity", mostly represent the structure of the graph, when running a traversal-based primitive (e.g. BFS). When running a graph primitive $X$, eccentricity features (e.g. "radius" or "diameter") determine how many iterations $X$ requires to finish the traversal, while "connectivity" (e.g. vertex degree distribution) relates to the amount of parallelism exists when $X$ is running.

For vertex-centric graph frameworks (including the one studied in this chapter, and explained in Section 6.2), a vertex is considered as a unit of work. Thus, having
a uniform degree distribution across the vertices will significantly help with a balanced execution (i.e. effective parallelism) across the threads (or warps). Due to the selection of vertex-centric graph frameworks, this chapter only focuses on studying the amount of parallelism of a given graph primitive; thus the "eccentricity" features are left for future work.

![Figure 6.1: Visualizing degree distribution of "skewed" vs "non-skewed" graphs](image)

Connectivity features of the graph dataset studied in this chapter (listed in Table 6.1) are selected to mainly quantify degree distribution. For instance, mean number of edges per vertex (µ in Table 6.1) along with the standard deviation (σ) can be used to interpret imbalance execution of an algorithm. The graphs in this chapter can then fall into two categories:

1. **Skewed** graphs with highly-skewed degree distribution (i.e. "power-law"). Examples are social networking graphs with very few high-degree vertices.

2. **Non-skewed** graphs, e.g. road networks, with very low-degree variation across vertices (e.g. normal distribution of vertices based on degree).
<table>
<thead>
<tr>
<th>Group</th>
<th>Source</th>
<th>#Graphs</th>
<th>#Skewed</th>
<th>#Non-Skewed</th>
<th>$CV_{\text{min}}$</th>
<th>$CV_{\text{max}}$</th>
<th>$CV_{\mu}$</th>
<th>$CV_{\sigma}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMAT</td>
<td>RMAT [10]</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>766.2</td>
<td>1146.8</td>
<td>948.2</td>
<td>125.4</td>
</tr>
<tr>
<td>SNAP</td>
<td>SNAP [48]</td>
<td>19</td>
<td>16</td>
<td>3</td>
<td>35.1</td>
<td>2528.6</td>
<td>442.7</td>
<td>623.9</td>
</tr>
<tr>
<td>DIMACS10</td>
<td>UFL [18]</td>
<td>11</td>
<td>5</td>
<td>6</td>
<td>13.3</td>
<td>745.2</td>
<td>256.9</td>
<td>293.5</td>
</tr>
<tr>
<td>FreeScale</td>
<td>UFL [18]</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>486.4</td>
<td>26301.1</td>
<td>11155.3</td>
<td>9994.8</td>
</tr>
<tr>
<td>Gleich</td>
<td>UFL [18]</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>632.5</td>
<td>1018.6</td>
<td>898.2</td>
<td>156.8</td>
</tr>
<tr>
<td>Janna</td>
<td>UFL [18]</td>
<td>13</td>
<td>2</td>
<td>13</td>
<td>8.4</td>
<td>33.6</td>
<td>12.5</td>
<td>7.9</td>
</tr>
<tr>
<td>LAW</td>
<td>UFL [18]</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>269.5</td>
<td>1113.1</td>
<td>639.8</td>
<td>336.9</td>
</tr>
<tr>
<td>Sandia</td>
<td>UFL [18]</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>4384.4</td>
<td>11677.9</td>
<td>7398.6</td>
<td>3807.8</td>
</tr>
<tr>
<td>TSOPF</td>
<td>UFL [18]</td>
<td>17</td>
<td>0</td>
<td>17</td>
<td>166.3</td>
<td>2092.4</td>
<td>620.9</td>
<td>550.3</td>
</tr>
<tr>
<td>Williams</td>
<td>UFL [18]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>743.5</td>
<td>743.5</td>
<td>743.5</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 6.2: Group of studied graphs (Coefficient of Variation: $CV = \frac{s}{\mu} \times 100$).

Figure 6.1 depicts the two distribution types for two example graphs in the dataset (note that incoming and outgoing edges are treated equally, in this case). The "skewed" degree distribution shows the very high variation of degrees across the vertices. Moreover, it shows that there are few high-degree edges that could be the dominant factor in execution time.

To further elaborate on the property of different graphs, Table 6.1 lists all the graphs in 10 different groups (overall 87 graphs). The grouping is done based on the application domain (e.g. UFL graphs [18]), based on the original source of origin (e.g. Stanford’s SNAP large network data collection [48]), or based on ways the graphs are constructed (e.g. R-MAT generated graphs [10]).

In addition to the number of graphs in each group, Table 6.1 also shows how many of the graphs fall into either of the two categories: "skewed" vs "non-skewed". Some groups are skewed-dominant (e.g. UFL_LAW) while others are either all non-skewed (e.g. UFL_TSOPF) or mixed. In order to show variation across graphs in each group, the "Coefficient of Variation" (CV) metric is computed, as: $CV = \frac{s}{\mu} \times 100$. For a given graph, a low value of $CV$ implies that vertex degree is almost constant on the entire graph. However, the higher the $CV$ value is, the more variability has been seen.
for the vertex degree for the graph. The table also shows min, max, mean and std-dev of the CV value itself to show how variable (or similar) graphs in each group are. For instance, even though UFL_LAW is dominated by the "skewed" graphs, but those graphs have very different variations. This is opposed to UFL_Janna, for instance, where the very low CV values shows that graphs inside that group have similar properties. As described in Table 6.1, there is almost same number of "skewed" vs "non-skewed" graphs in the dataset, which is important for the characterization and modeling aspects, as described later in this chapter.

