SVAR: A Virtual Machine for Portable Code on Reconfigurable Accelerators

SVAR: A Virtual Machine for Portable Code on Reconfigurable Accelerators

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
May, 2023

by

Nathaniel Fredricks

Abstract

The SPAR-2 array processor was designed as an overlay architecture for implementation on Xilinx Field Programmable Gate Arrays (FPGAs). As an overlay, the SPAR-2 array processor can be configured to take advantage of the specific resources available on different FPGAs. However once configured, the SPAR-2 requires programmer's to have knowledge of the low level architecture, and write platform-specific code. In this thesis SVAR, a hardware/software co-designed virtual machine, is proposed that runs on the SPAR-2. SVAR allows programmers to write portable, platform-independent code once and have it interpreted for any specific configuration. Results are presented that verify the virtual machine enables the same code to run without modification on different configurations of the SPAR-2 array running on different FPGA platforms. The results show that the performance cost of this portability is modest, incurring an average 5.6% decrease in performance in partial MLP simulations compared to hand-tuned custom code.

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed Nathaniel Fredricks

Nathaniel Fredricks

Refused _____

Nathaniel Fredricks

ACKNOWLEDGEMENTS

I would like to first thank Dr. Andrews and the Honors College for this thesis opportunity.

I also thank the Department of Computer Science and Engineering for my education.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The emergence of deep learning has propelled a new rise in Artificial Intelligence (AI) [1]. Deep learning is used in areas such speech recognition and computer vision. New advancements in art, speech synthesis, and natural language processing are being pushed by Craiyon, Eleven Labs, and ChatGPT. AI also shows potential in the medical field for uses such as diagnosis [2]. Needless to say, AI is not going away anytime soon.

Traditional compute components, such as CPUs, have proven to be too slow and power inefficient to be suitable for deep learning [3]. According to the Stanford 2019 AI Index Report, the computation growth rate for AI outpaces Moore's Law [4]. CPUs will only continue to fall shorter. In the wake of this, reconfigurable computing has become an appealing option for accelerating deep learning. In the applications of training, FPGAs have been found to offer lower power consumption and lower latency than GPUs while also offering greater flexibility than ASICs [3]. For machine learning inference, research has already been underway with work by those such as Sitao Huang et al [5] and Dong Wang et al [6]. Another area of interest for AI acceleration has been Processing-In-Memory (PIM). Overlaps in PIM and FPGAs have resulted in research such as Compute-in-Memory Blocks for FPGA (CoMeFa) [7]. One particular AI accelerator within this overlap is the second generation SIMD Processing Array, or SPAR-2 [8]. As an FPGA overlay, SPAR-2 was designed so that programmers without hardware design expertise could compile, link, and run software applications on an FPGA.

In terms of resources, not all FPGAs are created equally. Even among Xilinx's Ultrascale+ FPGA offerings, there are huge differences in the number of system logic cells, DSP slices, LUTs, flip-flops, etc. between boards. A hardware design that might fit on one FPGA may not fit on another FPGA with different resources. As an overlay, the SPAR-2 took a step towards enabling portability

across different FPGA architectures; the SPAR-2 design can be easily scaled and reconfigured to fit onto a multitude of FPGAs with different available resources.

Although the SPAR-2 hardware design is very portable, code for it is not. The current SPAR-2 libraries are relatively low-level and, as a result, do not lend well to portability or scaling. Even when working on the same instruction set, the way the programmer writes their application is heavily tied to the specific SPAR-2 configuration. The same operation, such as matrix multiplication, may have to be executed differently across configurations. For example, a program might be able to store a certain matrix in one register on a large configuration of the SPAR-2, but on a smaller configuration that same matrix might have to be split up into multiple registers.

In addition to portability issues, FPGAs have always suffered the problem of accessibility for programmers. Ignoring the problems with running synthesis, coding for designs on FPGAs is not always at an abstraction level high enough for programmers. According to Stack Overflow's 2022 Developer Survey, high-level languages such as Python, JavaScript, and Java were used by many more professional developers than lower-level languages such as C++, C, and Assembly [9]. Although current functions for working with SPAR-2 abstract away some of the intricacies of the hardware, most functions are still closely mapped to SPAR-2's Instruction Set Architecture (ISA). This shares the same problem found with SIMD intrinsics programming: including code so closely tied to hardware with more hardware-independent/abstracted code is, to the programmer, a mismatch similar to embedding assembly code into a high-level language [10]. Many who want to just code at a high level will find programming with the SPAR-2 to be too low-level and tedious.

This thesis presents a solution to the programmers problems: the SIMD Virtual processing Array (SVAR). SVAR is a virtual representation of the SPAR-2 that acts as both a virtual machine and a runtime interpreter. Throughout the remainder of this thesis SVAR will be referred to as a virtual machine. SVAR presents programmers with a new higher-level instruction set architecture (ISA)

