Enabling Vector Load and Store Instructions on HammerBlade Architecture

Robert Ramstad

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science
University of Washington
2024

Committee:

Michael Taylor
Jacques Rudell
Scott Huack

Program Authorized to Offer Degree:
Electrical and Computer Engineering
Abstract

Enabling Vector Load and Store Instructions on HammerBlade Architecture

Robert Ramstad

Chair of the Supervisory Committee:

Michael Taylor

Electrical and Computer Engineering

Traditionally, computer architecture has been dominated by overly complex instruction sets that created a "solution" to every problem by adding another instruction. If these complex instructions sets are one side of a coin, the Reduced Instruction Set Computer (RISC)-V architecture is the other. RISC-V processors consist of 47 base instructions. Having such a low amount of instructions is both the biggest strength and the biggest weakness of the new era of RISC-V processors. Currently, there is a remarkable lack of high performance RISC-V processors. The Hammerblade architecture is one of the few. The main difference between Hammerblade and other RISC-V processors is its leveraging of parallel computer architecture as a multi-core system. However, while Hammerblade consists of potentially thousands of cores, it does not perform any data-level parallelism.

The primary intent of this thesis is to further understand how to increase the local memory throughput by extending the RISC-V core to include Single Instruction Multiple Data (SIMD) loads and stores. This will add the capability to locally load and store four words of data on top of the existing singular word loads and stores. A single Vanilla RISC-V core can now have the potential of a 4x speedup in loads and stores. This will allow Hammerblade to not only leverage the parallelism of the manycore architecture, but also the data-level parallelism on each individual core.
Acknowledgement

I would like to thank my advisor Professor Michael Taylor, and BSG member Tommy Jung, for their help.
Declaration

I, Mr. Robert Ramstad hereby declare that this thesis is the record of authentic work carried out during the academic year 2023 - 2024 and has not been submitted to any other University or Institute towards the award of any degree.
BSG Acknowledgement

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([16, 28, 109, 106, 65, 64, 111, 12, 103, 76, 34, 81, 5, 77, 33, 35, 57, 102]), ASIC Clouds ([98, 107, 96, 85, 59, 58, 62, 88]), open source hardware ([97, 87, 26]), RISC-V ([68, 75, 74, 24, 110, 2, 101, 53, 73, 63, 20, 72]), Network-on-Chips ([55, 69, 112, 92, 60]), security ([19, 45, 44, 4]), benchmark suites ([99, 61, 10]), dark silicon ([84, 83, 84, 89, 14, 36, 33, 102]), multicore ([68, 42, 41, 43, 86, 92, 91, 95, 82, 60, 90, 93, 94, 104]), compiler tools ([48, 30, 49, 32, 51, 50, 31, 108, 1, 8]) and FPGAs ([113, 47, 10]).
# Contents

1 Introduction 1  
1.1 Chapter 1.1. Thesis Overview 2  

2 Literature Review and Background 3  
2.1 HammerBlade 3  
2.1.1 Architecture 4  
2.1.2 On-Chip Network 5  
2.1.3 Cache Tiles and Memory 6  
2.1.4 HammerBlade Software 7  
2.1.5 HammerBlade Compute Tiles 8  
2.2 Vanilla Core 9  
2.2.1 Instruction Set Architecture 9  
2.2.2 Programming Model 9  
2.2.3 Pipeline Architecture 11  
2.3 Single Instruction Multiple Data 12  
2.3.1 Basics of SIMD 12  
2.3.2 RISC-V “V” Extension 13  

3 Implementation 14  
3.1 Design Overview 14  
3.2 Instructions and Uses 16  
3.3 Register File 17  
3.3.1 Store Instruction 18  
3.3.2 Load Instruction 19  
3.3.3 System Verilog Module 19  
3.4 EXE Stage 21  
3.4.1 Load-Store Unit 22  
3.4.2 recFNToFN Module 23  
3.5 MEM Stage 24  
3.5.1 DMEM 24  
3.5.2 Float Scoreboard 25  
3.6 WB Stage 26  
3.6.1 Pipeline changes 26  

