FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

VALUE SETS COMPUTATION:

CHIPPING AWAY AT THE MEMORY WALL

By

JACOB MCCALL SMITH

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

2026

Copyright © 2026 Jacob McCall Smith. All Rights Reserved.

Jacob McCall Smith defended this thesis on April 22, 2026.
The members of the supervisory committee were:

David Whalley
Professor Directing Thesis

Soner Önder
Committee Member

Grigory Fedyukovich
Committee Member

Xin Yuan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

Foremost, I would like to thank Dr. David Whalley for his unflagging support over the years. I hope that my time working in your lab was as successful and worthwhile for you as it was for me. Thank you for the opportunities, the guidance, and everything you've done.

To Andrew Frey, thank you for your patience and monumental effort through the years. To Roman Kaspar Brunner, thank you for your guidance and clear vision, which were critical to this project's success. To John Dewey, thank you for being my peer, working with you was always a pleasure. To Dr. Soner Önder, thank you for originating this project. This has been one of my greatest prides. To everyone at NTNU, thank you for being such gracious hosts, even during the renovations. I will never forget Trondheim. To the entire MTU team, my admiration for what you do is immense. I'm so thankful that ADL and the simulator are your domain. To all the FSU Compiler lab's denizens, future and past, we have built something great and I look forward to all your coming successes.

This work was supported in part by the US National Science Foundation (NSF) under grants CCF-1900788, OISE-2103103, and CCF-2211354. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and may not reflect the views of the NSF.

TABLE OF CONTENTS

List of Figures vi

List of Abbreviations vii

Abstract viii

1 Introduction **1**

1.1 The Memory Wall 1

1.2 Helping to Breach the Memory Wall 1

1.3 Division of Labor 2

1.4 History and Context 3

1.4.1 IRES in Trondheim 3

1.4.2 Simulator and ADL 3

1.4.3 ASOPT 4

1.5 Code Processing 5

2 Motivation **6**

2.1 Data Level Parallelism 6

2.1.1 Comparison of Localities 6

2.2 Spatial Value Locality 6

2.2.1 SVL In Values 7

2.2.2 SVL In Computation 7

3 Value Sets Computation **10**

3.1 Simple VSC 10

3.2 Memory Perspective 11

3.2.1 Controller Modifications 11

3.2.2 Our Data Structures 11

3.2.3 Forming Sets 12

3.3 Programmer Perspective 13

3.3.1 Application Limitations 13

3.3.2 Enacting the Scheme 14

3.4 Compiler and CPU Perspective 15

3.5 Detailed Look . . . 16
3.5.1 Alignment . . . 17
3.5.2 Detailed Instructions . . . 18
3.6 VSC Integration with DRAM . . . 19
3.7 Banking . . . 19
3.7.1 Bank Level Parallelism . . . 19
3.7.2 Code Example . . . 19
3.8 Prior Work . . . 21
3.8.1 ECC vs ID . . . 21
3.8.2 Pre and Postloop vs Alignment and Masking . . . 21
3.8.3 Automatic vs Intrinsic-based Transformation . . . 21

4 Results **23**
4.1 Experimentation Infrastructure . . . 23
4.2 Statistics . . . 23
4.2.1 Baseline vs VSC vs Banking . . . 23

5 Related and Future Work **27**
5.1 Related Work . . . 27
5.1.1 PIM . . . 27
5.1.2 PNM . . . 27
5.1.3 In-Memory Scatter-Gather . . . 28
5.1.4 Sparsity . . . 28
5.2 Future Work . . . 28
5.2.1 Partial Transformations . . . 28
5.2.2 Cache Implementation . . . 29
5.2.3 Cumulative Operations . . . 29
5.2.4 Commutative Operations . . . 29
5.2.5 Input Reordering . . . 30

6 Conclusions **31**

Bibliography . . . 32

Biographical Sketch . . . 34

LIST OF FIGURES

1.1 Diagram of Vector Scalar Memory Flow. 2

1.2 Source Code and Simulator Workflow 5

2.1 SVL Equation 7

2.2 Histogram showing the redundancy percentage for the MNIST, Fashion MNIST, OCT, and Imagenet datasets. The spatial value locality of each image in the datasets was determined using Figure 2.1. 8

2.3 Access Pattern Comparison 9

3.1 Comparison of Original C and Transformed Code 10

3.2 Entry in the Redundancy Detection Table 12

3.3 Metadata Table Format 13

3.4 Identifying Parameters in MIPS 14

3.5 Intrinsic functions, their associated machine instructions, and descriptions 15

3.6 Matrix Multiply from Livermore Kernels 17

3.7 Banking on Vector Scalar Multiply, Derived from Figure 3.1 20

3.8 Contrasting the Traditional and Transformed Block Code Flow 22

4.1 Processor configuration 24

4.2 Speedup for Vector Scalar, Vector Addition, and Matrix Multiply using the VSC and Banking approaches. 25

4.3 Execution Cycles for varying number of DRAM banks 26

5.1 Example for Commutative Operations 29

LIST OF ABBREVIATIONS

- CAS: Column Access Strobe
- DIMM: Dual Inline Memory Module
- DLP: Data Level Parallelism
- DRAM: Dynamic Random Access Memory
- ECC: Error Correcting Code
- ISA: Instruction Set Architecture
- PIM: Processing In Memory
- PNM: Processing Near Memory
- RAS: Row Access Strobe
- RDT: Redundancy Detection Table
- RISC: Reduced Instruction Set Computer
- SIMD: Single Instruction Multiple Data
- SVL: Spatial Value Locality
- VSC: Value Sets Computation

ABSTRACT

When operating on data structures, it is common for some values to repeat. These repeated values lead to redundant computations— the results have been calculated already, just for a different element of the structure which happens to hold the same value. This thesis introduces Value Sets Computation (VSC), a novel method to avoid both redundant loads and computations and in so doing help to address the Memory Wall. Instead of sequentially iterating through each element, we iterate over each unique value, calculate the result, and distribute it to the corresponding locations in memory where the results for the unique value are stored. VSC predominantly relies on a modified memory controller, which calculates the indices of repeated values and utilizes them in a scatter/gather-like fashion. In this thesis we concentrate on the software support for the VSC programming model. This support has taken a number of different forms over the project's development, from relying completely on automatic compiler transformations, to requiring the programmer to insert calls to intrinsic functions.

CHAPTER 1

INTRODUCTION

Value Sets Computation (VSC) is a novel approach designed to help to overcome the memory wall through a modified DRAM controller working alongside the existing CPU and cache structures. As the disparity between CPU speed and memory latency widens, approaches such as processing in-and-near memory techniques, SIMD (Single Instruction Multiple Data) architectures, and widening the reorder buffer have attempted to bridge this growing gap.

1.1 The Memory Wall

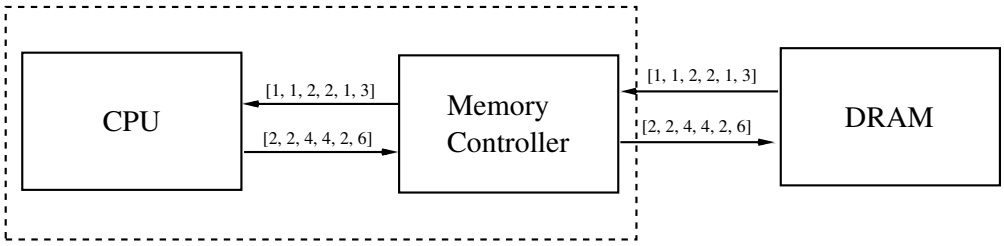
The term was coined in 1994 by a Cornell University and a University of Virginia computer scientist, referring to the growing disparity between processor speed and memory latency [8]. Even then had the trend emerged, though detractors at the time claimed they were making a mountain of a molehill. True to expectations, memory latency has improved since then, but not at the rate CPU speeds have advanced.

In the modern paradigm, programs are either compute-bound or memory-bound, with most falling into the latter camp. Wulf and McKee recognized that the exponential rate of computational advancement we enjoy is due to unforeseen technologies completely changing the landscape, and anticipated the only way to breach the memory wall was to utilize a new approach.