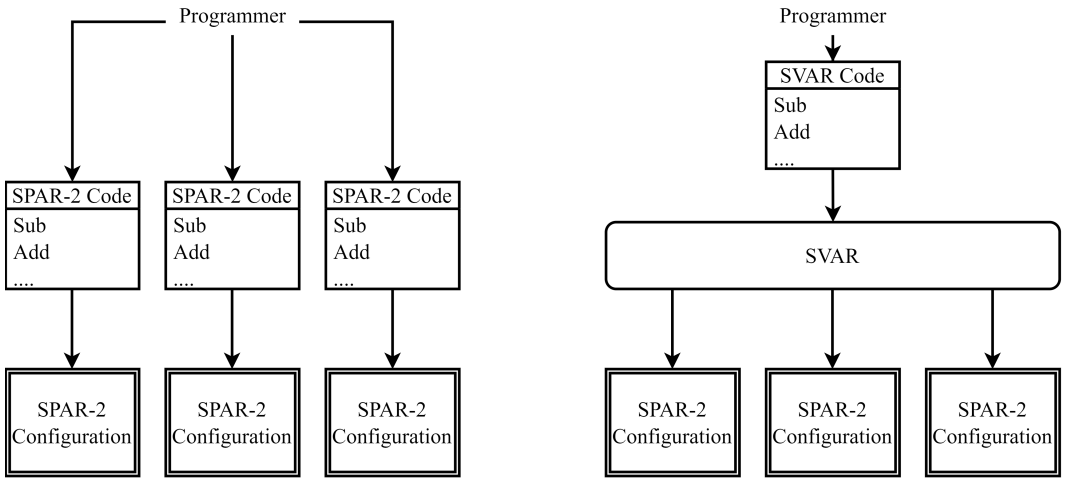

Figure 1.1: Programmers use to have to rewrite code for each configuration of SPAR-2 they wanted to use. SVAR is a virtual machine and run time interpreter that allows programmers to now write and run portable code across multiple SPAR-2 configurations.

based on virtual registers. SVAR code written by the programmer is translated at run time into low-level instructions tuned to a specific SPAR-2 configuration. Thus SVAR 1) provides a virtual platform for code portability across SPAR-2 configurations, 2) abstracts tedious, low-level aspects of SPAR-2 programming away from programmers. Nothing is free. SVAR delivers these benefits at the cost of a modest increase in run time latency.

2 Background

2.1 Related Work

2.1.1 Virtual Machine Technology

Virtual machines are representations of a machine that provide a "virtual environment" that is perceived the same as a "real environment" by application programs [11]. This virtual environment is not required to give an accurate representation of underlying hardware. In fact, the virtual machine can appear to behave differently from the physical machine. One of the most famous examples is the Java Virtual Machine (JVM). The JVM operates on its own instruction set. All Java programs are converted to Java byte code during compilation. This byte code is eventually translated into the host machine's native machine code during run time. This decoupling of compiled byte code from machine code is how Java achieves platform-independence and code portability.

2.1.2 SIMD Portability

There have already been efforts to enable code portability for Single-Instruction-Multiple-Data (SIMD) hardware. Liquid SIMD addressed the problem of hard coded SIMD assembly by compiling to virtual SIMD code and translating that into processor-specific SIMD instructions at run time [12]. The decoupling of written code from hardware instructions allowed portability of code across devices with different SIMD ISAs. Similarly to this, the Generic SIMD Library provided a layer of abstraction from the ISA of its underlying hardware [10]. Both of these offer portability across different ISAs. However, these cases of SIMD instructions do not address portability across reconfigurable or scaling designs such as SPAR-2.

2.1.3 Virtual Registers

As an alternative to stack-based designs for virtual machines, virtual register approaches have been implemented in the past for a variety of reasons such as reducing the number of executed instructions [13]. The use of virtual registers also allowed for the use of register-based code that resembles the machine code of physical hardware. Virtual registers also have their uses outside of the world of virtual machines. In register allocation there is a concept called register pressure: the number of hard registers needed to store values of the pseudo-registers at a given point [14]. Virtual registers are able to aid with dynamic scheduling by reducing register pressure [15]. Reduction in register pressure could allow hardware designs to get away with a smaller register file or increase performance by enabling larger instruction windows on the same-sized register file. Both implementations of virtual registers work by creating virtual representations that behaved similarly to actual physical registers. However, virtual registers may not have to be representative of their physical counterparts. Virtual registers with differing behavior from physical registers could grant additional opportunities for abstraction and allow for easier programming.

2.2 SPAR-2 Architecture Overview

As an array processor, the SPAR-2 performs Single-Instruction-Multiple-Data (SIMD) operations on a square array of processing elements (PEs). SPAR-2 scales by altering the number of PEs in a configuration. Each PE consists of a fixed-point, bit-serial Arithmetic Logic Unit (ALU) attached to local storage modeled as a register file. Instructions are register-based operations that each PE simultaneously executes on the same registers. Since there is an individual instance of each register number on every PE, a physical register could be thought of as a square array rather than just a singular element or word. The size of each register is equal to the number of PEs and thus also tied to scaling of the SPAR-2.

While the general architecture of the SPAR-2 is understandable, there are

still a host of aspects with SPAR-2 programming that are tedious or hinder portability. These will be targets for SVAR to abstract away from the programmer.

2.3 Targets for Virtualization and Abstraction on SPAR-2

2.3.1 Register Sizes and Segmentation

The first opportunity for abstraction is the size of the array processor (number of PEs), which is subject to change with scaling. If a data structure such as a matrix cannot fit within one register, it could be possible to store segments of it across multiple registers. From there the segmented data could be operated on one register at a time. Segmentation across registers may not be a problem with simpler operations such as matrix addition. However, operations that involve data movement such as accumulation are more complicated. Additional data structures would be necessary to track the segments and describe how those segments relate to each other. Custom virtual registers could lend themselves well to these situations by providing a layer of abstraction between the programmer and these aspects of working with the physical registers.

2.3.2 Properties of Data

A vector refers to an ordered list of values and matrices refer to a rectangular layout of values. SPAR-2 operates on a 2D array, which naturally lends itself to matrix operations such as matrix addition and matrix subtraction. SPAR-2 can also be used to perform operations with vectors. As long as the vectors are stored along the same PEs, all the same SIMD operations can be performed on them. The SPAR-2 and its current library have no concept of the data's type, size, or location, so it is usually up to the programmer to adjust the code accordingly. This can become more complicated and tedious if the user wants to perform type-mixing operations such as matrix-vector multiplication. There is potential and incentive to make these adjustments automated.