4 Verification and Benchmarks 27  
4.1 Verification 27  
4.1.1 Basic Verification 27
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Architecture of Celerity Chip</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>HammerBlade Node and Tile Layout</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Ruche Network</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>Cache Tile Logic</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>Tile Group Layout of SPMD Software</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Compute Tile Layout</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Tile Floor Plan in TSMC 16nm</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>EVA Mapping in HammerBlade Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.9</td>
<td>Vanilla Core Tile Group Coordinates</td>
<td>10</td>
</tr>
<tr>
<td>2.10</td>
<td>The Architecture of RISC-V Vanilla Core</td>
<td>11</td>
</tr>
<tr>
<td>2.11</td>
<td>Goal of SIMD Architecture</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>SIMD Vanilla Core (Changes in red)</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>RV32 opcode structure for load and store instructions</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>SIMD Register File Schematic</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Standard vs SIMD regfile outputs</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Store Example</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>EXE Pipeline with SIMD data in red</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>Local Load Buffer Schematic</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>DMEM Output Mux</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>All Successfully Completed SPMD Programs</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Remote Load Delay for SIMD Store</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Tables

4.1 bsg_barrier_simulation_results .............................................. 27
4.2 asm_dmem_test_report ....................................................... 32
4.3 asm_memcpy_dram_results .................................................... 34
4.4 Instruction count for SGEMM .................................................. 36
4.5 SGEMM Speedup ................................................................. 37
4.6 SGEMM Core Utilization ....................................................... 38

5.1 DMEM Area ........................................................................... 39
5.2 DMEM & IMEM read/write power .............................................. 40
5.3 Instruction Power from IMEM DMEM ...................................... 40
5.4 Various DMEM configuration read/write power .......................... 41
5.5 Instruction Power from IMEM DMEM ...................................... 41
Chapter 1

Introduction

Even with the RISC-V open source revolution\cite{27} still in its infancy, the world of hardware design has been drastically changed. Traditionally instruction set architectures (ISA’s) are hidden behind closed doors and require potentially millions of dollars to get permission to use\cite{7}, the RISC-V ISA offers a new approach. Currently there is a lack of powerful RISC-V cores, not due to limitations of the ISA itself, but limitations of the designs available.

The HammerBlade Manycore\cite{75} is a processing behemoth. Simply put, this chip is a modern spectacle of parallel computing. Each core is made up of compute tiles connected via a mesh-like network on chip (NoC)\cite{22}. The software running on HammerBlade is written in HammerBlade CUDA-lite allowing for dispatching of workloads to subsets of the core. All of this is compiled by the open source GNU Compiler Collection (GCC).

While the HammerBlade Manycore takes advantage of parallel computing via multiple cores, it currently does not take advantage of any parallel processing in the individual cores. This is a huge missed opportunity, as each and every one of the cores can be optimized to load and store much quicker.

This thesis will summarize my work while I address the problem proposed above by investigating potential solutions to vectorizing memory access in HammerBlade. This will only be for float loads and stores that occur locally within a singular RISC-V core. While most of my work was concentrated on the hardware changes, I also worked within GCC to add custom instructions and rewrote multiple benchmarks using said instructions.
1.1 Chapter 1.1. Thesis Overview

The goal of this thesis is to provide a survey of modern parallel computer architecture, my proposed architecture changes, and future work to be done.

Chapter 2. Literature Review

This chapter goes over relevant literature and background information required to fully understand the task at hand. It will primarily focus on ISA design, vectorization techniques, and HammerBlade specific architecture and programming.

Chapter 3. RISC-V Vanilla Core Design Overview

All changes made to the existing RISC-V Vanilla Core are outlined and described. First going over the intended use of the added instructions, followed by the changes required to make it happen.

Chapter 4. Verification and Benchmarks

The scope of this thesis does not include full compiler support in the GCC toolchain, however some work must be done for the system to understand the added instructions. This chapter covers all these changes and why they must be made for proper testing of SIMD instructions. In addition, it describes how the core reacts to a variety of benchmarks in the HammerBlade benchmark suite, quantifying the gains achieved from the changes made.

Chapter 5. Cost Analysis

This chapter focuses on the cost of implementing the augmented core. The power and area costs are analyzed to further understand the cost of modification.

Chapter 6. Conclusions and Future Works

The final chapter talks about where the logical next steps could be to further advance this research. It also reflects on the issues and limitations of the current implementation.

Appendix. Python Code for Power Analysis

The appendix contains the code related to calculating the net zero power breakpoint.
Chapter 2

Literature Review and Background

2.1 HammerBlade

The HammerBlade Manycore is a highly parallel processor that is designed with open source hardware in mind. The basic format of this architecture is a tile array connected by a mesh network. The Celerity chip (2017)[3] was one of the first fully taped out versions of the HammerBlade architecture. Celerity is a 511 core system built in 16nm which achieved a world record in RISC-V and Coremark performance. This chip is also supported by the RISC-V Rocket Core [6] and a binarized neural network accelerator for real time computer vision applications.