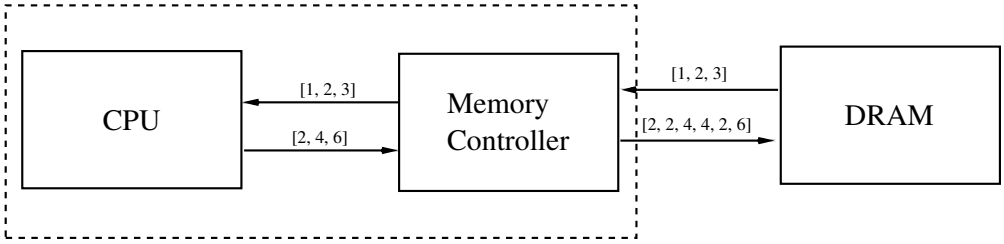
1.2 Helping to Breach the Memory Wall

Perhaps VSC is not *the* solution, but it is a powerful approach when there exists redundant data values. Our approach fits Wulf and McKee's criterion: instead of improving upon existing techniques, it offers a new solution.

Discussed in detail shortly, VSC relies on Data-Level Parallelism (DLP) to improve both bandwidth utilization and compute speed. Instead of sequentially loading each value from memory, our modified memory controller iterates over each unique value. This reduces memory traffic when redundant values are present. The CPU then operates as normal on the unique value and returns the



(a) Baseline data movement



(b) Data movement with VSC

Figure 1.1: Diagram of Vector Scalar Memory Flow.

result to the memory controller, which distributes it to all locations receiving the result associated with the original value.

Figure 1.1 illustrates the reduction in memory traffic: Six values must be loaded from DRAM, six results are computed by the CPU, and six values are stored back to memory for the traditional approach (a). The VSC approach (b) avoids the repeated values, loading only three, computing three results, but writing back the same six to DRAM.

1.3 Division of Labor

Due to the large scale of this project, researchers from Florida State University (FSU), Michigan Technological University (MTU), and Norway University of Science and Technology (NTNU) collaborated to meet the requirements.

Dr. Soner Önder came up with the original idea of detecting redundant values in memory to avoid redundant computations by the CPU. Under his leadership, the MTU team designed much of the architecture to exploit VSC at the DRAM level and built and maintained the simulator through which testing and data gathering was accomplished. Much of the effort was performed by Andrew Frey, who integrated a DRAM simulator within the ISA simulator.

The NTNU team was responsible for some of the architecture design, which took many forms over the project's development. Roman Brunner conceived the notion of *Value Sets* for this approach, enacting the concepts laid by Dr. Önder.

The FSU team maintained the assembly optimizing compiler *ASOPT*, adding functionality to reflect the requisite changes for the project, in addition to developing techniques to easily facilitate the hand-modification of source code in order to exploit VSC. While I was involved with the overall effort of the project, my primary contribution was generating the code required for the paradigm explored in this thesis, as well as a method to exploit bank-level parallelism.

1.4 History and Context

This paper relies on legacy projects from MTU and FSU, and was initially developed under the National Science Foundation's (NSF) International Research Experience for Students (IRES) program. This IRES grant allowed students from FSU and MTU to spend a summer working at NTNU in Trondheim, Norway.

1.4.1 IRES in Trondheim

The foundations of the project were laid in the Summer of 2024 at the NTNU campus. Representatives from both U.S. universities met with their Norwegian counterparts and were able to discuss the project face to face. In the ten week summer program, the shape of the project formed and some preliminary results were gathered.

1.4.2 Simulator and ADL

These results could not have been obtained so quickly without the use of an easily-modifiable architectural simulator. In MTU's system, one specifies micro-architecture features and an instruction set architecture via a specification file written in Architecture Description Language (ADL) [13]. The system then generates an assembler, linker, and disassembler, which are then used alongside source code to create an executable which can be fed into the simulator.

This system allows us to iterate on hardware design and rapidly test software kernels. Results are realistic, as the simulator is quite complete in its scope. Instructions and data values pass through their respective cache hierarchies, so cycle accuracy is maintained. Detailed statistic collection is feasible since a realistic simulation is performed.

ISA. The Instruction Set Architecture (ISA) utilized in this project is MIPS. This ISA is a Reduced Instruction Set Computer (RISC) architecture chosen partially for its simplicity, but primarily due to having been the ISA used in most preceding projects by the team.

Re-targeting the compiler-optimizer and the simulator to another architecture, while feasible, would have required time and effort not available to the team. To make more fair of comparisons, the more modern $RISC-V$ will be the next ISA targeted by our respective projects.

1.4.3 ASOPT

To reflect the code changes required by the VSC approach, we implement a new compiler optimization in the Assembly Optimizer, affectionately called $ASOPT$ or simply OPT. OPT supports both low-level code generation and code-improving transformations [9], allowing the user to specify which optimizations are desired at compile-time.

The OPT system requires some additional information when compiling and optimizing a given program for VSC, where some of the features can be automatically generated, while some features need to be written by the programmer.

RED File. The manually-entered data is minimal, but quite critical to the process. We dub this file the RED file, short for redundant. OPT must be provided with the names of the global structures one wishes to target, along with their total size, element size, as well as the DRAM row size.

Shifting Scope. Back in the days of Trondheim, the intention was for this transformation to almost entirely automatically occur. All the user would need to do is to fill the RED file, as detecting redundant values required inspecting memory values, which was not feasible to do in a single compilation pass.

Early editions of the project were successful in their automation, but as the project scope shifted and the optimized code grew more complex, it became clear that automatic transformation would be prohibitively cumbersome for the resources available to us.

We implement a number of intrinsic function calls that are similar to the avx directives used by approaches like $SIMD$ extensions to convey information to the hardware. These intrinsics are discussed in more detail alongside Figure 3.5.

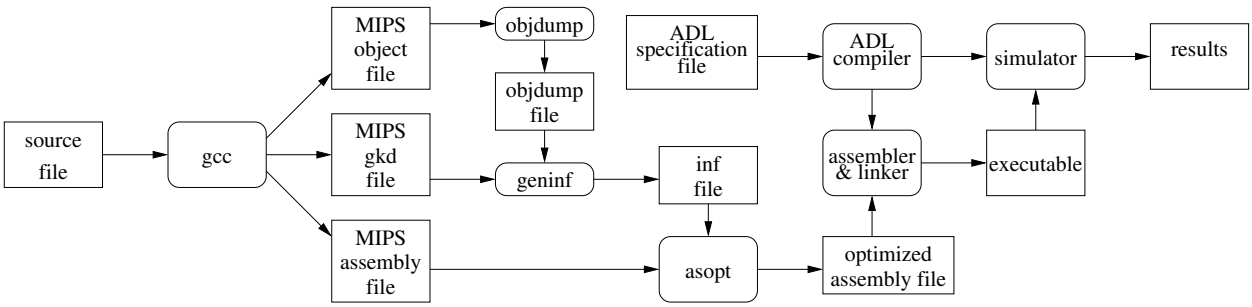


Figure 1.2: Source Code and Simulator Workflow

1.5 Code Processing

To get an overview of the process, our transformation begins with a C source file. This file is first compiled with the commonplace C compiler *gcc* to produce a MIPS object file, a *.gkd* file, and a MIPS assembly file [9]. Rather than performing additional code analysis, we gain the requisite data on functions and register use as a side effect from *gcc*.

The MIPS object file is further refined, gathering information on function return types and the registers to which they are written. The *.gkd* file is used to trace register values being passed to functions. These two files are used to create an inf file, allowing OPT to trace registers used as function parameters and return values without requiring interprocedural analysis.

This is needed to determine which registers are live at each point in a function, so that accurate analysis and code-improving transformations can be performed. With a inf file and the MIPS assembly file, OPT can now transform the code and output optimized assembly code, which is then assembled, linked, and finally, simulated. Figure 1.2 elucidates the process more clearly, showing the entire workflow from a C source file to simulation.

CHAPTER 2

MOTIVATION

This chapter further explores the memory trends that inspired our approach. First we clearly define Data Level Parallelism and Spatial Value Locality, and then explore the concept in practice.

2.1 Data Level Parallelism

Data Level Parallelism (DLP) centers around performing the same instruction on multiple data elements, commonly achieved through $SIMD$ extensions. These $SIMD$ extensions reduce the number of instructions fetched, but do not reduce memory bandwidth requirements.