For many different operations, data has to be oriented in a specific way. For example, matrix-vector multiplication requires that the vector be oriented as a row. The result is a vector oriented as a column. The programmer would usually have to account for this by either reorienting the data or altering the direction of the next set operations. Data orientation should be a target to abstract since it only adds to the complexity while providing no extra functionality.

2.3.3 Multiplication Register Overflow

When multiplying two X-bit numbers, the resulting product is 2X-bit wide. On the SPAR-2, a 32-bit fixed point result is cut from a larger 64-bit product. However, that additional 32-bits is still required for the multiplication operation. These additional bits come from the register immediately above the destination register and will overwrite any existing data. This force programmers to keep track of another register in addition to their sources and destination. Means such as spilling can help address multiplication overflow, but dealing with it is just another inconvenience to the programmer.

2.3.4 Spilling and Register Assignment

Optimally, data should be segmented to fill as few physical registers as possible. As the physical registers get smaller between configurations, the number of segments and thus the number of required registers also increases. However, the number of physical registers does not change between configurations to meet any increases in demand. When the number of registers is insufficient, code can be used to "spill" the contents of the registers to memory in order to reload them to the registers later [16]. The assignment of registers and spilling are subject to change between configurations. Allowing for portability will require automating both of these.

3 Implementation

3.1 Instruction Set

The instruction set was designed as a collection of high level, register based operations. Like with Liquid SIMD, the high level instructions would be translated at run time to the SPAR-2's machine code. The registers in the instruction set no longer refer to physical registers, but rather to virtual registers. Virtual registers are automatically sized and typed to whatever data the programmer stores on them. With virtual register instructions, the programmer will only need to specify the registers and the operation they want (matrix addition, multiply-accumulate, etc.). All details of the operation such as segmentation, orientation, and data movement will be handled in the background. However, the programmer will have a new responsibility of tracking types. SVAR instructions are type specific – matrix addition will require virtual registers with matrices, vector subtraction will require virtual registers with vectors, etc. However, this does not have to be entirely within the programmer's head as the register assignment allows for checking the data type of any virtual register at any time. The necessary instructions for storing and loading data is also included. Store refers to storing data from memory to physical SPAR-2 registers. Load refers to loading data from the registers to memory. While it cannot be directly tested, this instruction set is at a higher level than the existing SPAR-2 libraries and requires much less knowledge of the SPAR-2 hardware to use.

3.2 Register Assignment Table

The register assignment table is responsible for storing all relevant information for the virtual registers. The register assignment table holds the orientation, status, and size of virtual register data. For instruction execution, the register assignment table also tracks what physical registers the virtual register is in. The

necessity of this is the result of having the physical registers dynamically assigned at run time. Dynamic run time assignment was chosen as the use of registers will change from configuration to configuration. The table also contains an additional array to track the status of physical registers and the virtual register to which they are assigned. While it is possible to figure out the status of physical registers by going through the information of every virtual register, it is faster to have a singular, separate reference just for the physical registers. This array is used during register assignment to identify which physical registers are available.

3.3 Virtual Registers

Unlike previous implementations of virtual registers, the SVAR virtual registers are not 1-to-1 representations of physical registers. The virtual registers instead act as more convenient registers that abstract away the size, segmentation, and orientation of data. The size of data is stored for each virtual register. When the data for a virtual register is about to be loaded into the physical registers, the sizes are used to determine the number of segments needed.

Segmenting is also abstracted with the virtual registers. Since the segmenting will change from configuration to configuration, the segments are calculated at run time. To track the segments, every virtual register has an array for storing the placement. Segments are entered into the placement array starting with the top left corner and then moving along the columns and down the rows. An example of this can be seen in Figure 3.1. Because there will be situations where not all virtual register data can fit on the physical SPAR-2 design at once, each virtual register will also have dedicated space in memory to spill into. Each virtual register is also given a status variable to help track whether it is currently in memory or in the SPAR-2 registers.

Orientation is considered in every operation, so each virtual register has a variable to track its current orientation. At run time, SVAR will use the orientation to automatically alter the different operations such as data movement and

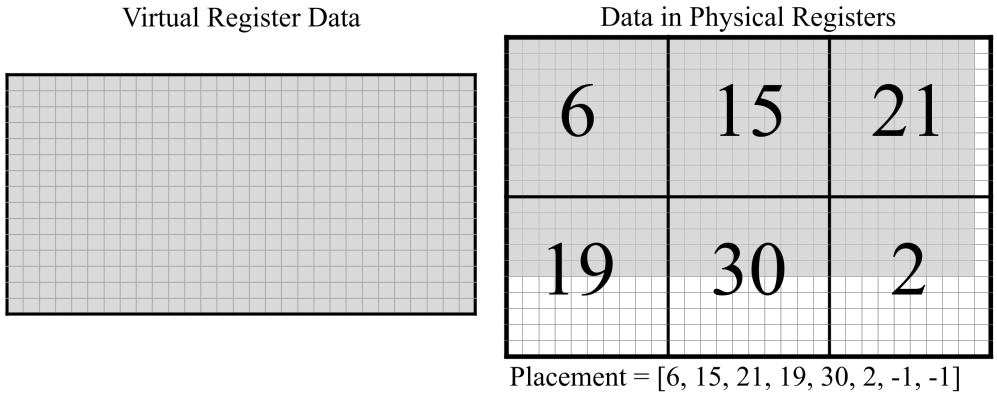

Figure 3.1: To the programmer, a virtual register is the exact dimensions of the data they put in it. However, the data is actually segmented across multiple physical registers at run time. Physical register locations are stored within the placement array for instructions to reference. Unused placements are populated with -1.

loading/storing. All aspects of orientation will be completely abstracted from the programmer.