![Architecture of Celerity Chip](image.png)

Figure 2.1: Architecture of Celerity Chip [75]
2.1.1 Architecture

In the simplest terms, the HammerBlade architecture is an array of interconnected tiles by a mesh network on chip (NoC). Each tile can serve a different function, either as a throughput optimized Vanilla RISC-V cores[67], or any custom accelerator. To support these tiles is victim cache (Vcache) tiles to coordinate with off-chip DRAM (HBM2).

![HammerBlade Node and Tile Layout](image)

Figure 2.2: HammerBlade Node and Tile Layout [21]
2.1.2 On-Chip Network

The HammerBlade architecture makes use of a Ruche Network [22] which is a 2-d mesh with additional parallel links. This network has two separate networks, the forward and reverse network. The forward network deals with remote load and store requests while the reverse network deals with all responses. The Ruche network is dimension-order routed, meaning that it is either X-then-Y in the case of the forward network, or Y-then-X for the reverse network. Dimension-order routing prevents deadlock and also removes path diversity [70].
2.1.3 Cache Tiles and Memory

As seen in Figure 2.2, each tile array is supported by rows of victim caches (Vcaches). These Vcaches communicate with memory controllers that manage various types of high bandwidth memory [52].

![Diagram of Cache Tile Logic](image)

Figure 2.4: Cache Tile Logic [37]

The Vcache follows a 2-stage blocking cache, tag lookup (TL) and verify (V). The Vcache will handle one miss at a time and those misses will stall the cache pipeline. Vcaches are 8-way set associative [54]. To communicate with the Vcaches, compute tiles will send request packets out to the network which are dealt with and responded to by the respective Vcache. The off-chip DRAM address space is interleaved between the Vcache tiles to balance requests among the Vcache array.
2.1.4 HammerBlade Software

The HammerBlade software architecture follows the CUDA-lite model[^100]. This is a derivative model of the CUDA model popularized by NVIDIA[^25]. The main idea behind this programming model is to treat the GPU as a separate workhorse. The host dispatches instructions to portions of the GPU which subsequently computes and responds with the relevant data. On HammerBlade this is done via tile groups.

![Figure 2.5: Tile Group Layout of SPMD Software][15]

Each of these groups collectively execute a kernel in parallel, share local memory, and are synchronized using barriers. This allows the desired code to be split into manageable slices and dispatched to tile groups. Each individual tile has access to its current tile group dimensions, their position within said group, and their portion of the entire work[^15].
2.1.5 HammerBlade Compute Tiles

While the compute tiles on the HammerBlade system can be any specialized accelerator, the general purpose Vanilla Core will be the primarily focus. Similarly, it is where most of this thesis’ work will be done. The Vanilla Core is a 5-stage in-order RISC-V core. The standard Vanilla Core implements the RV32IMF(A) ISA. This thesis proposes the addition of two new vector-like instructions to this base ISA. The Vanilla Core has a 3-stage ALU, and 4KB i-cache (IMEM) and local DMEM. The Vanilla Core supports non-blocking remote loads which allows it to have pending memory operations (in this case remote loads) while still allowing for other instructions to complete. These tiles also include a network router, to send and receive data from the network itself.

![Figure 2.6: Compute Tile Layout](image-url)
2.2 Vanilla Core
The Vanilla Core is the workhorse and most basic building block of the HammerBlade chip. While any type of accelerator is accepted, the Vanilla Core is the standard compute tile used in the HammerBlade architecture.

2.2.1 Instruction Set Architecture
The Vanilla Core is a 5-stage partially in-order RISC-V processor that implements the RV32IMF(A) ISA. Rather than being optimized for individual core performance, it is optimized for aggregated performance. The Vanilla Core is silicon proven with both 180nm and 16nm tapeout and a prototype manycore chip with 16x31 cores achieved a record CoreMark score of 812,350.

2.2.2 Programming Model
The Vanilla Core operates in a SPMD (Single Program Multiple Data) setting when used in the HammerBlade architecture. Each Vanilla Core is computing the same program on a different subset of the data that is dispatched via the host tile. Each Vanilla Core operates in a 32-bit virtual address, called the Endpoint Virtual Address (EVA) allowing for in-group memory sharing.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Address Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local DMEM</td>
<td>0000_0000_0000_0000_0001_????<em>????</em>????</td>
</tr>
<tr>
<td>Global</td>
<td>01yy_yyyy_xxxx_xx??<em>????</em>????<em>????</em>????</td>
</tr>
<tr>
<td>In-Group</td>
<td>001y_yyyy_xxxx_xx??<em>????</em>????<em>????</em>????</td>
</tr>
<tr>
<td>L2/DRAM</td>
<td>1xx?<em>????</em>????<em>????</em>????<em>????</em>????_????</td>
</tr>
</tbody>
</table>

Figure 2.8: EVA Mapping in HammerBlade Architecture
• Local DMEM: Accesses the local 4KB SRAM on an individual tile.