![Figure 6.2: Density $D = \frac{|E|}{|V|(|V|-1)}$ of the studied graphs.](image)

Figure 6.2 shows the density of studied graphs with respect to the number of vertices (i.e. $V$). The distribution shows that the dataset of 87 graphs covers a very broad range of graphs. Those one the left side have small number of vertices, but higher degrees per vertix. Conversely, the graphs on the far right have almost four order of magnitude fewer edges per vertix, but more than three order of magnitudes
more vertices. Such a variety in density shows the good coverage of the entire space and will be important in later analysis and modeling of the performance.

6.2.2 Correlation Analysis

Graph features (listed in Table 6.1) are selected to quantify the degree distribution and connectivity of the given graph. Some of the features may be inherently correlated, and thus will reduce the chance of quality prediction for graph primitives (more to be described in Section 6.4). For instance, even though incoming and outgoing degree of vertices may be independent, for each vertex alone, however those may be closely correlated when considering the graph topology altogether. In order to select subsets of uncorrelated features, further analysis on the feature dependence is required.

First approach taken for the correlation analysis is to measure the strength of association between two ranked variables. For this purpose, “rank-ordering” has been used to measure the possible monotonic relationship between any two separate features. Since two features with such a potential rank-order relationship may play similar roles in the prediction of final performance, (as observed and discussed in Section 6.4), it becomes unnecessary (and inefficient) to keep both in the feature set, when developing prediction models. For total of $n$ samples for every two features $X$ and $Y$, a perfect rank-ordering of $X$ and $Y$ should meet the following condition:

$$\forall x_i \in X, y_i \in Y, i \in [1 : n] : x_i = y_i$$

where $x_i$ and $y_i$ represent rank of $i^{th}$ item of features $X$ and $Y$, respectively. To apply rank-ordering on the set of features in Table 6.1, four following methods have been tested.
**Next-Match** This is an intuitive way of finding rank-ordering relationship between $X$ and $Y$. After sorting the samples based on feature $X$, every value in $Y$ is compared (numerically) against next neighboring value, in order to count total number of ordering violations. The resulting number of violations divided by total number of comparisons (i.e. $n - 1$) is defined as "Next-Match" coefficient, and belong to the range of $[0, 1]$, where 0 implies that $Y$ is also sorted (thus rank-ordered).

**All-Match** This approach is similar to the previous one, except that each value is compared against the remaining points ahead in the list (i.e. distance 1 to $n - i$). As opposed to "Next-Match" which performs $n - 1$ comparison, "All-Match" requires $\frac{1}{2}n(n - 1)$ overall comparisons. The number of violations divided by total number of comparisons is the final computed metric, for this method.

**Spearman** The previous two methods are devised experimentally to measure the potential sorting of the target feature, $Y$. A more statistically-established method is Spearman's $\rho$ coefficient [72] which is computed using Equation 6.1:

$$
\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)} \quad (6.1)
$$

where $d_i = x_i - y_i$ is the difference between ranks. Unlike previous two, Spearman’s method is done without actual sorting of the values. At first, each feature $X$ and $Y$ are rank-ordered separately, with the smallest value getting a value (i.e. rank) of 1. Cases with the same value each receive the average rank they would have received. Next, for each pair of values, the difference between the two ranks (i.e. $d$) is computed.
Square root of differences are summed up to compute the final Spearman’s coefficient, $\rho$.

**Kendall**  This rank ordering is very similar to "All-Match" in that all the possible pairs of values (i.e. $n(n-1)/2$) are considered. Kendall rank correlation [41] measures the strength of dependence between two variables. For two features, $x$ and $y$, the following formula is used to calculate the value of Kendall rank correlation:

\[
\tau = \frac{n_c - n_d}{\frac{1}{2} n(n - 1)}
\]  

(6.2)

where $n_c$ is number of concordant (i.e. ordered in the same way) and $n_d$ is number of discordant (i.e. ordered differently).

**Figure 6.3:** Histogram of four rank-ordering coefficients.

Figure 6.3 shows a histogram of coefficients from the four rank-ordering techniques and the number of pairs (of features) that ended up falling into each of the 10 buckets. The buckets cover the range of [0, 1] for "Next-Match" and "All-Match", and the range of $[-1, 1]$ for "Spearman" and "Kendall". For the first two, the last bucket
Table 6.3: Graph feature subsets after removing the correlated features.

<table>
<thead>
<tr>
<th>Subset</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>$V, E, min_{in}, max_{in}, \mu_{in}, \sigma_{in}, min_{out}, max_{out}, \mu_{out}, \sigma_{out}, min, max, \mu, \sigma, D$</td>
</tr>
<tr>
<td>alive</td>
<td>$min_{in}, max_{in}, \mu_{in}, \sigma_{in}, min_{out}, max_{out}, \mu_{out}, \sigma_{out}, min, max, \mu, \sigma, D$</td>
</tr>
<tr>
<td>rho88</td>
<td>$V, E, max_{out}, \sigma_{out}, max, \mu, \sigma, D$</td>
</tr>
<tr>
<td>rho88ve</td>
<td>$max_{out}, \sigma_{out}, max, \mu, \sigma, D$</td>
</tr>
<tr>
<td>rho88ve</td>
<td>$V, E, max, \mu, \sigma, D$</td>
</tr>
<tr>
<td>rho88ve</td>
<td>$max, \mu, \sigma, D$</td>
</tr>
</tbody>
</table>

((0.9, 1]) contains the number of pairs of features that are strongly correlated (according to those methods), while for the other two methods (Spearman and Kendall), the coefficient range for very strong correlation is $[-1, -0.9]$ and $(0.9, 1]$. So, on the left histogram, no pair of features fall into the last bucket, while for the right histogram, there are several pairs of features that show strong correlation (i.e. fall into first and last buckets).