3.4 Moving Data Between Memory and SPAR

SVAR makes use of a form of lazy loading for virtual register data. Data is only moved into the physical registers immediately before they are needed for an instruction. When the programmer "stores" it into a virtual register, the data is actually copied to a part of the memory dedicated to the virtual register. From here, the data will be moved to the physical registers and back. Saving a reference to the original data would be more efficient than copying it to another part of memory. However, this would be error-prone, as the original data might be altered from when the programmer calls "store" to when the data is actually written into the physical SPAR-2 registers.

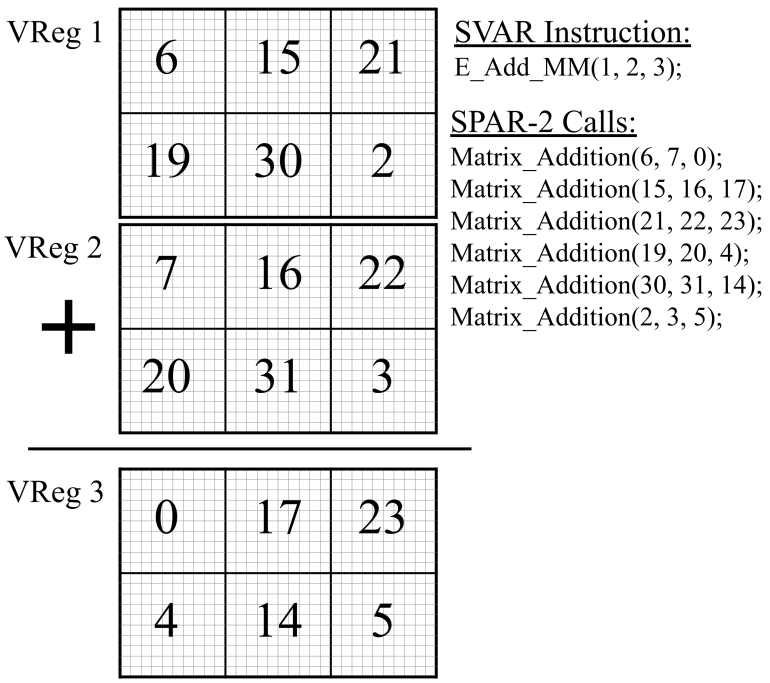

Figure 3.2: Example of matrix addition using virtual registers and data segmentation. The programmer provides the high level instruction. SVAR then executes the given instruction by making multiple lower-level calls to the SPAR-2 for each physical register.

3.5 Operations on Virtual Registers

The SPAR-2 design uses register-based SIMD instructions such as matrix addition, matrix subtraction, and element-wise multiplication. Execution of these element-wise instructions do not require data movement, making them easy to execute with segmented data. Before executing one of these instructions on the SVAR, all virtual register source data is automatically loaded into the assigned SPAR-2 registers. After that, the chosen SPAR-2 instruction is executed on all corresponding segments. An example of matrix addition with virtual registers is shown in Figure 3.2.

3.6 Data Shifting with Virtual Registers

Shifting data between PEs is a necessary component for operations such as accumulation. However, it can be very tedious for the programmer and, as will be explained later, needs to be corrected in cases of segmentation. SPAR-2 has instructions for moving data from a physical register to another in four different directions referred to as north, south, east, and west. Only 1 register can be shifted in a cardinal direction at any given time.

When shifting in any cardinal direction, the opposite edge is populated with 0. If the data is shifted east, for example, then the west-most column will be filled with 0. Data on the edge of the physical register will not shift over to the destination. This is because SPAR-2 does not automatically retain and move data that goes out-of-bounds. To prevent data from being lost, vulnerable edge data between segments must be moved separately. This correction can be seen in Figure 3.3. Combining the SPAR-2's original shifting instruction with edge movement corrections, data will be shifted across the entire segmented virtual register. While it might be possible to use the parallel-serial converters to move data more efficiently, attempts to do so with the current hardware did not work. For now, the edge data is moved by reading and writing data along the edges.

3.7 Handling Multiplication Overflow

Multiplication overflow is also handled by SVAR. There are three register overflow situations that have to be accounted for: overflowing into source registers, overflowing into other destination registers, overflowing into any other registers. If miscellaneous virtual register data (data that is neither part of the source nor destination virtual registers) is at risk of being overwritten by the multiplication overflow, then that data can simply be spilled into memory where it will no longer be at risk. Data for source virtual registers have to be treated differently since the sources have to be in the physical registers for the operations. When a source