2.1.1 Comparison of Localities

Just as locality of addresses can be found in the dimensions of time and space, as in Temporal Locality and Spatial Locality respectively, so too does locality exist in terms of data values. To better differentiate Spatial Value Locality (SVL) from similar terms, let us first define SVL's peers.

Temporal Address Locality. Typically called Temporal Locality, this term refers to the tendency for memory addresses that were recently accessed to be accessed again in the near future.

Spatial Address Locality. Also typically referred to by Spatial Locality, this term refers to the tendency for memory addresses nearby to prior accesses to be accessed in the near future.

Temporal Value Locality. This term describes when the same value in a particular memory address is accessed again. This occurs when two loads of the same address are performed without an intervening store [7].

Frequent Value Locality. This term refers to the trend for a small set of distinct values to be loaded from memory repeatedly over execution.

2.2 Spatial Value Locality

To quantify the degree to which we can exploit DLP, we introduce Spatial Value Locality (SVL). If a particular value appears in multiple locations in a data structure, that value exhibits SVL. Figure 2.1 defines SVL clearly.

$$SVL = 1 - \frac{Unique\ Values}{Total\ Values} \tag{2.1}$$

Figure 2.1: SVL Equation

Although similar to other forms of value locality, SVL is unique in that it is defined for a particular region of memory, not the whole memory space. In this thesis, we exploit SVL at the granularity of a DRAM row.

2.2.1 SVL In Values

One of our more surprising findings was the discovery that high levels of redundancy (or SVL) are relatively consistent across standard image processing datasets. All four of the tested datasets exhibited levels over 70%, illustrated in Figure 2.2. To simplify the SVL calculation, each image in the dataset is used as the targeted memory region, meaning that if the value of a pixel repeats anywhere in the image, it is considered redundant.

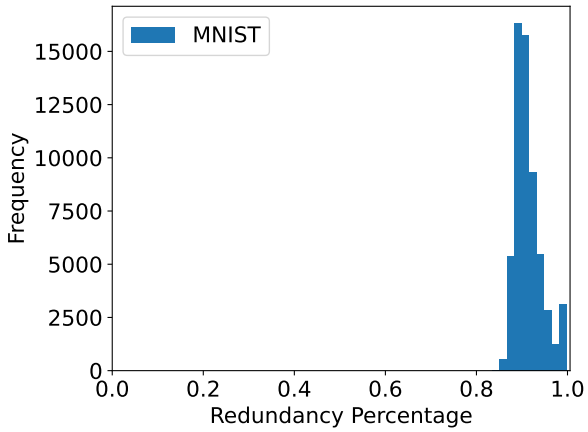
Imagenet (d) is the only dataset containing pixels representing colors, but the dataset still exhibits high levels of redundancy. The grayscale datasets are also surprisingly redundant, thanks in part to the pigeonhole principle– that with a limited range (grayscale colors), the domain (each pixel of the image) is guaranteed to have redundant values. With an average level of redundancy so high, it suggests ample opportunity for benefits if this approach were to have widespread adoption.

2.2.2 SVL In Computation

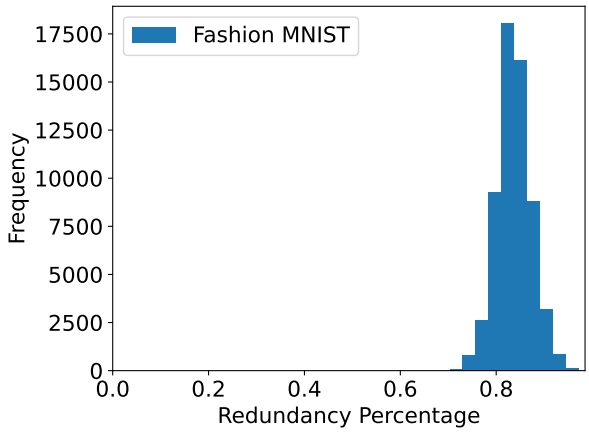
Consider a vector-scalar multiplication, and the number of steps required to complete the task. For a vector of length N, it requires N loads from memory, N computations, and N stores back to memory.

Figure 2.3 illustrates a small example of the operation, and disregarding the minimal overhead of the VSC approach, reduces the number of required computations by 70%. Also consider the reduction in memory traffic; Where the traditional approach loads a value for each computation (though the cost is diffused by caching), the VSC approach potentially avoids the loading cost entirely for any repeated values.

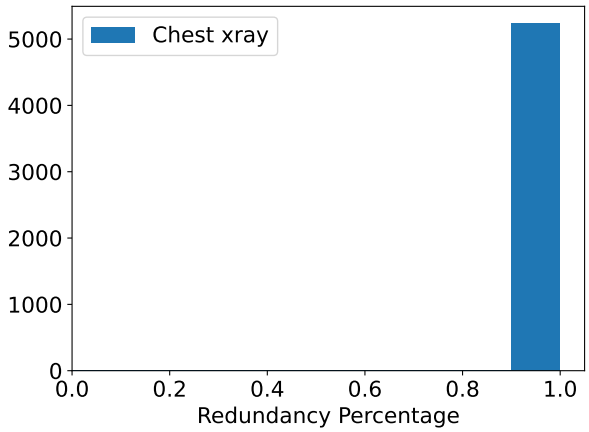
The benefit here should be clear, especially for more costly computations. Although much of our benefit is gleaned from avoiding memory latency, avoiding computations requiring multiple cycles or steps is extremely beneficial to our approach.



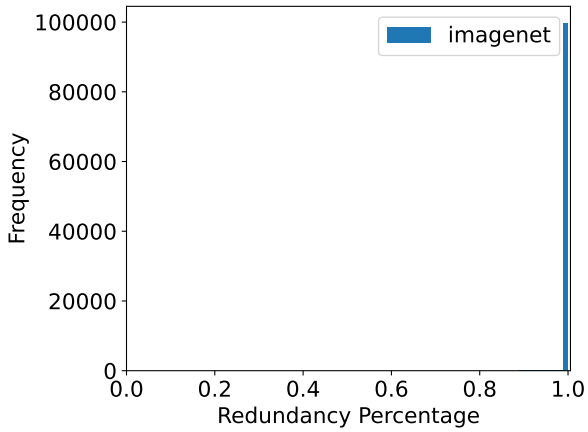
(a) MNIST [2]



(b) Fashion MNIST [15]



(c) Labeled Optical Coherence Tomography (OCT) [5]



(d) Imagenet 1k [12]

Figure 2.2: Histogram showing the redundancy percentage for the MNIST, Fashion MNIST, OCT, and Imagenet datasets. The spatial value locality of each image in the datasets was determined using Figure 2.1.

```
Original Loop
X = [1, 2, 3, 1, 3, 2, 1, 1, 3, 2]
Y = 5
for (i = 0; i < 10; i++)
   Z[i] = X[i] * Y;
```

Original Pattern	VSC Pattern
C1: 1*5	C1: 1*5
C2: 2*5	C2: 2*5
C3: 3*5	C3: 3*5
C4: 1*5	
C5: 3*5	
C6: 2*5	
C7: 1*5	
C8: 1*5	
C9: 3*5	
C10: 2*5	

Figure 2.3: Access Pattern Comparison

CHAPTER 3

VALUE SETS COMPUTATION

Having now covered the motivation for the VSC approach, we may discuss in greater detail just how we exploit SVL. First, we give a brief overview of our simplest transformation. Next, we provide a description from the memory controller's perspective, followed by how the programmer may enact our scheme. Finally, we cover the perspective of the CPU, and take a detailed look at the more complicated transformations that may occur. Lastly, we summarize a brief overview of prior efforts for this project.