Spearman’ and Kendall’s coefficients are very similar in terms of distribution of the feature pairs into the right buckets. Since "Next-Match" and "All-Match" both, to some extent, follow the spirit of the other two methods, they are more vulnerable to noises and do not perform very good in rank ordering. Thus, Spearman’s coefficient (i.e. $\rho$) is chosen to further remove correlated features from the prediction models.

In addition to rank-ordering correlation analysis, more statistical correlation may be helpful to find more opportunities for detecting relationships. Although, density features are uncorrelated to degree distribution metrics, but they have correlations internally (e.g. $D = \frac{E}{V(V-1)}$).

In order to construct feature subsets for the later modeling experiments, the original graph feature set (shown in Table 6.1) is divided into six subsets, as shown in Table 6.3. This division is done using both $\rho$ coefficient and the implicit correlation.
between density features. In other words, three thresholds are chosen for $\rho$: $\pm 1.0$, $\pm 0.95$, and $\pm 0.85$. For $\pm 1.0$, the Spearman’s coefficient must be exactly 1.0 or -1.0. However, $\pm 0.85$ for instance represents all features except those whose $\rho$ is greater than 0.85. This implies that, for every feature pair with $|\rho| \geq 0.85$, one of the features is dropped in the subset. In addition to using $\rho$, each of the three versions above has a similar copy subset that only exclude the $V$ and $E$ (e.g. $\rho_{95ve}$ is a copy of $\rho_{95}$ minus the number of vertices and edges).

6.3 Performance Analysis

Before modeling and prediction, this section elaborates on the performance of "VWC" [36], an example graph programming model used in this study. Further descriptions of the VWC system can be found in Section 6.1.

6.3.1 Algorithm Performance

The main run-time parameter for VWC is the virtual warp size, which determines how many vertices an actual warp will be working on. Choosing the size accurately helps with a balanced execution across the warps. Consequently, the choice significantly impacts the kernel execution time. Figure 6.4 shows how often different warp sizes are the best, across different graph primitives.

Two important observations can be made from the histogram in Figure 6.4. First, there does not exist a single warp size that works best for all the four graph primitives. Even though warp size of 32 (i.e. largest) is the best option for almost 50% of the graphs, on all four primitives, but using a fixed size for all can hurt performance significantly. Second, different primitives show variability for a certain warp size. For instance, BFS performs best on 20% of the graphs with warp-size of only 2, while
with the same size, SSSP runs fastest only for 10% of the graphs. Such a variability demands for accurate tuning and modeling.

Table 6.4 shows how often a certain warp size performs the best for all the three algorithms (i.e. BFS, SSSP, CC). For instance, warp-size 4 is never the best for all four algorithms, while it is individually selected as the best 22 times. On the other side, 128 times warp size 32 was selected the best (of total $3 \times 87 = 261$ combinations of algorithms and graphs), out of which, 93.8% of the selections were the same across all the algorithms. In summary, only 72.4% of the time, any of the warp sizes (2, 4, 8, 16 or 32) were the best for all three algorithms.

Table 6.4: How often best warp size is the same for all algorithms.
6.3.2 Analysis of VWC Performance

Another important aspect of performance analysis is to understand the differences between the "best" configuration versus the remaining ones. It only makes tuning and modeling interesting if the distance is significant, between a chosen best vs. a fixed configuration. To shed lights more on this, Table 6.5 shows, for each warp size and graph primitive, how slow the execution time of a kernel gets when a fixed warp size is applied for all the graphs.

<table>
<thead>
<tr>
<th>Warp Size</th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.7x</td>
<td>3.3x</td>
<td>2.9x</td>
</tr>
<tr>
<td>4</td>
<td>2.0x</td>
<td>2.1x</td>
<td>2.2x</td>
</tr>
<tr>
<td>8</td>
<td>1.7x</td>
<td>1.7x</td>
<td>1.8x</td>
</tr>
<tr>
<td>16</td>
<td>1.4x</td>
<td>1.4x</td>
<td>1.5x</td>
</tr>
<tr>
<td>32</td>
<td>1.5x</td>
<td>1.4x</td>
<td>1.4x</td>
</tr>
</tbody>
</table>

Table 6.5: Average (geometric mean) slowdown when a fixed warp size is chosen.

As an example, the SSSP kernel, when always running with warp size of 2, would finish (in average) 3.3 times slower than when it were to run with the best warp. It is also evident that the amount of damage for not selecting the best warp size is different across benchmarks, as well as for different warp sizes. So as far as the kernel execution time is concerned, it is evident that one warp size could not work for all the graphs, and all the algorithms.

In order to further investigate the performance difference between different groups of graphs, Table 6.6 shows, for three algorithms, what is the average slowdown within each graph group (as opposed to Table 6.5 which shows average across all the graphs). There are important observations can be made from the table. First, for a single
graph algorithm, the warp size creating the lowest slowdown is different (e.g. *w*2 for *Janna* of *DIMACS10*, vs *w*32 for others). Also, for a fixed warp size, the average slowdown varies across different graph groups, and this variation is more obvious in the smaller warp sizes. Second, across different algorithms, same group experiences similar ordering among *w*2 to *w*32, except for *Janna*, where *SSSP* shows different slowdowns for various warp sizes. Overall, the table emphasizes on the fact that even within a group of graphs, there is no obvious best warp size which also performs the best for all the algorithms.