• Global: Accesses remote endpoints using x,y coordinates.

• In-Group: Accesses the tiles via tile group origin. This is set via configurable registers on each Vanilla Core.

• L2/DRAM: Each address of the off-chip DRAM corresponds to an address in each L2 cache located on the border of the tile array. The x-coordinate and virtual bank address provide a physical address in the off-chip DRAM.

Figure 2.9: Vanilla Core Tile Group Coordinates [37]
2.2.3 Pipeline Architecture

The Vanilla Core follows the standard 5-stage pipeline (IF,ID,EXE,MEM, WB). However, there are a few unique quirks to the Vanilla Core. After ID, the pipeline splits into two paths EXE and FP_EXE (Floating point execute). The EXE path has the LSU, ALU, FP_INT, and IDIV units while the FP_EXE has a 2-stage FPU and an FDIV/FSQRT unit.

Figure 2.10: The Architecture of RISC-V Vanilla Core
2.3 Single Instruction Multiple Data

The bulk of this thesis analyzes the implementation of Single Instruction Multiple Data (SIMD) memory instructions into the aforementioned Vanilla Core. The key understanding with SIMD is that many computation heavy algorithms rely on a series of consecutive loads and stores before and after computing the workload. This is commonly a considerable percentage of the overall instructions executed by each individual tile.

2.3.1 Basics of SIMD

The basic idea behind SIMD instructions is that sequenced loads and stores in consecutive memory locations can be leveraged to convert a series load/store instructions into one larger load/store instruction. This SIMD instruction would load/store 4x the amount of data in the same amount of time a normal load/store would take to complete. In RISC-V architecture, this can require minimal hardware additions [80]. This is often done by something called a “vector unit”, essentially treating the string of data as a vector and storing the entire vector at once.

![Figure 2.11: Goal of SIMD Architecture](11)

The general idea behind SIMD is that it is often desirable to have the capabilities to do the same computation on separate sets of data in parallel.
2.3.2 RISC-V “V” Extension

There is currently a vector extension for the RISC-V ISA that has been implemented in a variety of processors. One of note is the RISC-V2 \[66\], a scalable RISC-V processor with a full vector processing unit (VPU) and up to 16 lanes for vector instructions. While it achieved impressive speeds, the hardware costs are incredibly high. In the case of the RISC-V2 the 16-lane Vector core required over 10x average power and 6x area\[66\]. This highlights the expensive hardware overhead required to fully implement the V extension for the RISC-V ISA.
Chapter 3

Implementation

3.1 Design Overview

The primary goal behind this instruction extension on the Vanilla Core is twofold. First, to increase the speed at which each individual tile can compute its individual workload. And second, to ensure that this speed does not come at the cost of power or area. With 2048 cores in the current HammerBlade design, a drastic increase in power or area could be heavily detrimental to the performance of the design. As mentioned earlier, the Vanilla Core is optimized for computing in parallel with other Vanilla Cores rather than only individual computation. The desired change is to add local SIMD float loads and local SIMD float stores. An important aspect of the proposed design is to keep the existing logic intact as much as possible, this means adding parallel buses for SIMD data and adding logic only in required locations to ensure proper data movement throughout the core.

The high-level changes to the core are as follows

- Decode (ID) Stage: replaced float register file with SIMD float register file
- Execute (EXE) Stage: added SIMD output from register file, Altered LSU to support SIMD data
- Memory (MEM) Stage: DMEM changed to 4x width, Float scoreboard changed
- Write Back (WB) Stage: Added logic to support SIMD enable signal to register file
Figure 3.1: SIMD Vanilla Core (Changes in red)
3.2 Instructions and Uses

The instructions must be described in order to understand how the modifications will support the implementation. Two instructions will be added to the existing Vanilla Core: Float Load SIMD (FLS) and Float Store SIMD (FSS). The opcodes follow the instructions single word counterparts FLW and FSW respectively.

**Float Load SIMD**

`fls register grp, address`

The SIMD load instruction will load 4 words from local memory given by address into the register group given by `register grp`.

**Float Store SIMD**

`fss register grp, address`

The SIMD store instruction will store 4 words from the register group given by `register grp` into the DMEM address given by `address`.