3.1 Simple VSC

Of our three test cases, the simplest is by far is the vector scalar multiply. Figure 3.1 shows the original C code followed by the resulting modified source code. The function calls will be defined further in Figure 3.5.

```
1  --------------DEFINITIONS---------------
2  int LENGTH;       // The number of elements per-array.
3  int DRAM_ROW_LEN; // The size of a DRAM row in bytes.
4  int scalar;       // Some scalar value.
5  int A[], B[];     // Source and Destination array.
6  ----------------Original----------------
7  for (int i=0; i<LENGTH; i++)
8      B[i] = A[i] * scalar;
9  ----------------Modified----------------
10 const int num_words = DRAM_ROW_LEN/sizeof(int);
11 int* EOV_A = &A + LENGTH;
12 VSC_flush(&A);
13 for (int i = 0; i < LENGTH; i += num_words) {
14     VSC_mask(&A+i, (EOV_A - (&A+i)));
15     VSC_copy_sets(&B+i, &A+i);
16     int count = VSC_get_cardinality(&A+i);
17     for (int k = 0; k < count; k++) {
18         int tmp = VSC_get_value(k, &A+i+k) * scalar;
19         VSC_broadcast(tmp, k, &B+i+k);
20     }
21     VSC_clear(&B+i);
22 }
```

Figure 3.1: Comparison of Original C and Transformed Code

Instead of sequentially handling each value from A, we strip-mine the vector into chunks of *num_words* elements. To rephrase for clarity, we instead iterate over each DRAM row's worth of unique values. On line 14, we mask the current row for a width equal to the end of the source vector minus the current position. If this value is less than a DRAM row, the values outside the structure are not included in subsequent VSC instructions. On line 15, we copy the locations in memory associated with the Value Sets from A to B, offset by the number of DRAM rows we have handled so far. On line 16, we fetch the number of unique values in the current source DRAM row. Using a simple count of how many values have been handled, we request the k'th Value Set from the i'th row on line 18 and perform the scalar multiplication.

Finally, we broadcast the result to all locations in B that are associated with the current value. On line 21, we clear the metadata to ensure correctness on the next iteration. Although the source code appears more complex (not to mention the resulting assembly), once the instructions become familiar the idea is easy to follow.

3.2 Memory Perspective

The memory controller takes most of the responsibility for enacting the VSC approach, as our project centers around the modified memory access patterns previously mentioned. Detecting the redundant values in memory, forming the Value Sets, fetching each unique value and writing the results back to all associated locations are the responsibility of the memory controller.

3.2.1 Controller Modifications

Despite all these changes, the modifications to reflect these requirements are quite realistic in scope. To perform the VSC approach, an additional chip must be added to the Dual Inline Memory Module (DIMM) in order to store our sideband data. Due to the addition of a chip, the memory bus needs to be widened by one bit, but this cost is minimal compared to the benefit.

3.2.2 Our Data Structures

As there exists many pieces of data that must be maintained to perform the VSC approach, we introduce a select few structures to store, detect, and efficiently access that information.

Location Table. The first hurdle is the detection of repeating values. One could naively compare each element against all the others in the region for $O(n^2)$ total comparisons. Instead, we hash it to index into a table of previously-seen values. The Location Table maintains a set of all locations

in the memory region associated with a value. The index in this table is the set ID used in the Redundancy Detection Table.

Redundancy Detection Table. The Redundancy Detection Table (RDT), as the name implies, is used to organize repeated values as they are being written to DRAM. Our approach requires only one RDT, irrespective of the number of sources being used.

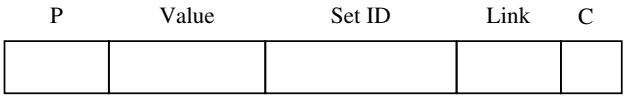


Figure 3.2: Entry in the Redundancy Detection Table

Before being written to the DRAM row, every data word is first entered into the RDT. The format is visualized in Figure 3.2. Each entry contains a present bit, the value in question, the set ID, a link field, and a collision bit.

Notably, the RDT is not indexed by the set ID, instead by a hash of the value to make checking if it is already present faster. If a value has a hash collision with another entry, the collision bit and link field are employed to handle the requisite chaining. The present bit is used to determine whether the value being written is already in the RDT, or if a new entry must be used.

Metadata Table. We also maintain a group of three vectors dubbed the Metadata Table, for each source in the targeted computation. As shown in Figure 3.3, each metadata table consists of a distribution vector, mapping vector, and a broadcast vector. The distribution vector holds the set ID for each location in the DRAM row, as dictated by the RDT. The mapping vector, also named quite aptly, maintains a mapping between the set IDs used by the CPU and those used by the memory controller. The broadcast vector tracks which locations in the DRAM row have already been written during the execution of a broadcast operation.

In addition to these vectors, each Metadata Table also has a present bit, the address tag, the number of Value Sets present in the DRAM row, and a bit determining if the structure is the destination.

3.2.3 Forming Sets

Consider when three words are being written with the values of 6, 7, and 6. Assume an empty RDT and Metadata Tables. The first 6 will be hashed to index into the RDT, the present bit will be found blank, and therefore 6 will be written into the value field, the present bit will be set, and

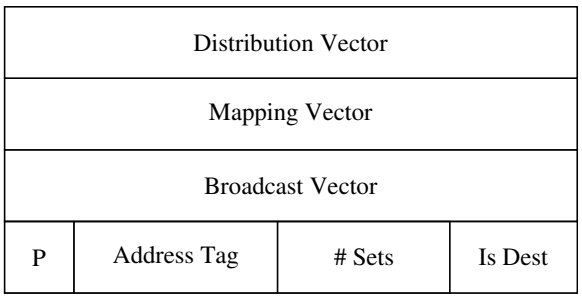


Figure 3.3: Metadata Table Format

the set ID will be assigned. In the Metadata Tables, the first index of the distribution vector will be assigned the set ID, and the mapping vector will associate the memory controller's set ID with the CPU's ID of 0.

There are two set IDs in use, one for the memory controller, and one for the CPU. While the memory controller calculates the set ID, the CPU treats them as a sequential count for ease of access. The aforementioned mapping vector from the Metadata Tables links the two. The last of our structures are two simple queues for read and write operations, respectively.

The value 7 will be treated just the same, with the distribution vector's second element holding the set ID, and the mapping vector associating that ID with 1. The second 6 will index into the same entry as the first 6, and the values will be compared to ensure they match. As they do, and the present bit is already set, nothing else needs to change in the RDT. In the Metadata Tables, the third index of the distribution vector will be assigned the first set ID (the same as in the first index).

3.3 Programmer Perspective

The programmer need not be aware of how the Value Sets are formed. Instead, the programmer needs to only correctly utilize the custom instructions and access pattern required by the VSC approach.

3.3.1 Application Limitations

Due to our modified access pattern, there are some code characteristics that must be in place for the VSC approach to be applicable.

1. Unit Stride: As the Value Sets are formed on contiguous memory regions, the source code must use a unit stride.

2. Independent Iterations: We are unable to transform regions of code that perform recurrence operations. For example, loop bodies of the form A[i] = A[i-1]*s cannot be transformed, as we lose the positional information required to get correct results.

3. No Side Effects: VSC removes redundant computations. If the removed computations produced side effects, we are unable to reproduce them.

3.3.2 Enacting the Scheme

As shown in Figure 3.1, we introduce a number of intrinsic function calls to signal to *ASOPT* key information. Due to the function calling conventions, these calls convey not only the structure of the targeted code, but the registers which will be read or written.

```
1 ----------------C----------------
2 val1 = VSC_get_value(id, b)
3 ---------------MIPS--------------
4 mov $4,$14
5 mov $5,$16
6 jal VSC_get_value
7 mov $8,$2
8 -----------Transformed-----------
9 VSC_get_value $2,$4,$5
```

Figure 3.4: Identifying Parameters in MIPS

Figure 3.4 shows the multiple steps of the transformation, from the intrinsic function calls to the generated assembly instructions, consisting of move instructions and a call to the function. These moves are unnecessary, as is the jump-and-link. The *jal* signals to *ASOPT* the location in code where we wish to insert a VSC instruction, and the *mov*s signal the relevant registers.
Figure 3.5 enumerates our additions to the instruction set. As they are machine instructions, all of the parameters beginning with $ refer to registers. In the actual assembly, they would be numbered,

not named.

- Intrinsic Function: **int VSC_get_value(int id, void* src)**
  Machine Instruction: **VSC_gv $id, $src**
  Description: Retrieves the value for the set identified by *id*. *src* contains the address of the DRAM row it will be read from.

- Intrinsic Function: **void VSC_broadcast(int val, int id, void* dst)**
  Machine Instruction: **VSC_bcst $val, $id, $dst**
  Description: Distributes the result in *val* to the destinations identified by ID in the *dst* address.