<table>
<thead>
<tr>
<th>Group</th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.5x</td>
<td>2x</td>
<td>&gt;2x</td>
</tr>
<tr>
<td>RMAT</td>
<td>3.3</td>
<td>2.3</td>
<td>1.5</td>
</tr>
<tr>
<td>SNAP</td>
<td>1.6</td>
<td>1.4</td>
<td>1.3</td>
</tr>
<tr>
<td>DIMACS10</td>
<td>1.5</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>FreeScale</td>
<td>3.4</td>
<td>3.5</td>
<td>2.2</td>
</tr>
<tr>
<td>Gleich</td>
<td>3.1</td>
<td>2.1</td>
<td>1.6</td>
</tr>
<tr>
<td>Janna</td>
<td>1.2</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>LAW</td>
<td>2.4</td>
<td>1.7</td>
<td>1.3</td>
</tr>
<tr>
<td>Sandia</td>
<td>7.3</td>
<td>4.9</td>
<td>3.1</td>
</tr>
<tr>
<td>TSOPF</td>
<td>8.7</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Williams</td>
<td>3.3</td>
<td>2.2</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 6.6: Average (geometric mean) slowdown across group of studied graphs.
Table 6.7 shows histogram of average slowdown (geometric mean) across all the graphs, all the three primitives, and for all five warp sizes. The histogram shows distribution of slowdowns when a fixed-size warp is chosen for each sample. For example, for BFS, warp size 8 leads to having 28 (of the 87) graphs run more than two times slower than when they run with the best warp size. It is evident (from the table) that choosing smaller warp sizes (2, 4 and 8) leads to slowing down execution of almost half of the graphs (for each primitive) by more than 2 times. Although such a number is lower for the larger warp sizes, but the number of cases with more than 2x slowdown shows the need for accurate warp-size selection for each sample. This is an indicator of the fact that warp size has to be chosen as a function of graph features, and it differs for different given primitives (and architectures). It also proves again that no single warp size can work best for all samples. Although the best warp selection for a group of graphs may be the same, but more analysis and modeling is required to extract the similarities and make the correct selection.

6.4 Automatic Selection of Warp Size

It has become evident, from the previous section, how sensitive execution of a graph primitive on GPU is, to the graph features as well as the GPU architecture. Building on top of the findings in this chapter, and inspired by the automatic selection of sparse matrix representation in Section 5.5, this section expands the study and introduces machine learning techniques for automatic selection of virtual warp size for a given graph.
6.4.1 Learning Problem and Quality Metric

The learning problem in hand is mainly related to the performance of a graph primitive on a GPU. An observation (i.e. sample) is a vector of graph features, $\vec{x}$, that is constructed based on the chosen feature set (described in Table 6.3). As a result of applying different machine learning techniques, a predictor will be constructed that outputs a "predicted warp size" for any given point (i.e. a graph or an observation). The major "quality metric" to evaluate prediction of the developed models are: 1) the performance difference (i.e. average slowdown) between the predicted warp size and the actual best, 2) misprediction rate for a given test set, and 3) distribution of achieved performance (i.e. slowdown) across the test points.

<table>
<thead>
<tr>
<th>Used Warp</th>
<th>BFS</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>w2</td>
<td>1.0</td>
<td>1.6</td>
<td>1.6</td>
<td>2.1</td>
<td>6.3</td>
<td>1.0</td>
<td>2.1</td>
<td>2.3</td>
<td>2.7</td>
<td>6.6</td>
<td>1.0</td>
<td>1.1</td>
<td>1.6</td>
<td>2.3</td>
<td>7.1</td>
</tr>
<tr>
<td>w4</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
<td>1.5</td>
<td>3.9</td>
<td>1.2</td>
<td>1.0</td>
<td>1.2</td>
<td>1.6</td>
<td>4.0</td>
<td>1.1</td>
<td>1.0</td>
<td>1.2</td>
<td>1.6</td>
<td>4.2</td>
</tr>
<tr>
<td>w8</td>
<td>1.5</td>
<td>1.1</td>
<td>1.0</td>
<td>1.2</td>
<td>2.4</td>
<td>1.8</td>
<td>1.1</td>
<td>1.0</td>
<td>1.2</td>
<td>2.5</td>
<td>1.5</td>
<td>1.0</td>
<td>1.2</td>
<td>1.2</td>
<td>2.6</td>
</tr>
<tr>
<td>w16</td>
<td>2.2</td>
<td>1.4</td>
<td>1.1</td>
<td>1.0</td>
<td>1.5</td>
<td>2.9</td>
<td>1.5</td>
<td>1.1</td>
<td>1.0</td>
<td>1.5</td>
<td>2.3</td>
<td>1.4</td>
<td>1.1</td>
<td>1.0</td>
<td>1.5</td>
</tr>
<tr>
<td>w32</td>
<td>3.8</td>
<td>2.1</td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
<td>6.4</td>
<td>2.1</td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
<td>3.8</td>
<td>1.9</td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 6.8: 2D map of relative slowdown and performance differences.

To further emphasize on the importance of "warp size prediction", Table 6.8 shows, for each algorithm, a map of slowdown for a "used warp" versus the "best warp" configuration. For example, $w8$ column on SSSP shows that, among all those graphs whose best warp size was 8, what is the average slowdown relative to when another warp size is "used". For instance, using warp size of 2 brings a lot more slowdown than choosing a warp of size 16, for the same column. This table put emphasis on the possible relation between two warp sizes, when the objective (i.e. "quality metric") is
average slowdown. In a nutshell, worst-case performance loss (i.e. slowdown) appears when an algorithm with small best-warp size is run with large warp sizes, and vice versa. As obvious from the table, the diameter is always one, and the points around the diameter are also close to one; thus proving the previous point about the warp selection.