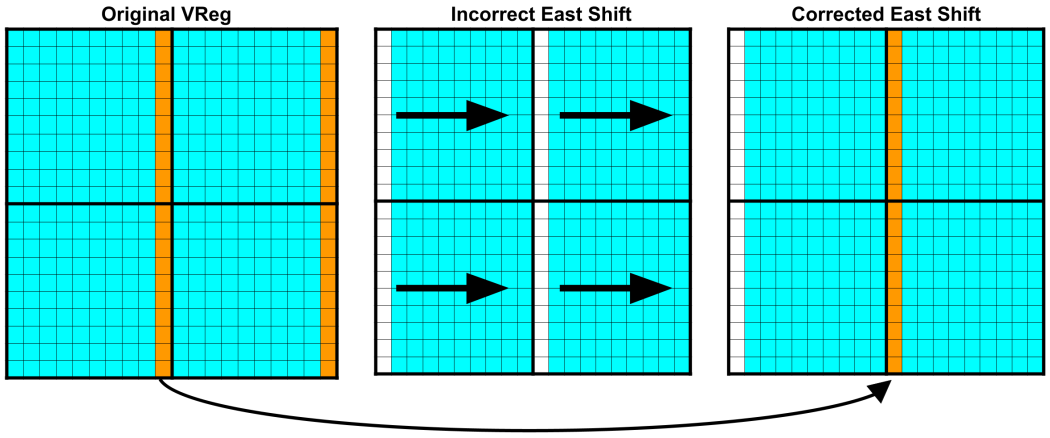

Figure 3.3: The figure above shows an example of shifting data east on virtual registers consisting of 4 physical registers. Calling SPAR-2's shift east instruction can only move data within each physical register. The result is losing the east-most columns of data in each physical register (seen in orange) while generating new columns of 0 (seen in white). This can be corrected by moving select edge columns over to the next corresponding physical register.

register is about to be overwritten by multiplication, SVAR will identify a new safe physical register to place the source data. The source data will then be copied over into that newly assigned register. After that, multiplication can occur as normal without corrupting the source data.

Overflow to other destination registers does not require reassigning any registers. Instead, reordering the multiplication of segments can ensure that no multiplication results will be lost. By executing SPAR-2 multiplication operations in ascending order of the destination segment registers, any overflow data will just be overwritten by the next results. This can be seen in Figure 3.4. Like previous aspects, all multiplication overflow scenarios are automatically handled by SVAR and abstracted from the programmer.

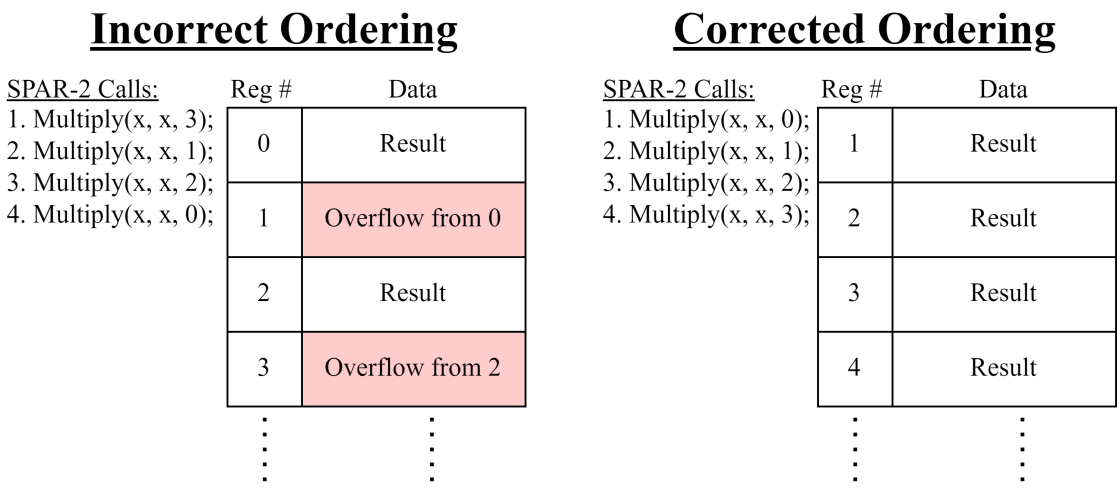

Figure 3.4: When the destination physical registers are close together during multiplication operations, order is very important. If care is not taken, overflow might overwrite some of the previous results as seen on the left side of the figure. By executing multiplication in ascending order of the destination physical registers, results will overwrite the overflow instead.

3.8 ReLU

While the SPAR-2 includes hardware for activation functions such as rectified linear unit (ReLU) and sigmoid, the current SPAR-2 design has errors when executing the activation functions. The activation functions would fail generate the correct outputs for all or even most inputs. This problem probably requires a hardware-level solution, which is beyond the scope of this thesis. For now, ReLU functionality was implemented in C rather than SPAR-2 instructions. For native code, the ReLU instructions would go down either the first row or first column in a register and rewrite all negative values to 0. For SVAR code, the ReLU instruction would move all data out of the SPAR-2 and into the virtual register's assigned memory. From there, any negative values would be replaced with 0.

3.9 Methodology for Testing

3.9.1 Instruction Overhead

To properly test the overhead of SVAR, 1-to-1 comparisons were made with "native" implementations of the same instructions. Native implementations would also make use of data segmentation and generate the same results from the same inputs, the only difference being that they were coded using the pre-existing SPAR-2 library. Multiple variations of data sizes were tested for each instruction. The virtual machine instructions that were tested were matrix addition, matrix subtraction, element-wise multiplication, and matrix-vector multiplication (which includes accumulation). ReLU was not included in this set of tests, as it was unable to be implemented properly with the hardware. Every test would record execution time for the instruction(s) being tested and count of each SPAR-2 instruction call. Call counts help distinguish the overhead due to additional SPAR-2 operations from the overhead due to additional CPU operations.

For element-wise instructions (matrix addition, matrix subtraction, element-wise matrix multiplication), data would first be loaded onto the SPAR-2 registers. Storing the data first would help isolate the execution time of instructions from the data loading time. From there, the operation would be run once to count the number of SPAR-2 instruction calls. Subsequently, the instructions would be timed to run 100 times. The element-wise instructions were too fast to compare accurately in a single iteration, so repetition was added to increase the times to a more comparable scale.