- Intrinsic Function: **void VSC_copy_set(void* dst, void* src)**
  Machine Instruction: **VSC_cpsi $dst, $src**
  Description: Copies Value Sets information associated with the address in *src* to the address in *dst*.

- Intrinsic Function: **void VSC_copy_sets(void* dst, void* src1, void* src2)**
  Machine Instruction: **VSC_cpsim $dst, $src1, $src2**
  Description: Forms the concatenated sets between DRAM rows identified by *src*1 and *src*2 and copies them to *dst*.

- Intrinsic Function: **int VSC_get_cardinality(void* src)**
  Machine Instruction: **VSC_gcardin $src**
  Description: Retrieves the number of unique values in the DRAM row specified by the address in *src*.

- Intrinsic Function: **void VSC_clear(void* src)**
  Machine Instruction: **VSC_clear $src**
  Description: Clears the Metadata Table associated with the address in *src*.

- Intrinsic Function: **void VSC_flush(void* dst)**
  Machine Instruction: **VSC_flush $dst**
  Description: Flushes the values associated with *dst* from the cache to ensure the DRAM values are not stale.

Figure 3.5: Intrinsic functions, their associated machine instructions, and descriptions

3.4 Compiler and CPU Perspective

Unlike approaches such as Processing in Memory or Processing Near Memory, one of VSC's greatest strengths is that computation is still done in the CPU. No additional hardware logic is required in

memory, beside the minor memory controller modifications. This makes VSC significantly easier to adapt into existing systems, as it can still benefit from novel CPU approaches.

The compiler needs only to search for the forenamed intrinsic function calls and replace them with the corresponding machine instructions. This is a far cry from *ASOPT*'s previous responsibilities, explored further in Section 3.8. As the transformation grew more complicated, especially with the introduction of exploiting DRAM bank-level parallelism, the effort required to automatically transform code outweighed our available resources.

3.5 Detailed Look

We now have the terminology and context to cover a more complicated case. It is known that matrix multiplication is a common operation, and a critical one at that. Figure 3.6 is derived from the Livermore Loops benchmark suite, and displays the greatest performance improvement of our test cases.

The innermost loop indexed by j is the one we target. Like in Figure 3.1, the loop stride is changed to be the width of a DRAM row, and we insert an additional loop inside to iterate over each unique value in the row. The transformation is nearly identical otherwise, fetching the value, calculating the result, and distributing it to all associated locations in the row.

```
--------------DEFINITIONS---------------
int LENGTH;         // The number of elements per 1D array.
int DRAM_ROW_LEN;   // The size of a DRAM row in bytes.
int scalar;         // Some scalar value.
int C[], B[], A[]; // Destination and Source array(s).
----------------Original----------------
int i, k, j, scalar;
for (i = 0 ; i < LENGTH; i++)
  for (k = 0; k < LENGTH; k++) {
    scalar = A[i][k];
    for (j = 0; j < LENGTH; j++)
      C[i][j] = C[i][j] + (B[k][j] * scalar);
  }
----------------Modified----------------
int i, k, j, x, scalar;
const int num_words = DRAM_ROW_LEN/sizeof(int);
int* EOV_B = &B + (LENGTH * LENGTH);
for (i = 0; i < LENGTH; i++) {
  for (k = 0; k < LENGTH; k++) {
    VSC_flush(&B[k]);
    VSC_flush(&C[i]);
    scalar = A[i][k];
    for(j = 0; j < LENGTH; j += num_words) {
      VSC_mask(&B[k]+j, (EOV_B - &B[k]+j));
      VSC_copy_sets(&C[i]+j, &C[i]+j, &B[k]+j);
      int count = VSC_get_cardinality(&B[k]+j);
      for (x = 0; x < count; x++) {
        int val1 = VSC_get_value(x, &B[k]+(j+x));
        int val2 = VSC_get_value(x, &C[i]+(j+x));
        int result = val2 + (val1 * scalar);
        VSC_broadcast(result, x, &C[i]+(j+x));
      }
      VSC_clear(&B[k]+j);
      VSC_clear(&C[i]+j);
    }
  }
}
```

Figure 3.6: Matrix Multiply from Livermore Kernels

3.5.1 Alignment

To ensure data structures are aligned with respect both to each other and to the DRAM row boundary, we insert additional directives into the assembly code. These directives force the structure to begin at an integer multiple of the DRAM row width, so that processing a partial DRAM row at the beginning of the data structure is unnecessary.

The *.align* assembly directive is all it takes to force alignment, with the minimal cost of the possible fragmentation of small regions of memory– the space between one structure's end and the row

boundary. This is not a great burden as memory space is plentiful, and often smaller structures can still be placed within the fragment.

3.5.2 Detailed Instructions

Having already discussed the broad strokes of our data management scheme, a more detailed discussion covering the main three machine instructions is now possible.

Copy Sets. – **VSC_cpsi $dst, $src** and **VSC_cpsim $dst, $src1, $src2** To enact this instruction, the memory controller receives the address of the relevant DRAM rows, which include the destination and the source(s). The memory controller then allocates a Metadata Table for each source and the destination, reads in the source set IDs, concatenates the two values if there are two sources, then places that information into both the distribution and broadcast vectors in the associated Metadata Tables.

The concatenation of the source IDs makes them no longer sequential, but as they have been previously written to the mapping vector, the Metadata Table for that row can maintain the connection between the sequential IDs used by the CPU and those stored in the distribution vectors.

Get Value. – **VSC_gv $id, $src** If provided with a sequential set ID and source address, the memory controller can query the Metadata Table's mapping vector associated with that address for the memory-side ID. With that ID, one can then index into the distribution vector and fetch the relevant locations in the DRAM row.

A DRAM read is issued for that particular column of the row, and the value is added into the read buffer for efficient access.

Broadcast. – **VSC_bcst $val, $id, $dst** Similar to the **VSC_gv** instruction, the broadcast operation first queries the Metadata Tables to recover the associated memory locations for a given ID. As a single broadcast may not be enough to fill a DRAM burst, we place them into the write buffer until enough accumulate.

Once the value is placed into the buffer, the memory controller marks the locations in the broadcast vector. Once every position has been marked, meaning each result for that row has been written, that Metadata Table's purpose has been fulfilled and can be repurposed for the next row.

3.6 VSC Integration with DRAM

To avoid having to modify existing cache structures, while still taking advantage of temporal reference locality, we store location sets in a special buffer. More specifically, we only store the addresses, which can be queried with the mapping vector from the Metadata Tables to rediscover the set ID. We use a write-through, no write-allocate policy for accessing DRAM when performing the VSC calls, as we must maintain the metadata until the distribution of sets is complete.

3.7 Banking

To glean yet more performance, we realized there was an aspect of memory we could further exploit. Our approach relies on Spatial Value Locality to discover new sources of parallelism, but there exists another source of parallelism we had yet to exploit.

3.7.1 Bank Level Parallelism

VSC improves memory utilization by avoiding redundant values within a row, but much time is still spent waiting for the memory operations to occur. Accessing a block in DRAM requires a Row Access Strobe (RAS) to load the block into a buffer, followed by a Column Access Strobe (CAS) to access a portion of that buffered row. Much time is spent loading these DRAM row buffers. We posit that if each memory bank worked on an independent portion of the targeted data structures, the filling of the DRAM row buffers could be performed in parallel and the required runtime would be reduced even further.

3.7.2 Code Example

We will now discuss a code example using this approach. The vector scalar multiply (Figure 3.7) is the simplest to follow. The code differences between the VSC and banking approaches are minimal compared to the differences between the baseline and VSC approaches.