<table>
<thead>
<tr>
<th>Used Warp</th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>w2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>w4</td>
<td>44</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>w8</td>
<td>43</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>w16</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>w32</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.9: 2D map of distribution (%) of slowdown below 10% of fastest configuration.

The distribution of average slowdown is not uniform, when incorrect warp size is selected. Table 6.9 shows another map for the percentage of points giving below 10% slowdown when warp size is not chosen carefully. For example, in the case of BFS, among all the graphs whose best warp size is 32 (i.e. w32 column), choosing warp size of 2, 4, or 8 will lead to slowdown "higher" than 10% (due to the percentage value of 0). Conversely, BFS on 43% of the graphs with best warp size of 4 (i.e. w4 column) will perform about (or below) 10% of the best when running with warp size of 8 (i.e. the w8 column).

From the two 2D-map tables above, one can conclude that, given best-warp of $W_i$ and used-warp of $W_j$, a low percentage number in $[W_j][W_i]$ (Table 6.9) means that selecting $W_j$ (instead of $W_i$) is more "costly", than for the high percentage values. In other words, low percentage implies that very rarely such a selection performs well
So, replacing a $W_i$ with $W_j$ is only efficient (from performance perspective) if the corresponding percentage value is relatively high. Note that, the same 2D map with a different slowdown threshold (e.g. 20% instead of 10%) will result in increasing some of the percentage values. Overall, by considering such a relations and distribution, one can aim for finding pairs of $(W_i, W_j)$ with their map value close to 100% (ideal) and with a distribution threshold as low as possible. The remaining of this chapter shows how machine learning techniques are used to capture this property and find the "performance efficient" warp alternatives.

### 6.4.2 Experimental Methodology

Two classes of machine learning techniques, namely classification and clustering, are used to help selecting the best warp size. The entire automated process is shown with a diagram in Figure 6.5.

**Figure 6.5:** Training and evaluation process for warp-size selection

The process is done in the following few steps. Some of the parts are done using Weka library [76] (tagged in the diagram as "Weka"), some are implemented as stand-alone tools (tagged as "Script"), and few using both (depends on the learning technique used).
1. **Database Creation**: before the learning process starts, and in order to create the graph database (i.e. "Graph DB"), each graph is run for all the primitives on all the GPU devices. Each run is further processed to create necessary timing stats. The graph is also processed separately to extract features, listed in Table 6.1. An entry in the graph database is created for each graph, along with the timing stats of each primitive and GPU, plus the features. The graph features are analyzed more (e.g. for correlation) to select few representative feature subsets, listed in Table 6.3. The feature subsets are collected in a separate database, "Feature Set DB".

2. **Normalization**: if required by the learning technique, the input samples are normalized in advance. This is mainly done for clustering for which the default normalization (of Weka) is turned off, in order to have the ability to try various normalization techniques. Min-max, Z-Score, and Decimal Scale are of the few normalizations that have been implemented.

3. **Partitioning**: each graph dataset (i.e. "csv" containing 87 graphs, for one GPU and one primitive) is partitioned into two sets, following the 80%-20% rule (similar to Section 5.4). The training is done on the larger set containing 80% of the samples, while the validation is done on the remaining 20%. This process is repeated 5 times, each with another 80-20 split, so that each graph is used for testing exactly once.

4. **Learning**: depends on the technique, this step either perform classification (i.e. to build decision trees) or clustering on the train set, and using the required parameters.
5. **Validation**: the trained model is cross-validated in this phase (using Weka for classification, or Script for clustering) to compute the quality metric and grade the test set.

The two machine learning techniques used in this research require somewhat different pieces, so each is explained separately next.

### 6.4.3 Classification

Similar to the Section 5.4, two classification algorithms are chosen: BFTree (best-first decision tree [24]), and SimpleCart [9] which implements a hierarchical optimal discriminant analysis using minimal cost-complexity pruning [76].

**Algorithm parameters**  Both algorithms take two parameters as argument for the training phase, determining (1) \textit{minNum} the minimal number of instances at the terminal node; and (2) \textit{numFold} the number of folds used in the pruning. All the combinations belong to the Cartesian product of \( A \times B \) are generated, where \( A = \{2, 3, 4, 5\} \), \( B = \{5, 10, 15\} \), \textit{minNum} \( \in A \) and \textit{numFold} \( \in B \) (total of 12 combinations for each tree type, each primitive, and each feature subset).

Table 6.10 presents statistics about the best performing decision trees found, for different graph primitives and feature subsets (listed in Table 6.3), along with the parameters to learn the best tree. The table also reports, for each best trained tree, the obtained misprediction rate as well as the number of mispredicted cases with slowdown higher than 1.5x and 2x. Since the training and testing are done on 5 separate partitioning of the input dataset, the metrics are averaged across the 5 attempts. Below are few important observations on the quality of the classification.
Table 6.10: Performance and quality of trained decision trees.

Observation1  First and foremost, as the main "quality metric", the trained classifier trees have very low average slowdown (between 1.03X to 1.07X). Also, for a given feature subset, average slowdown suggests that BFTree outperforms SimpleCart by some small margin that varies across graph primitives. For instance, best trained BFTree for all feature set can predict a warp size, that if used to run SSSP on "an unseen" graph, the obtained performance will be about 1.03x lower than when the best warp size is used for the same run. The misprediction rate indicates that the
trained predictor’s decision was different between 14% to 30% of times on the test graphs. However, as the 1.5x and 2x columns show, such high misprediction “rarely” caused slowdowns more than 100% (i.e. zero in 2x column).