Test for storing and loading were done similarly. When timing, the store/load instructions were not looped. Storing had to be done on the SVAR differently. SVAR usually only writes data to the registers when another instruction is called. To account for this, the function that is automatically called by SVAR was manually called by the tests to move data from the virtual register's memory to the physical registers. To account for variations in time, the tests were run manually 5 times and the averages of the times were recorded. The variances in time between

runs were in the tens of microseconds, so additional runs were not necessary. The call counts did not change between runs.

The matrix-vector multiplication test was performed similarly. It was impossible to isolate the execution time from the storage time when using loops. Matrix-vector multiplication was also much longer than element-wise operations and did not require scaling up the time with multiple iterations. Although there was an implementation of matrix-vector multiplication in the preexisting SPAR-2 library; it was not able to be used for testing. It did not use the corrected data movement across segmented registers; the results for it would be incorrect for all of the tests with more than one segment. As with the store and load instruction tests, the time was averaged across 5 runs of the matrix-vector multiplication tests. Again, the execution time between runs only varied by tens of microseconds, so additional runs were not necessary.

For data segmentation, there are multiple permutations of data dimensions that will result in the same number of data segments. For example, data with 2 segments cannot have the same width and height. The specifics of data segmentation did not affect the performance of the element-wise instructions since those instructions did not involve data shifting on the SPAR-2. However, the specific permutations of data did affect the performance matrix-vector multiplication as the number of shifts necessary for accumulation are directly proportional to the number of columns. To account for this, alternative permutations for 2-segment and 4-segment tests were included for matrix-vector multiplication.

3.9.2 Comparing Portability

Portability was tested by implementing the same set of operations in different SPAR-2 configurations using native code and SVAR code. The different SPAR-2 configurations varied in size, but were all run on a ZCU-104 board. Dimensions for each SPAR-2 configuration were measured in the total number of PEs in a row/column. The chosen set of operations were two sets of matrix-vector

multiplication, vector-vector addition, and ReLU in order to simulate 2 MLP layers. In addition to the SPAR-2 call count and execution time, the number of lines of code was also recorded. The code line count would count all operations would include all lines of code except for comments and white space. The purpose of line count was not to compare the counts of native code to SVAR code, but to help describe the way the code implementations change between configurations. The size of the input vector was set to be 50. The first hidden layer had 75 nodes. The second layer had 50 nodes. Weights and biases were sized accordingly. All inputs, weights, and biases were populated the same between tests and configurations. The resulting vector was used to check that the implementations performed correctly, but that vector was not recorded in the results.

4 Results

1 2 3 4

Table 4.1: Matrix Addition Call Count and Execution Time

Matrix size:	64x64 (1 Segment)		64x128 (2 Segments)		128x128 (4 Segments)		128x256 (8 Segments)	
	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR
Add	1	1	2	2	4	4	8	8
100x Time (s)	**0.00637**	**0.00641**	**0.01275**	**0.0128**	**0.02549**	**0.02556**	**0.05098**	**0.051072**

Table 4.2: Matrix Subtraction Call Count and Execution Time

Matrix Size:	64x64 (1 Segment)		64x128 (2 Segments)		128x128 (4 Segments)		128x256 (8 Segments)	
	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR
Sub	1	1	2	2	4	4	8	8
100x Time (s)	**0.00637**	**0.00641**	**0.01274**	**0.01279**	**0.02549**	**0.02555**	**0.05097**	**0.051064**

Table 4.3: Element-wise Multiplication Call Count and Execution Time

Matrix Size:	64x64 (1 Segment)		64x128 (2 Segments)		128x128 (4 Segments)		128x256 (8 Segments)	
	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR
Mul	1	1	2	2	4	4	8	8
100x Time (s)	**0.00637**	**0.00658**	**0.01274**	**0.01295**	**0.02549**	**0.02571**	**0.05097**	**0.051239**

Table 4.4: Storing Matrix on SPAR-2 from Memory

Matrix Size:	64x64 (1 Segment)		64x128 (2 Segments)		128x128 (4 Segments)		128x256 (8 Segments)	
	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR
Write	4096	4096	8192	8192	16384	16384	32768	32768
1x Time (s)	**0.145513**	**0.29103**	**0.29105**	**0.58207**	**0.58210**	**1.16416**	**1.16424**	**2.32834**

Table 4.5: Loading Matrix to Memory from SPAR-2

Matrix Size:	64x64 (1 Segment)		64x128 (2 Segments)		128x128 (4 Segments)		128x256 (8 Segments)	
	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR
Read	4096	4096	8192	8192	16384	16384	32768	32768
1x Time (s)	**0.098962**	**0.09906**	**0.19782**	**0.19812**	**0.39584**	**0.39625**	**0.78787**	**0.789065**

[1]All recorded SPAR-2 call counts are for one run/iteration only.

[2]Times (unless specified as 100x) are the average execution times for a single iteration run.

[3]All calls not shown in a table have counts of 0.