The key difference here is the interleaving of RAS requests. To accomplish this, we mask, copy the set IDs, and fetch the first value from each bank in a preliminary pass then handle all the values and broadcast them afterwards.

```
--------------DEFINITIONS---------------
int LENGTH;         // The number of elements per-array.
int DRAM_ROW_LEN;   // The size of a DRAM row in bytes.
int scalar;         // Some scalar value.
int A[], B[];       // Source and Destination array.
int NUM_MEM_BANKS; // The number of memory banks in use.
----------------Original----------------
for (int i=0; i<LENGTH; i++)
    B[i] = A[i] * scalar;
----------Vector Scalar Banking---------
int main() {
  VSC_flush(&A);
  VSC_flush(&B);
  // iterate over the whole data structure in strides
  int stride = DRAM_ROW_LEN * NUM_MEM_BANKS;
  int *src_addr, *end_addr, *dst_addr;
  for (src_addr = &A, end_addr = &A + LENGTH, dst_addr = &B;
    src_addr < end_addr;
    src_addr += stride, dst_addr += stride)
  {
    int *tmp_src, *tmp_dst, val, lim, i, j;
    // prefetch the row
    for (i = 0, tmp_src = src_addr, tmp_dst = dst_addr;
      i < NUM_MEM_BANKS;
      i++, tmp_src += DRAM_ROW_LEN, tmp_dst += DRAM_ROW_LEN)
    {
      VSC_mask(tmp_src, (end_addr - tmp_src));
      VSC_copy_sets(tmp_dst, tmp_src);
      val = VSC_get_value(0, tmp_src);
    }
    // iterate over each bank
    for (i = 0, tmp_src = src_addr, tmp_dst = dst_addr;
      i < NUM_MEM_BANKS;
      i++, tmp_src += DRAM_ROW_LEN, tmp_dst += DRAM_ROW_LEN)
    {
      for (j = 0, lim = VSC_get_cardinality(tmp_src);
        j < lim;
        j++)
      {
        val = VSC_get_value(j, tmp_src + j);
        // perform calculation
        val *= scalar;
        VSC_broadcast(val, j, tmp_dst + j);
      }
      VSC_clear(tmp_src);
    }
  } // end of strided loop
} // end of main
```

Figure 3.7: Banking on Vector Scalar Multiply, Derived from Figure 3.1

3.8 Prior Work

The VSC project, being so novel in its approach, was burdened by multiple reworks over its development.

3.8.1 ECC vs ID

Previous versions of the project attempted to utilize the Error Correcting Code (ECC) as an ID, as it is automatically generated for all values loaded into memory in a typical architecture. While trying this approach, we found that too often two distinct values were being mapped to the same ECC. Additionally, we found that replacing the ECC chip's responsibilities could lead to data mismatch, as the ECC is used to determine if the value is correct or has been corrupted in some way.

3.8.2 Pre and Postloop vs Alignment and Masking

To handle the common occurrence of a source vector not being aligned in memory with respect to a DRAM row, the form of our transformed code was once much larger. For values in memory preceding the DRAM row boundary, we would insert a traditional loop to iterate until the addresses aligned. The VSC approach could only then be applied. To avoid this overhead, we aligned the data structures as specified in the *.RED* file.

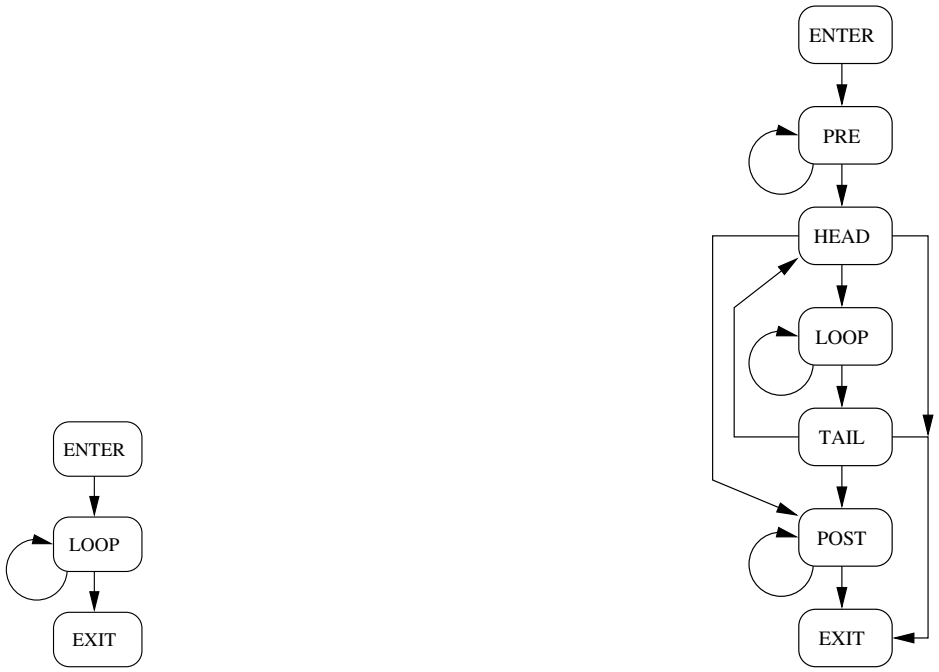
An additional loop was required after the VSC transformation to handle values extending into a partial row. To get around this additional overhead, we introduced the **VSC_mask** operation. Given the current address and an offset in bytes, it blocks other VSC instructions from operating on data past the specified offset.

3.8.3 Automatic vs Intrinsic-based Transformation

Up until the addition of the banking approach, *ASOPT* automatically detected the loops to which VSC could be applied and transformed them. Figure 3.8 shows the resultant blocks of assembly code, including the additional loops for pre and postloop alignment and partial row handling, respectively.

The example illustrated by Figure 3.8 is a simplified one even still. Some assembly blocks have been combined for the sake of the reader's coherence. On the left portion of the diagram, $ENTER$ is the block preceding the targeted loop, $LOOP$ is the loop in question, and $EXIT$ is the block following. This is the traditional approach.

The right portion of the diagram shows the blocks following application of VSC. PRE is the preloop, which sequentially iterates on values until the structure is aligned on a DRAM row boundary or bereft of values. $HEAD$ checks how much data is left to handle. If there is none left it branches to $EXIT$, if there is a partial DRAM row it branches to $POST$, and if there is at least a full row it falls through to $LOOP$. $LOOP$ here is the primary VSC loop similar to what's shown in Figure 3.1. Once complete, it falls through to $TAIL$, which performs a completion check like in $HEAD$. If there's a full row or more it goes back to $HEAD$, if there's no more values it branches to $EXIT$, otherwise it falls through to $POST$. $POST$ is the postloop, sequentially handling any partial DRAM row.



(a) Traditional assembly block code flow

(b) Transformed assembly block code flow

Figure 3.8: Contrasting the Traditional and Transformed Block Code Flow

CHAPTER 4

RESULTS

As mentioned previously, we implement a simulator to gather the performance statistics from the VSC approach. This simulator is an eight-stage, cycle-accurate superscalar system, the details of which are in Figure 4.1.

To simulate memory, we use DRAMSim [6] with the DDR4-8Gb-x8-3200 profile. The non-blocking cache utilizes coalescing MSHRs, and has static access latency. To perform branch prediction, we use Scarab's [10] 64 KiB TAGE-SC-L implementation with the initial prediction provided by a gshare predictor. Lastly, prefetching is accomplished with the IPCP prefetcher [11] for the data cache, and a next-line prefetcher for the instruction cache.

4.1 Experimentation Infrastructure

In order to efficiently simulate different levels of SVL, we modify our test cases to *#include* a C file wherein we define the source and destination structures. This C file, dubbed the VSC_init, is generated by a simple python script. Structuring our testing this way makes changing the levels of SVL easy, as one can simply change the parameters passed to the python script, which in turn automatically generates source structures with the according amount of redundancy, and the corresponding *RED* file.

4.2 Statistics

Our three primary test cases are, in order of complexity, Vector Scalar Multiply, Vector Addition, and Matrix Multiply. There are two versions of each, one with and one without the banking approach. All six versions were run with a varying amount of SVL per DRAM row.