**Observation2** Second, as different feature sets provide (Table 6.3), sensitivity of different tree types to the removal of more features (moving from the largest set, *all*, to the smallest, *rho85ve*), is different for BFTree vs. SimpleCart, and for different primitives. For instance, judging solely based on average slowdown, the BFTree trained trees with less features almost performs best for all primitives, even though the difference is very marginal. Also, graph dimension information (*V* and *E*) appear to be not important, as *all* vs *allve* shows a very small difference.

**Observation3** Another important message to take from the classification results is that the tree parameters are different for most of the best trained trees. For example, for a fixed feature set and tree type (e.g. *allve* and *BFTree*), the choice of best parameters is different between graph primitives (more so about the *numFold* than *minNum*).

In summary, it appears that BFTree outperforms SimpleCart (even marginal) for all the case on the total 87 graphs, and for all three graph primitives. Moreover, even with rather high misprediction rate, the created predictor (i.e. decision tree) is able to limit the number of “very bad choice” (i.e. slowdown above 1.5x) to a very few (or none). The created trees have very small sizes, but the learning parameters are different (across primitives and feature sets). Finally, choice of feature subset is very dependent on the primitive, but the study clearly shows that having more features seem to be not effective.
6.4.4 Clustering

As an alternative learning technique to classification, this section explains how clustering can be used to automatically select (i.e. predict) the best warp size. Assuming data points scattered in an k-D space (where k is the number of features or attributes), a clustering technique tries to discover whether individual points fall into different groups, using the point distribution and coordination. In this study, and using the Weka library [76], KMeans clustering is implemented, with the number of clusters as an input parameter. For a dataset of N observations (i.e. points), KMeans can partition the space into C clusters such that each point in the space belongs to the cluster with the nearest distance (e.g. Euclidean).

In this study, set of features is selected from the ones listed in Table 6.3) plus the "best warp size". In other words, each point is tagged with its obtained "best warp size" (as a numeric value) to help the tool finding similarities between the points. Overall, a set of 5 warp sizes have been studied (2, 4, 8, 16 and 32).

Due to the difference in the range and the nature of values of different features, and in order to make the data appropriate for mining, it is required to normalize (and re-scale) the values a priori. However, the best normalization method depends on the data to be normalized. For an input dataset of $X = x_1, ..., x_n$, the following normalization techniques have been implemented for this study:

1. **Min-Max (MM)**: this is the classic technique that uses the range of values to linearly translate each into a number between 0 and 1, as follows:

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

where $z_i$ is the $i^{th}$ normalized value.
2. **Z-Score (ZS):** by measuring the number of standard deviations a point has
gone above the "mean" value, this approach scores the points by subtracting the
population mean from an individual raw value and then dividing the difference
by the standard deviation, as follows:

\[
z_i = \frac{x_i - \mu}{\sigma}
\]

3. **Decimal Scaling (DS):** for a given feature \( Y \), this method normalizes values
by moving the decimal point, based on the maximum absolute value of the
population of \( Y \). A new normalized value \( z_i \) is obtained using:

\[
z_i = \frac{x_i}{10^c}
\]

where \( c \) is the smallest integer such that \( max(|z_i|) < 1 \).

**Methodology**  In addition to the three normalization methods, the **No Normal-
ization (NN)** is also added to the mix to be used as a baseline for comparison.
Similar to the classification study, 5 different 80%-20% partitioning of input points
are used to learn and cross-validate the clustering. The number of cluster (\( NC \)) is
an input parameter and belongs to the range of 2 to 20. When \( c \) clusters are created
from the training set, the cross-validation is done by computing the Euclidean dis-
tance of a given test point to all the \( c \) "centroids" and finding the closest one. As a
result, a test point \( x \) is assigned to a cluster and the warp size which is "the best"
for the majority of that cluster is chosen to be the "predicted" warp size for \( x \). The
difference between "used" and "actual best" warp sizes shows the performance loss
(i.e. slowdown).
Table 6.11: Performance and quality of KMeans clustering.

Table 6.11 presents the obtained results for clustering on three algorithms, using 6 different feature sets, and 4 different normalizations (i.e. MM, ZS, DS and NN). Few observations can be made regarding the quality of KMeans clustering for warp size prediction. First and foremost, the average slowdown appears to be much higher than what is achieved by the alternative technique; classification (presented in Table 6.10). The normalization is rarely effective, as most of the cases, NN (i.e. No Normalization) is the choice. However, for cases where it is effective, the best choice is different across the algorithms (for a given feature set). Also, the best clustering is obtained by different number of clusters (i.e. NC) for each algorithm and under different feature sets. In average, CC requires the least number of clusters for different feature sets. Note that, since 5 total warp sizes exist in this study, an ideal (oracle) clustering would be 5 separate clusters where the best warp size for all the points inside each cluster is the same. However, as apparent from the table, this is not the case. In addition, the high misprediction rate indicates the poor quality of clustering achieved.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Feat Set</th>
<th>Slowdown(%)</th>
<th>Norm</th>
<th>NC</th>
<th>Mispred (%)</th>
<th>&gt;1.5x</th>
<th>&gt;2x</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>all</td>
<td>31.3</td>
<td>NN</td>
<td>3</td>
<td>93.0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>alive</td>
<td>30.1</td>
<td>NN</td>
<td>11</td>
<td>62.9</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho95</td>
<td>31.2</td>
<td>ZS</td>
<td>6</td>
<td>55.8</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho95ve</td>
<td>26.2</td>
<td>NN</td>
<td>8</td>
<td>70.0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho85</td>
<td>31.2</td>
<td>ZS</td>
<td>4</td>
<td>54.7</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho85ve</td>
<td>32.2</td>
<td>MM</td>
<td>13</td>
<td>50.0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>SSSP</td>
<td>all</td>
<td>29.6</td>
<td>NN</td>
<td>5</td>
<td>90.7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>alive</td>
<td>25.6</td>
<td>NN</td>
<td>2</td>
<td>82.7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho95</td>
<td>29.5</td>
<td>MM</td>
<td>6</td>
<td>56.9</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho95ve</td>
<td>22.3</td>
<td>NN</td>
<td>2</td>
<td>78.0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho85</td>
<td>29.6</td>
<td>NN</td>
<td>5</td>
<td>90.7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho85ve</td>
<td>24.4</td>
<td>NN</td>
<td>2</td>
<td>81.6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>all</td>
<td>31.0</td>
<td>ZS</td>
<td>6</td>
<td>59.3</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>alive</td>
<td>29.6</td>
<td>NN</td>
<td>2</td>
<td>88.6</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho95</td>
<td>31.3</td>
<td>ZS</td>
<td>5</td>
<td>59.2</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho95ve</td>
<td>26.8</td>
<td>NN</td>
<td>2</td>
<td>82.8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho85</td>
<td>31.5</td>
<td>NN</td>
<td>2</td>
<td>93.1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>rho85ve</td>
<td>31.2</td>
<td>NN</td>
<td>2</td>
<td>87.5</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
for this dataset. It is also evident that the number of mispredicted cases causing more than 50% slowdown is very high, compared to the classification results.