---

Figure 3.2: RV32 opcode structure for load and store instructions
3.3 Register File

In order to support the execution of 4-word wide loads and stores, the float register file must be changed accordingly. It’s important to note that in this implementation the SIMD instructions can only be called on registers that are a multiple of 4 (0, 4, 8, etc). The function of the register file largely stays the same, but now the SIMD data is output alongside the original read data.

![Figure 3.3: SIMD Register File Schematic](image)

Note that the registers are grouped into groups of 4. This allows the requested data to be outputted using only the input address and the SIMD signal. This is what is a considered a banked register due to the grouping of the registers (banking). This heavily decreases the area cost by limiting unnecessary multiplexers being added.
3.3.1 Store Instruction

The semantics of the normal FSW (float store word) is as follows: when valid read\((r_{\text{v,i}})\) is turned on, the register file outputs the data from the individual register decided by the read address\((r_{\text{addr,i}})\). This data is then sent through the pipeline to the EXE stage.

The new semantics of FSS (float store SIMD) is the exact same as before, however now the registers are broken into groups of 4. When valid read is turned on, not only is the read address register outputted, but also the data from registers 2-4 of the register group which contains the read address.

When FSW is called, only the read address data is required. However when FSS is called, both the read address data and the data from that register group are required.

![Figure 3.4: Standard vs SIMD regfile outputs](image)

Shown below are example store instructions to further understand the SIMD register file function. The new register file SIMD output allows the existing logic to stay intact, while also enabling the SIMD data to follow in parallel to the MEM stage.

<table>
<thead>
<tr>
<th>fsw, addr = 1</th>
<th>fss, read_addr = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>frs2 = 20</td>
<td>Not allowed, read_addr must be mult of 4</td>
</tr>
<tr>
<td>frs2_simd = 20,30,40</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fsw, read_addr = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>frs2 = 10</td>
</tr>
<tr>
<td>frs2_simd = 20,30,40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fss, read_addr = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>frs2 = 10</td>
</tr>
<tr>
<td>frs2_simd = 20,30,40</td>
</tr>
</tbody>
</table>

![Figure 3.5: Store Example](image)
3.3.2 Load Instruction

The semantics of the normal FLW (float load word) is as follows: when write valid (w.v.i.) is turned on, the data at the input port is stored into the write address (w.addr.i).

The SIMD core FLW instruction acts slightly differently. The input address corresponds to the register groups rather than the individual registers. To access the individual registers write valid (w.v.i.) is converted to a 4-bit enable signal. Each bit corresponds to the register within the group that is being written to.

The new FLS instruction acts in the exact same as the SIMD core FLW instruction, however the 4-bit enable signal is all 1’s (4'b1111) meaning the entire register group is written to. Again, the SIMD instruction is only available on registers that are a multiple of 4.

3.3.3 System Verilog Module

The following code is the module used for the SIMD register file.

```verilog
'include "bsg_defines.sv"

module simd_regfile
    #('BSG_INV_PARAM(width_p), 'BSG_INV_PARAM(els_p),
    , localparam addr_width_lp='BSG_SAFE_CLOG2(els_p))
    #
    (input clk_i,
     input reset_i,
     input [3:0] w_v_i //one hot write enable,
     input [addr_width_lp-1:0] w_addr_i,
     input [3:0][width_p-1:0] w_data_i //width = 132,
     input [2:0] r_v_i,
     input [2:0][addr_width_lp-1:0] r_addr_i,
     output logic [2:0][width_p-1:0] r_data_o //carries frs1,2,3,
     output logic [2:0][width_p-1:0] rs2_simd_data_o //NEW OUTPUT, width 99)
    #
    wire unused = reset_i;

    logic [2:0][addr_width_lp-1:0] r_addr_r;

    always_ff @(posedge clk_i)
        for (integer i = 0; i < 3; i++)
            if (r_v_i[i]) r_addr_r[i] <= r_addr_i[i];

    logic [3:0][width_p-1:0] mem_r [(els_p>>2)-1:0];

    assign r_data_o[0] = mem_r[r_addr_r[0][4:2]][r_addr_r[0][1:0]];
```
assign r_data_o[1] = mem_r[r_addr_r[1][4:2]][r_addr_r[1][1:0]];
assign r_data_o[2] = mem_r[r_addr_r[2][4:2]][r_addr_r[2][1:0]];

for (genvar i = 0; i < 3; i++)
    assign rs2_simd_data_o[i] = mem_r[r_addr_r[1][4:2]][i+1];

always_ff @(posedge clk_i) begin
    for (integer i = 0; i < 4; i++) begin
        if (w_v_i[i]) begin
            mem_r[w_addr_i[4:2]][i] <= w_data_i[i];
        end
    end
endmodule

'BSG_ABSTRACT_MODULE(simd_regfile)
3.4 EXE Stage

To support the SIMD data coming from the register file the EXE stage needed to be modified to include said data in the pipeline. The load store unit (LSU) needed to be modified to support the SIMD data.