[4]Size for matrices are number of columns by number of rows

Table 4.6: Matrix-Vector Multiplication

Matrix:	64x64 (1 Segment)		64x128 (2)		128x64 (2)		128x128 (4)		128x256 (8)		256x128 (8)	
Vector:	64 (1 Segment)		64 (1)		128 (2)		128 (2)		128 (2)		256 (4)	
	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR	Native	SVAR
Add	126	126	189	189	380	380	634	634	1142	1142	2292	2292
Sub	1	4	1	7	2	8	2	14	2	26	4	28
Mul	1	1	2	2	2	2	4	4	8	8	8	8
Shift N	0	1	0	2	0	2	0	4	0	8	0	8
Shift S	64	65	64	66	128	130	128	132	128	136	256	264
Shift E	0	0	0	0	0	0	0	0	0	0	0	0
Shift W	63	63	126	126	254	254	508	508	1016	1016	2040	2040
Write	0	0	0	0	8128	8128	16256	16256	32512	32512	97920	97920
Read	0	64	0	64	8128	8256	16256	16384	32512	61312	97920	126848
Time (s)	**0.0163**	**0.0182**	**0.02434**	**0.02663**	**0.5327**	**0.5366**	**1.0414**	**1.4280**	**2.0817**	**2.8621**	**6.1223**	**6.9032**

Table 4.7: 2 MLP Layers

SPAR-2 Size (PEs):	32x32		64x64		96x96	
	Native	SVAR	Native	SVAR	Native	SVAR
Add	898	898	438	438	249	315
Sub	8	39	5	15	3	19
Mul	12	12	4	4	2	2
Shift N	444	8	148	3	74	2
Shift S	64	168	64	195	76	194
Shift E	99	0	130	0	51	0
Shift W	297	738	100	246	50	123
Write	21865	21926	12411	12486	7675	7750
Read	14336	17152	4928	4986	192	250
Time (s)	**1.240655**	**1.383494**	**0.618993**	**0.624799**	**0.310299**	**0.323866**
Code Line Count	153	11	69	11	42	11

5 Evaluation and Discussion

5.1 Analyzing Overhead of Individual Instructions

For matrix addition, matrix subtraction, and element-wise multiplication, SVAR made the exact same SPAR-2 call counts as the native implementations. Matrix addition and subtraction tests averaged a 0.37% increase in execution time from native code to SVAR code. Element-wise multiplication had a higher time overhead averaging a 1.5% time increase with SVAR. The identical call counts suggest that the time overhead is caused by additional work on the CPU's side such as instruction translation, finding placements, etc.

Loading matrix data from the SPAR-2 to memory showed similarly little overhead. SVAR code made the same number of calls to SPAR-2 as the native code. The time only increased by less than 0.15% in all cases. Much greater overhead was seen with storing matrix data from memory to the SPAR-2. Even with the same number of SPAR-2 instruction calls, SVAR always took about twice as long as native code. The most likely cause is how data moves. The native code copied data from memory directly to the SPAR-2 registers. However, SVAR first moved the data to the virtual registers' assigned memory space before it was finally stored on SPAR-2 registers for the operation. This essentially doubled the amount of data movement, resulting in the time doubling. The storing performance is less than ideal and may need to be addressed in the future. Matrix-vector multiplication had overhead from as low as a 0.73% time increase to a 37.49% time increase with the average for all cases falling at 18.25%. In general, the call counts for both the native implementation and the SVAR implementation scaled similarly to the dimensions of the matrix and vector. Generally, SVAR used additional subtraction, shift south, and read calls; some of the overhead was due to the less efficient use of the SPAR-2 instructions. Even with the variations in time increases, the overhead

for matrix-vector multiplication generally remained within an acceptable range.

5.2 Analyzing Cross-Configuration MLP Code

For this set of data, the total number of shift calls should be analyzed since the distribution of shifting differs a lot between the native code and SVAR code. This is caused by the SVAR ReLU function having to remove data from the physical registers. When SVAR stores it back, it is stored in a different orientation. This difference in orientation is subject to change if the ReLU instruction is changed in the future. Accounting for this, the additional calls SVAR makes varies. SVAR can call as many as 68 additional shifts on the 96x96 PE configuration or as few as 2 on the 64x64 PE configuration. Other call counts, such as addition, reading, and subtraction, also varied. The distribution of these additional calls did not necessarily correlate with each other. SVAR overhead was still able to remain modest, only incurring a 5.6% penalty in performance on average.

These losses in performance were made as a trade off for portability of code. Portability can be seen by analyzing how each implementation's code changes across configurations. The number of lines of code were included in the table to help demonstrate these changes. The native code length is inversely related with the configuration size; the smaller the configuration, the larger the code. This is likely caused by increased segmentation; as the configuration size decreases, the same data will have to be split up more and more. As mentioned earlier, SVAR abstracts aspects like segmentation that would require altering code for each configuration, so the resulting SVAR code lengths showed no change at all. In fact, the exact same SVAR code ran on all three configurations and generated the same results with no modifications. SVAR successfully showed its ability to run portable code with different underlying configurations.

5.3 Discussing Code

Qualitative aspects such as ease of programming are difficult to show using quantitative tests. Instead, looking at shorter segments of native code with SVAR equivalents is a good way to illustrate some of these properties. The code examples given in Figure 5.1 will be evaluated.

While the difference in the amount of text on each side is very obvious, it is not necessarily relevant as any programmer can make code briefer with means such as creating additional functions. Instead, what should be analyzed are the more subtle aspects related to how the programmer codes. Starting with element-wise multiplication, there are two aspects that make native code harder to understand: segments and order of multiplication. In native code, the programmer has to keep track of what registers the data segments are, how each of the segments relate to each other, and know that multiplication has to occur in a certain order due to overflow. As explained earlier, these aspects are abstracted by SVAR to relieve the programmer. A programmer using SVAR only needs to know what virtual registers they want to multiply.

Similar aspects are seen with having to manually correct data shifting with column movement. This correction depends entirely on how individual segments relate to each other. In this case, the segment in register 25 is to the "west" of segment 26, and the programmer has to know this in order to make the correction. On the other hand, SVAR's data shift function only requires the programmer to know their source and destination virtual registers. More importantly, the programmer actually does not actually need to use any data shift functions with the current SVAR library; instructions such as accumulate and matrix-vector multiplication already implicitly call these required shift functions. Programmers do have access to data shifting if they want to build more complex instructions in the future.