4.2.1 Baseline vs VSC vs Banking

Figure 4.2 illustrates the results, with all three cases meeting or exceeding the baseline with SVL as low as 25%, meaning that every fourth value in the row is repeated. At higher levels of SVL, all cases quickly outpace the baseline in terms of speed. At 99%, we have speedup of 4x, 4.3x, and

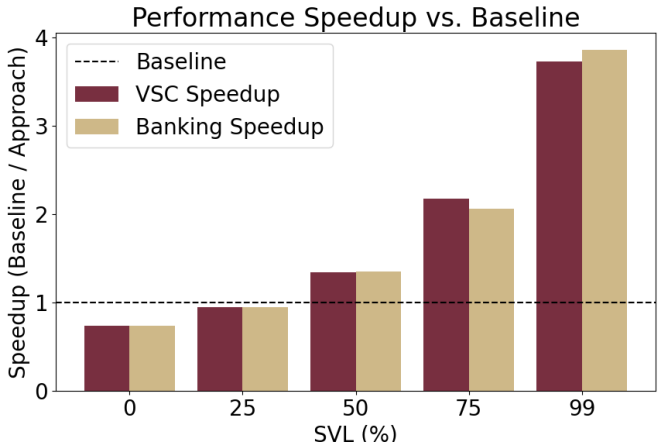
Superscalar Processor		**Cache Hierarchy**	
		Memory Ports (LD/ST)	2/2
Clock Speed	3.0 Ghz	Store Queue Entries	53
Fetch Width	6 Insts.	Block Size	64B
Issue Width	4 Insts.	iCache MSHR Entries	64
Retire Width	4 Insts.	dCache MSHR Entries	64
ROB Entries	320	iCache Prefetcher	Next-Line
		dCache Prefetcher	IPCP
	INT/FP	Level	Size/Assoc/Latency
ADD Units	4/2	L1I	32KB/8/1
MUL/DIV Units	2/1	L1D	64KB/8/4
Window Entries	128/ 64	L2C	256KB/16/12
Physical Registers	256/128	LLC	8MB/16/38
		RAM	DDR4-8Gb-x8-3200
		Meta Data Tables	12
		RDTs	1
		Meta Data Table Size (B)	12x1936
		RDT Size (B)	3264

Figure 4.1: Processor configuration

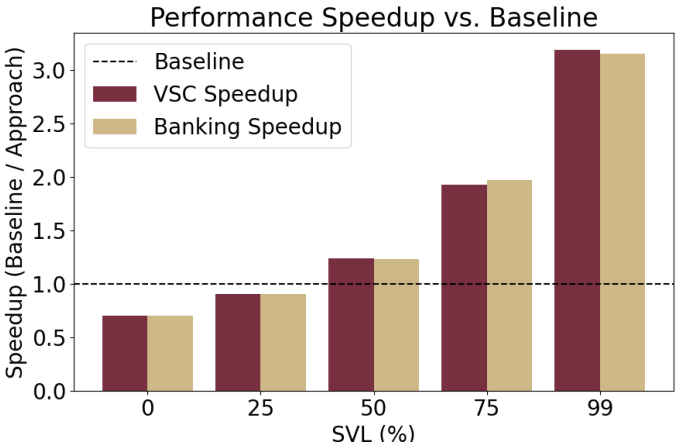
50.7x for Vector Addition, Vector Scalar, and Matrix Multiply respectively when using the VSC approach. The banking approach, here with 4 active DRAM banks shows results nearly-identical to the VSC approach, with what differences do exist likely stemming from noise.

Figure 4.3 is an exploration into the banking approach for the Vector Addition case, showing a negligible difference when employing different numbers of DRAM banks. In theory, the overlapping of Row Access Strobes (RAS) on different DRAM rows should fill the read queue faster, increasing our throughput. We believe there to be intricacies with the simulator which cause the approach to have such unimpressive results.

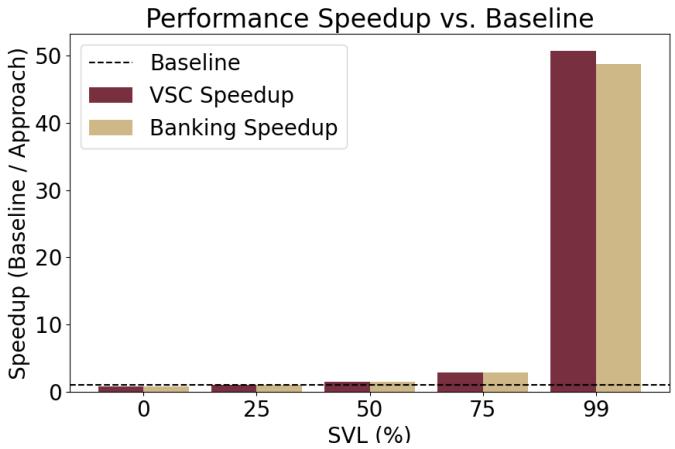
It is extremely important to remember Figure 2.2, specifically how all of the reviewed datasets enjoy an average SVL of 80% or higher. This implies that we can consistently expect the higher end of our statistics to be the case in real-world applications.



(a) Vector Scalar Speedup for VSC and Banking.



(b) Vector Addition Speedup for VSC and Banking



(c) Matrix Multiply Speedup for VSC and Banking

Figure 4.2: Speedup for Vector Scalar, Vector Addition, and Matrix Multiply using the VSC and Banking approaches.

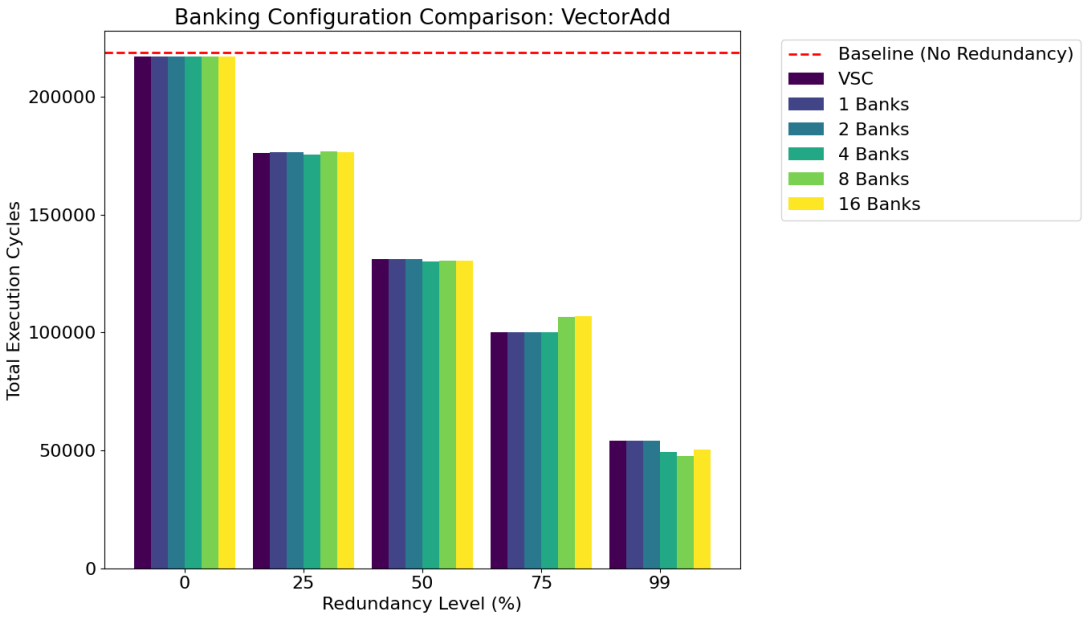


Figure 4.3: Execution Cycles for varying number of DRAM banks

CHAPTER 5

RELATED AND FUTURE WORK

In order to better contextualize VSC, we will discuss some related works. All of these approaches are working towards the same goal– better memory utilization and overall performance. Their methods vary, with varying levels of success. We will follow these related works with an overview on what could next be improved for VSC.

5.1 Related Work

All of the projects below are similar in at least one regard; their motivation. They all center around the limiting factor that memory has become in the modern paradigm. PIM and PNM physically move computation in or near memory, Piccolo and ExTensor explore alternative approaches. All four chip away at the Memory Wall in different ways.

Like PIM, VSC modifies the memory controller. Like PNM, we work on existing hardware with minimal modifications. Piccolo is similar in that it explores methods which enable other approaches. ExTensor is perhaps the most interesting comparison, as it also centers on avoiding redundant values, though their definition of redundancy is quite different to that of VSC.

5.1.1 PIM

The ComputeDRAM project seeks to prove the possibility of in-memory computation using unmodified DRAM modules [3]. To achieve this, they modify the memory controller to enable the violation of timing constraints in order to issue copies and logical AND / OR operations across many DRAM rows at once. This avoids bandwidth limitations by negating the need for data to pass through memory at all, instead processing values in memory.