Figure 3.6: EXE Pipeline with SIMD data in red
3.4.1 Load-Store Unit

The Load-Store Unit (LSU) is a combinational logic element in the EXE stage of the Vanilla Core. It is responsible for address generation, address checking, and handshaking logic with the pipeline and network. The LSU checks each memory address and determines whether it’s a local or remote address. Local instructions take precedence over remote instructions. A Local Load Buffer [Figure 3.7] ensures that a remote load does not overwrite the DMEM output.

![Figure 3.7: Local Load Buffer Schematic](image)

The main changes made to the LSU support the masking of the data being sent to the DMEM. This ensures that FSW instructions properly send data to the correct location, and that FSS instructions send all expected data.

The semantics of the LSU related to local loads/stores are as follows: If it is a SIMD store (FSS), the store data is set to the SIMD data and the mask is set to all 1s. For a singular word store (FSW), the store data is replicated 4x and the mask depends on the last two digits of the DMEM write address. This allows the store data to be written to the proper word-sized DMEM address using the last two bits of the address to mask the input data. Below are the SIMD related portions of the LSU that determine the data and mask being sent to the DMEM.

```verilog
always_comb begin
  if (exe_decode_i.is_simd_op) begin
    store_simd_data = {{exe_rs2_simd_i},store_data};
    store_simd_mask = {{4(store_mask)}};
  end
end
```
else begin
  store_simd_data = {{4{store_data}}};
  unique casez (dmem_addr_o[1:0])
    2'b00: begin
      store_simd_mask = {{3{4'b0000}}, {store_mask}}; //only store [0]
    end
    2'b01: begin
      store_simd_mask = {{2{4'b0000}}, {store_mask}, {1{4'b0000}}}; //only store [1] element
    end
    2'b10: begin
      store_simd_mask = {{1{4'b0000}}, {store_mask}, {2{4'b0000}}}; //only store [2] element
    end
    2'b11: begin
      store_simd_mask = {{store_mask}, {3{4'b0000}}}; // only store [3] element
    end
    default: begin
      store_simd_mask = 16'b0000000000000000;
    end
  endcase
end

3.4.2 recFNToFN Module

The recFNToFN module is the implementation of the Berkeley HardFloat module [13] to conform the data to the IEEE 2001 Verilog standard for floating points [71]. The data is stored in the float register file according to the IEEE standard for Floating-Point Arithmetic and must be converted before it is sent to DMEM. This is one of the first major downsides to the SIMD extension, as to ensure a SIMD store can occur in one cycle all the applicable data must be converted which requires 4 recFNToFN modules compared to the previous one.
3.5 MEM Stage

The MEM stage was largely kept the same but the DMEM needed to be altered to support four wide loads. Similarly, the dependencies on the float scoreboard needed to be updated to ensure accurate SIMD stores.

3.5.1 DMEM

The DMEM is altered to be 4x wider and have 4x less depth in order to support four wide loads and stores while also staying at a size of 4KB. To ensure that single wide loads are properly dealt with, a mux is required at the output of the DMEM for the desired single word. The key to understand here is that the DMEM address sent throughout the core is 9 bits, but only 7 are required in the DMEM module itself. This is due to the fact that the address width is essentially abstracted away from the pipeline, and the remaining last two bits of the original DMEM address are instead used on an output mux.

Figure 3.8: DMEM Output Mux
3.5.2 Float Scoreboard

The scoreboard is a tool the core uses to ensure that there are registers that are neither reserved nor dependent. Oftentimes, SIMD stores are done after a series of remote loads which take more time to complete than local loads. Currently, a register address is “scored” when there is a remote load placed to that location. Subsequently, a store is not allowed from that register until the score has been cleared. The SIMD stores, unlike the single word stores, are reliant on 4 registers being clear which is reflected in the code below taken from the SIMD core’s scoreboard module.

```verilog
always_comb begin
    if(is_simd_op) begin
        rs_depend_on_sb_simd = (scoreboard_r[src_id_i[1] + 1] & op_reads_rf_i[1]) | 
                                (scoreboard_r[src_id_i[1] + 2] & op_reads_rf_i[1]) | 
                                (scoreboard_r[src_id_i[1] + 3] & op_reads_rf_i[1]);
    end
    else begin
        rs_depend_on_sb_simd = 1'b0;
    end
end
```
3.6 WB Stage

The strategy of the WB stage is largely similar to the EXE stage. The pipeline is altered to support the SIMD data, and 3 fNToRecFN must be added to convert the data to the floating point standard before it is stored.