The last example of code is the accumulation of a matrix. Again, native code requires additional knowledge that the SVAR equivalent does not require: orientation and size of data. The shifting of data westwards is done for the current

Figure 5.1: Examples of Native Code and SVAR Code Equivalents

	Native Code	SVAR Equivalent
Element-wise Multiplication	ELEMENTWISE_MULTIPLICATION(1, 9, 17); ELEMENTWISE_MULTIPLICATION(2, 10, 18); ELEMENTWISE_MULTIPLICATION(3, 11, 19); ELEMENTWISE_MULTIPLICATION(4, 12, 20);	E_Mul_MM(1,2,3, table);
Shift Data West	SHIFT_WEST(17, 25); WEST_COLUMN_MOVE(18, 25); SHIFT_WEST(18, 26);	ShiftWest_M(2, 3, table);

Accumulate	SHIFT_WEST(17, 25);
	MATRIX_ADDITION(17, 25, 17);
	for(int x=0; x<colN-2; x++)
	{
	SHIFT_WEST(25, 25);
	MATRIX_ADDITION(17, 25, 17);
	}
		AccColumns_M(4, 5, table);

orientation of the matrix that it is accumulating. The size of data also determines how many times the loop occurs. Even if accumulation was turned into a function with orientation and data size as parameters, it would not change the fact that the programmer would still have to keep track of those properties. This example of native code also does not show the effects of segmented data, which would further compound the complexity of accumulation in native code. One aspect of SVAR that might be inconvenient is having to pass a reference to the current register assignment table into every function, but this can be altered in the future by using a global instance of the table.

Overall, SVAR is able to provide an instruction set that successfully abstracts away aspects such as data size, segmentation, orientation, multiplication overflow, etc. Programmers are able to add the same SPAR-2 functionality to their programs at an easier, higher level with SVAR.

6 Conclusion and Future Work

SVAR provides a common platform for code portability across SPAR-2 configurations. SVAR's new instruction set allows programmers to utilize SPAR-2 with less knowledge of its design. Finally, SVAR does all of this while providing a level of performance similar to existing low-level libraries. With AI continuing to grow and FPGA accelerators growing to meet its needs, the demand for portable, high-level code to leverage these technologies will also increase. While there is still much more work to be done with SPAR-2 and other reconfigurable designs, SVAR acts as a good stepping stone for code portability and ease of programming on FPGAs.

For future work, there is the possibility of looking into compiler solutions to help solve some of SVAR's run time overhead. More efficient data movement for storing should also be looked into as the current store instruction has the highest overhead according to the results. Lastly, there is work to be done with making SVAR compatible with more accelerator designs.

Bibliography

[1] L. Deng, "Artificial intelligence in the rising wave of deep learning: The historical path and future outlook [perspectives]," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 180–177, 2018.

[2] A. Haleem, M. Javaid, and I. H. Khan, "Current status and applications of artificial intelligence (ai) in medical field: An overview," *Current Medicine Research and Practice*, vol. 9, no. 6, pp. 231–237, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235208171930193X

[3] L. Gan, M. Yuan, J. Yang, W. Zhao, W. Luk, and G. Yang, "High performance reconfigurable computing for numerical simulation and deep learning," *CCF Transactions on High Performance Computing*, vol. 2, no. 2, pp. 196–208, Jun 2020. [Online]. Available: https://doi.org/10.1007/s42514-020-00032-x

[4] R. Perrault, Y. Shoham, E. Brynjolfsson, J. Clark, J. Etchemendy, B. Grosz, T. Lyons, J. Manyika, S. Mishra, and J. C. Niebles, *The AI Index 2019 Annual Report*. AI Index Steering Committee, Human-Centered AI Institute, Stanford University, 2019.

[5] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen, and W.-m. Hwu, "Accelerating sparse deep neural networks on fpgas," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[6] D. Wang, K. Xu, Q. Jia, and S. Ghiasi, "Abm-spconv: A novel approach to fpga-based acceleration of convolutional neural network inference," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317753

[7] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, "Comefa: Compute-in-memory blocks for fpgas," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9.

[8] S. Basalama, A. Panahi, A.-T. Ishimwe, and D. Andrews, "Spar-2: A simd processor array for machine learning in iot devices," in *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*, 2020, pp. 141–147.

[9] "Stack overflow developer survey 2022." [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-language-prof

[10] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira, "Simple, portable and fast simd intrinsic programming: Generic simd library," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 9–16. [Online]. Available: https://doi.org/10.1145/2568058.2568059

[11] S. N. T.-c. Chiueh and S. Brook, "A survey on virtualization technologies," *Rpe Report*, vol. 142, 2005.

[12] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid simd: Abstracting simd hardware using lightweight dynamic mapping," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 216–227.

[13] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The case for virtual register machines," in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, ser. IVME '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 41–49. [Online]. Available: https://doi.org/10.1145/858570.858575

[14] V. Makarov, "Fighting register pressure in gcc," in *Proceedings of the GCC Developers' Summit*, 2004, pp. 85–103. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e915a974bcf357a67c1affc82c9cb7aeab53d2aa#page=85

[15] A. Gonzalez, M. Valero, J. Gonzalez, and T. Monreal, "Virtual registers," in *Proceedings Fourth International Conference on High-Performance Computing*, 1997, pp. 364–369.

[16] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: Association for Computing Machinery, 1982, p. 98–105. [Online]. Available: https://doi.org/10.1145/800230.806984