6.5 Discussion

Based on the findings from Classification and Clustering experiments, one can make an early conclusion that decision trees will be the superior when it comes to predicting the warp size. However, there are some facts about the clustering that could very well affect the quality, and thus performance, of grouping points.

First and foremost, the dataset contains only 87 graphs and 6 feature sets (out of total \( \sum_{i=2}^{N-1} \binom{N}{i} \) of combinations for \( N = 12 \) distinct features). Such limitation may lead to lower statistical "confidence level". Moreover, the presented results are only for a single GPU, which again limits the credibility of the drawn conclusions. With larger dataset, more GPUs, and a more brute-force selection of features, one can make observations that are more statistically sound.

Regarding the clustering, each test point is cross-validated by computing the Euclidean distance to all the centroids. One drawback to this approach is that outliers in any of the space dimensions can have significant impacts on the distance calculation, and thus cause "mis-placement" of test point in that Euclidean space. Moreover, even when the outliers are removed and excluded from calculations, when the closest cluster is found, the choice of "best warp size" may not be necessarily the one that appears in the majority of population in that cluster. It could rather be a function of different features which itself may require learning. Further investigation in this matter may lead to more complete conclusions.
6.6 Related Work

Parallel graph processing systems have been extensively studied for multi-core CPU as well as distributed systems. Pregel [52] is a framework in distributed setting based on Bulk-Synchronous Parallelism (BSP) model [77]. It splits the computation into “super-steps” at which a vertex can execute a user-defined vertex function and then send results to neighbors (using MPI messages) along graph edges (followed up by synchronization barrier between the super-steps). The GPS [63] and Giraph [27] systems are public source implementations of the Pregel interface with some additional features. GraphLab is a framework for asynchronous parallel graph computations in machine learning. It works in both shared-memory and distributed-memory architectures [51]. It differs from Pregel in that it does not work in bulk-synchronous steps, but rather allows the vertices to be processed asynchronously based on a scheduler. The recent PowerGraph framework combines the shared-memory and asynchronous properties of GraphLab with the associative combining concept of Pregel [52]. GraphChi is a system for handling graph computations using just a PC [46]. It uses a novel parallel sliding windows method for processing graphs from disk. Other examples of parallel shared-memory multicore systems include X-Stream (an edge-centric system) [61], Prism (a deterministic graph processing framework) [39] and Polymer [89].

In addition to optimizing individual graph algorithms on GPUs (e.g. [53]), several (more generic) graph programming models have been developed as well, to support easier algorithm development. TOTEM [26] is a programming model for designing graph algorithms that run on both CPUs and GPUs. Other examples of graph processing systems developed for GPUs are [36, 90, 91, 43, 80, 25, 11].
Different GPU-based graph systems require certain auto-tuning for the input parameters (e.g. kernel geometry in [43] or warp size [36]) in order to achieve reasonable performance for the input graphs, when running different algorithms. This chapter studies the correlation between the parameters and performance for different algorithms, and develops machine learning techniques to predict the best configuration (i.e. warp size). Similar to this study is done in [65, 64] in order to predict the best representation of a sparse matrix when running SpMV on GPUs.

6.7 Conclusion

There has been a huge demand in using GPUs to enable high-speed graph analytics. The programming models, that are mainly "vertex-centric", have shown to be dependent on the features (i.e. topology) of the input graph. When tuning for performance, it is critical to understand such dependence and select the best algorithm parameters for a given graph. This chapter focuses on one of these graph programming models, Virtual Warp Centric (VWC), and characterizes performance bottlenecks when running different graph primitives. The analysis on this chapter showed the sensitivity of VWC (and the virtual warp size parameter) to the input graph and the importance of selecting the correct warp size in order to avoid performance penalties. In addition to such analysis, this chapter also provided machine learning techniques applied on the input dataset in order to predict the best warp size for a given graph. The quality of prediction is evaluated based on the performance difference with respect to the best warp size, and also based on the distribution of slowdown. The applicability of simple machine learning models to help improve
performance and reduce the auto-tuning time of graph algorithms on GPUs was the main objective for this chapter.
CHAPTER 7

Future Work

The foundation built in this research provides the opportunity to further explore and conduct experiments in other related directions. The following are few of the most interesting future directions of research.