3.6.1 Pipeline changes

To ensure that the data is stored properly to the float register file the original write enable must be converted to a 4-bit one-hot enable signal. The enable signal is 4'b1111 if it is a SIMD operation, or the one-hot encoded float register file write address. Below is the write back logic taken from the Vanilla Core System Verilog file.

```verbatim
always_comb begin
    float_rf_wen_li = 4'b0;
    float_rf_wdata_li = '0;

    if (flw_wb_ctrl_r.is_simd_op & ~float_remote_load_resp_v_i) begin
        float_rf_wdata_li = flw_recoded_data;
    end
    else begin
        float_rf_wdata_li = {4{float_rf_wdata}};
    end

    if (float_rf_wen & flw_wb_ctrl_r.is_simd_op & ~float_remote_load_resp_v_i) begin
        float_rf_wen_li = 4'b1111;
    end
    else if (float_rf_wen) begin
        unique casez (float_rf_waddr[1:0])
            2'b00: begin
                float_rf_wen_li = 4'b0001; //0001
            end
            2'b01: begin
                float_rf_wen_li = 4'b0010; //0010
            end
            2'b10: begin
                float_rf_wen_li = 4'b0100; //0100
            end
            2'b11: begin
                float_rf_wen_li = 4'b1000; //0001
            end
        endcase
    end
end
```

26
Chapter 4

Verification and Benchmarks

A large part of this thesis was ensuring that the SIMD Vanilla Core was properly executing both single word load/stores and SIMD load/stores. The work would be largely uninteresting without speedup analysis on common benchmarks comparing the SIMD Vanilla Core to the standard Vanilla Core.

4.1 Verification

The verification done on the SIMD Vanilla Core was done using the open source bsg_manycore repository[18] owned and maintained by the Bespoke Silicon Group at University of Washington. The verification suite is organized using a tiered system where levels 0 and 1 are verifying the execution of single instructions while levels 2 and 3 implement HammerBlade specific instruction loops (i.e remote loads) and the coremark benchmark suite[29]. To fully verify the core, all levels must be successfully completed.

4.1.1 Basic Verification

To verify successful execution of that standard Vanilla Core instruction set, the core was verified using all 3 levels of the bsg_manycore verification suite. The Vanilla Core uses a SPMD program loader to get relevant packets from the core to ensure it reaches the correct states (BSG_FINISH packets). This packet also gives simulation time stamps to ensure that there is no performance degradation between the SIMD and standard Vanilla Core.

To ensure that the core was not losing performance, the VCS simulation report was compared between the SIMD and standard Vanilla Core. Below is the bsg_barrier VCS simulation report compared between the SIMD and standard Vanilla Core.

<table>
<thead>
<tr>
<th>Runtime(ps)</th>
<th>Standard Core</th>
<th>SIMD Core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>580841000</td>
<td>580841000</td>
</tr>
</tbody>
</table>

Table 4.1: bsg_barrier simulation results
Figure 4.1: All Successfully Completed SPMD Programs
4.1.2 Vanilla Core Compiler

The HammerBlade architecture makes use of the Gnu Compiler Collection (GCC), an open source RISC-V compiler. GCC is responsible for turning C/C++ code into the assembly that runs directly on the core itself. To support the SIMD instructions the GNU-GCC toolchain must be altered to include SIMD load (FLS) and SIMD store (FSS). This is done by creating the associated match and mask of FLS and FSS instructions, and adding them to the toolchain. I used Hadi R. Sandid’s tutorial regarding this topic to successfully add the instructions to the toolchain. The SIMD instructions follow the same format as the standard FLW and FSW instructions and are largely the same outside of changed control logic. Below is the related code to the declaration of the FLS and FSS instructions and related FLW/FSW instructions.

```
#define RV32_FLW_S ‘RV32_Itype(‘RV32_LOAD_FP, 3'b010)
#define RV32_FSW_S ‘RV32_Stype(‘RV32_STORE_FP, 3'b010)

#define RV32_FLS_S ‘RV32_Itype(‘RV32_LOAD_FP, 3'b101)
#define RV32_FSS_S ‘RV32_Stype(‘RV32_STORE_FP, 3'b101)
```

4.1.3 Current Compiler Limitations

There are a few important caveats to understand with the addition of the FLS and FSS instructions to the Vanilla Core. First, the instructions can only be called on register addresses that are multiples of four. Second, the desired data must be in consecutive register addresses. As GCC converts the C/C++ code to assembly, it allocates registers accordingly; however it does not consider the use of consecutive registers. This eliminates the possibility of SIMD instructions being properly compiled by the current compiler, so a different approach to verifying the SIMD instructions must be taken.