5.1.2 PNM

Chameleon offers an architecture to achieve processing near memory, avoiding the costly stacking of memory devices typically employed in PIM techniques [1]. The Chameleon approach is derived from Load-Reduced Dual-Inline Memory Module (LRDIMM) architecture, which is distinguished from typical DIMM by the ability to easily increase memory capacity without expensive customization.

By extension, Chameleon can be easily and cheaply modified to match the target application domain.

5.1.3 In-Memory Scatter-Gather

In the domain of graph processing in particular, PIM is not always the ideal approach [14]. Piccolo describes one such flaw; the incompatibility with tiling, an approach commonplace in graph processing accelerators which reduces the effects of the Memory Wall. The paper offers an alternate design of the accelerator, avoiding this incompatibility while maintaining the benefit from both PIM and tiling.

5.1.4 Sparsity

Handling sparsely populated data structures is extremely dependent on the limitations of the memory system [4]. The ExTensor accelerator avoids useless computations, instead operating only on "intersections", or instances of computations where exclusively non-zero elements are in use.

5.2 Future Work

As the VSC project developed, many possible pathways were not explored due to resource limitations. Some of these have certainly been explored before in other paradigms, but applied to VSC have a particularly potent opportunity for benefit.

5.2.1 Partial Transformations

As shown in Section 4, the exploitation of Bank Level Parallelism reaps significant benefits. It stands to reason that even without the entire VSC approach, this could improve performance for traditional systems. We posit that additional functionality could be added to VSC, which would allow for partial transformations of code, depending on the levels of redundancy and details of the computation.

An additional check could be added at the beginning of execution, wherein the total levels of SVL are calculated and compared against some heuristic. If the levels are low, simply exploiting Bank Level Parallelism could be enough, avoiding the overhead of the full VSC transformation. This could even be accomplished at the source code level, branching to one execution path for high SVL and another for low SVL.

5.2.2 Cache Implementation

Our targeted memory region needs not necessarily be DRAM. The VSC approach (with some retooling) could take place almost anywhere in the memory hierarchy, depending on the level of SVL therein.

While accessing the cache is faster than DRAM, the smaller size of the region implies a reduction in opportunity for redundancy, which could negate the access speeds. Further research must be done on the applicability of VSC on different levels of the memory hierarchy, balancing access speeds with opportunity for SVL exploitation.

5.2.3 Cumulative Operations

One major flaw with VSC in its current state is the lack of ability to handle operations involving reductions. For example, if every element of a vector is being summed, VSC cannot be applied. This is because as of now, we do not have an instruction to fetch the frequency at which a given Value Set appears.

Without that required information, the CPU cannot correctly calculate the summation. Additionally, the **VSC_broadcast** instruction would not be usable as we would be writing the results to a single place in memory. Instead, one would have to eschew the scatter-like portion of our approach, relying only on the benefit from not loading redundant values from memory.

5.2.4 Commutative Operations

To uncover more SVL, consider a case where two sources have a commutative operation applied on them. Figure 5.1 enumerates such a case, the operation being addition. In the traditional approach, 10 loads, 10 computations, and 10 stores occur.

```
1  B = [9, 2, 7, 1, 8]
2  C = [1, 8, 3, 9, 2]
3  for (i = 0; i < 5; i++)
4      A[i] = B[i] + C[i]
5  // results in A = [10, 10, 10, 10, 10]
6  ----------------Reorganized----------------
7  B = [1, 2, 3, 1, 2]
8  C = [9, 8, 7, 9, 8]
9  // perform VSC
10 // results in A = [10, 10, 10, 10, 10]
```

Figure 5.1: Example for Commutative Operations

If VSC was applied on the original loop, it would also require 10 of each step. However, if we were able to reorganize the input arrays, we could lower these requirements significantly. After moving the smaller values to one of the sources, VSC needs to only handle three Value Sets.

5.2.5 Input Reordering

Similar to the previous section, this centers on the reordering of input values to extract higher levels of SVL. As established in Figure 2.2, image processing datasets already exhibit high SVL. Consider how the images in question are converted to numeric values.

Are the pixels read in from left to right and top to bottom, or by applying a grid similar to convolutions? Depending on how the data is read in, the same image could have different levels of SVL. We suggest that the pattern with which the image is read can be changed depending on what method extracts the most SVL.

CHAPTER 6

CONCLUSIONS

VSC is a new approach to exploit untapped sources of parallelism in order to chip away at the Memory Wall. This approach has proven successful, but yet has significant room for improvement or modification. Value Sets Computing revolves around the detection of redundant values in memory via a modified memory controller.

Once values are marked as redundant, the application of a new programming model to the relevant source code allows the hardware to iterate over these unique values and more efficiently handle them. Our approach sports both improved speed and bandwidth utilization over the baseline, especially with the introduction of DRAM bank-level parallelism, which could improve the results even further.

The inclusion of VSC or an approach like it is certain to better any adoptive system, especially those laden with large-scale computing tasks. Matrix and vector operations like those found in image processing or AI programs are extremely time consuming, but with VSC can be made quick and easy.

BIBLIOGRAPHY

[1] Hadi Asghari-Moghaddam et al. "Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783753.

[2] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]." In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[3] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 100–113. ISBN: 9781450369381. DOI: 10.1145/3352460.3358260. URL: https://doi.org/10.1145/3352460.3358260.

[4] Kartik Hegde et al. "ExTensor: An Accelerator for Sparse Tensor Algebra." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 319–333. ISBN: 9781450369381. DOI: 10.1145/3352460.3358275. URL: https://doi.org/10.1145/3352460.3358275.

[5] Daniel Kermany, Kang Zhang, and Micheal Goldbaum. *Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification*. Jan. 6, 2018.

[6] Shang Li et al. "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator." In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 106–109. DOI: 10.1109/LCA.2020.2973991.

[7] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. "Value locality and load value prediction." In: *SIGOPS Oper. Syst. Rev.* 30.5 (Sept. 1996), pp. 138–147. ISSN: 0163-5980. DOI: 10.1145/248208.237173. URL: https://doi.org/10.1145/248208.237173.

[8] Sally A. McKee. "Reflections on the memory wall." In: *Proceedings of the 1st Conference on Computing Frontiers*. CF '04. Ischia, Italy: Association for Computing Machinery, 2004, p. 162. ISBN: 1581137419. DOI: 10.1145/977091.977115. URL: https://doi.org/10.1145/977091.977115.

[9] Abigail Mortensen. "Isolating Errors for an Assembly Optimizer." MA thesis. Florida State University, 2022.

[10] Surim Oh et al. "UDP: Utility-Driven Fetch Directed Instruction Prefetching." In: *Proceedings of the International Symposium on Computer Architecture*. June 2024.

[11] Samuel Pakalapati and Biswabandan Panda. "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching." In: *ACM/IEEE Annual International Symposium on Computer Architecture*. 2020, pp. 118–131. DOI: 10.1109/ISCA45697.2020.00021.

[12] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[13] Skylar Scorca. "Implementation of a Poor Man's Trace Cache in an Assembly Optimizer." Florida State University, 2023.

[14] Changmin Shin et al. "Piccolo: Large-Scale Graph Processing with Fine-Grained in-Memory Scatter-Gather." In: *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2025, pp. 641–656. DOI: 10.1109/HPCA61900.2025.00055.

[15] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

BIOGRAPHICAL SKETCH

Jacob McCall Smith first worked as a Computer Science tutor at Tallahassee State College during the pursuit of his AA, then transitioned to working at the FSU Academic Center for Excellence during the first years of his undergraduate degree. In Summer of 2023 he attended a summer class in Aarhus, Denmark taught by the inventor of C++, Bjarne Stroustrup. This class, along with his experiences tutoring at TSC and FSU cemented his burning interest in Computer Science.

Having enrolled in the BA CS/MS CS Pathways program at FSU, he soon took a graduate level computer architecture class with Dr. Whalley and thereafter signed on to his compiler lab. While working in the lab, he implemented the compiler-side requirements for this project under the IRES program, as well as an implementation for eliminating branches through predication, which will be used as a baseline for comparison with other projects in the lab.

He graduated *summa cum laude* with a Bachelors of Arts in Computer Science in Winter of 2024, with his Master's Degree following soon after in Summer of 2026.