- **Cross-Architecture Modeling**: In addition to studying inter-relations between features of a sparse matrix with the representation, the GPU architecture itself has to be involved in the equation. In other words, different GPU architectures have different performance capabilities that directly affect execution of SpMV kernel. For instance, global memory bandwidth is a major key when running any of the memory-bound programs on GPUs, including SpMV. Another key architecture-related metric is the speed and performance of "atomic" instructions on a GPU, as it significantly impacts the time to reduce sum of products for a given row (when a row is split between multiple thread blocks). Thus, as a third part of the sparse triangle, and a future research direction, one can add the GPU parameters in the learning model and measures the prediction rate as well as the quality of model’s output. It will be interesting to find out if a single model can be trained to work properly for more than one GPU (each with different architecture features).
• **Edge-Centric Graph Processing:** Vertex-centric graph processing is the dominant solution to implement graph primitives on GPUs in the current state of the art. However, such a processing model makes the execution time of the graph primitive very dependent on the topology of the input graph. It also means that, depends on the work distribution, in a many-threaded architecture like GPUs, an imbalanced execution can happen, thus reducing the efficiency. One future direction is to design the graph primitives based on the sequence of edges, irrespective of where the edges are connected to. The edge list is streamed in the first step, scatter, where each edge is processed in parallel and an update is generated (conditionally). The stream of updates then is fed to the next step, gather, where the updates are reduced and applied to the destination vertices. Iterating over the edge list for a certain number of times is what will take for the edge-centric algorithm to finish. Such an algorithm seems to be more GPU friendly, and can reduce the data-dependence of the graph primitive significantly. As a future research direction, an evaluation of edge-centric graph processing can be done on different primitives to study the effectiveness and performance of the approach.

• **Energy Modeling and Optimization:** In addition to performance modeling and characterization for data-parallel architectures, it is also interesting to focus on modeling energy. The power and energy modeling can be done at different levels, depends on who uses the model, the accuracy and the modeling time can be different. For the architect or hardware designers, an energy model has to be very accurate and detailed. While for a high-level GPU programmer, who wants to make a decision between two kernel geometries, such low-level details may
not be as necessary. As a future research direction, it will be interesting to see if such a high-level energy modeling is doable for GPUs, either for memory-bound applications or for compute-bound ones.
CHAPTER 8

Conclusion

This research provided insights for optimizing memory-bound programs on data-parallel architectures. The first part addressed a fundamental performance limiting factor with implementation of stencil computations using current short-vector SIMD instruction sets such as SSE — due to the unavoidable overhead of performing multiple loads of data elements from memory or inter-register shuffle operations. An enhanced addressing mode was introduced that allows data elements from two different vector registers to be combined to form operands for vector instructions. A hardware implementation of register files was developed to implement the enhanced addressing mode and a compiler code generation scheme was described for the enhanced vector instruction set architecture. The effectiveness of the new architecture and code generation strategy were demonstrated by using a combination of optimistic and pessimistic emulation on four different x86 CPUs.

Considering another class of memory-bound kernels, sparse matrix-vector multiplication (SpMV), the second part of this research evaluated 27 different sparse matrices on each of the four cuSPARSE schemes, characterizing the SpMV kernel performance in each case. By reasoning on matrix features such as the mean and standard deviation of the number of non-zeros per row, and the matrix density, the
study led to finding a good correlation between matrix density and the best performing representation, on the selected dataset, enabling a priori choice of the best sparse representation for iterative SpMV schemes in many of our test cases. However, few clear outliers were also found for this simple approach, motivating the need for future work to develop more sophisticated models taking into account several matrix features, as well as to cover a wider range of matrices to gain statistical confidence in the classification heuristics.

Focusing more on analysis and characterization of SpMV kernel on GPUs, the next part addressed the problem of automatically selecting the best sparse representation for effective SpMV execution on a GPU, by using input-dependent features on the sparse matrix to be operated on. After an extensive characterization of the feature distribution of the popular UFL repository, 682 matrices were selected that are suitable for GPU acceleration. This work analyzed the performance distribution of two popular sparse libraries from NVIDIA, cuSPARSE and CUSP, on three high-end GPUs, GTX 580, K20c and K40c. The study showed that no representation performs best across this set, and that a one-representation-fits-all approach results in many cases of very poor performance. A machine learning approach was proposed to automatically predict the best sparse representation, using classification trees. Our experiments show that, using only basic features, our approach shows no more than 6 out of 682 matrices to have poor performance using our predictor. More complex features lead to a 98% efficiency, with only 1 matrix with poor performance using our predictor.
The final chapter of this research addressed the high-level graph processing models on GPUs. There are different generic frameworks designed to make easier implementation of graph algorithms on GPUs possible. However, selecting the right set of parameters to help any graph algorithm run fast is challenging. This chapter focused on Virtual Warp Centric (VWC) framework and characterized the performance sensitivity of different traversal-based algorithms to the input graphs. Features are extracted from the graph, and the set of mostly uncorrelated ones are selected. Two machine learning techniques, classification and clustering, were implemented to help predict the best warp size. The experimental results showed that classifiers (i.e. decision trees) are more suitable candidates to develop warp-size predictors, as they outperform the KMeans Clustering with respect to average slowdown, across different graph primitives and feature sets. The chapter also studied the sensitivity of different feature sets as well as performance penalties occurred due to misprediction of different models.
BIBLIOGRAPHY


[13] CUSP. The NVIDIA library of generic parallel algorithms for sparse linear algebra and graph computations on CUDA architecture GPUs.

[14] cuSPARSE. The NVIDIA CUDA sparse matrix library.


145