---

1. There is currently a concerted effort by members of The Bespoke Silicon Group to alter the HammerBlade compiler to support the SIMD hardware extension however it was not complete at this time.
4.1.4 SIMD Verification

Because of the current compilers limitations, the standard C/C++ code cannot be used to properly simulate the Vanilla Core. Instead, assembly code is used to send SIMD instructions to the Vanilla Core. Two of the bsg_manycore SPMD verification benchmarks were chosen to verify proper SIMD instruction execution: asm_dmem_test and asm_memcpy_dram. Both benchmarks ultimately do the same thing, which is write 1024 pieces of data to memory and then read said data. The difference being that asm_dmem_test writes to the local 4KB DMEM (exclusively local load/stores), while asm_memcpy_dram writes to the modeled off-chip DRAM (local and remote load/stores).

4.2 Benchmarking

Once verification of the SIMD Vanilla Core was complete, the next step was to ensure that the speedup provided outweighs the costs of the hardware changes. There were three benchmarks used to evaluate the speedup of the SIMD core.

- **asm_dmem_test**: SPMD program that copies 1024 pieces of data to and from local DMEM
- **asm_memcpy_dram**: SPMD program that copies 1024 pieces of data to and from remote off-chip DRAM
- **sgemm**: Single precision general matrix multiplication running on the HammerBlade benchmark suite [17]
This benchmark was altered by unrolling the write and read loops by a factor of 4, and replacing the SW and LW instructions with the float SIMD alternatives. This required a moderate increase in overhead due to the fact the standard loop was using integer loads, while the SIMD loop uses float loads requiring conversion from integers to floating point. Below is the write and read loop code from the standard and SIMD program.

<table>
<thead>
<tr>
<th>Listing 4.1: Standard loop</th>
<th>Listing 4.2: SIMD loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem_to_dram_loop:</td>
<td>li t0, 0</td>
</tr>
<tr>
<td>// write to DMEM</td>
<td>li t1, 0</td>
</tr>
<tr>
<td>li t0, 0</td>
<td>addi t2, t1, 1</td>
</tr>
<tr>
<td>li t1, 0</td>
<td>addi t3, t2, 1</td>
</tr>
<tr>
<td>li t2, N</td>
<td>addi t4, t3, 1</td>
</tr>
<tr>
<td>write_loop:</td>
<td>li t5, N</td>
</tr>
<tr>
<td>sw t1, 0(t0)</td>
<td>init_loop:</td>
</tr>
<tr>
<td>addi t1, t1, 1</td>
<td>fcvt.s.w f0, t1 // = 0</td>
</tr>
<tr>
<td>addi t0, t0, 4</td>
<td>fcvt.s.w f1, t2 // = 1</td>
</tr>
<tr>
<td>bne t1, t2, write_loop</td>
<td>fcvt.s.w f2, t3 // = 2</td>
</tr>
<tr>
<td>// read from DMEM</td>
<td>fcvt.s.w f3, t4 // = 3</td>
</tr>
<tr>
<td>li t0, 0</td>
<td>fss f0, 0(t0)</td>
</tr>
<tr>
<td>li t1, 0</td>
<td>addi t1, t1, 4</td>
</tr>
<tr>
<td>li t2, N</td>
<td>addi t2, t2, 4</td>
</tr>
<tr>
<td>read_loop:</td>
<td>addi t3, t3, 4</td>
</tr>
<tr>
<td>lw t3, 0(t0)</td>
<td>addi t4, t4, 4</td>
</tr>
<tr>
<td>bne t3, t1, fail</td>
<td>addi t0, t0, 16</td>
</tr>
<tr>
<td>addi t1, t1, 1</td>
<td>bne t1, t5, init_loop</td>
</tr>
<tr>
<td>addi t0, t0, 4</td>
<td>// read from DMEM</td>
</tr>
<tr>
<td>bne t1, t2, read_loop</td>
<td>li t0, 0</td>
</tr>
<tr>
<td></td>
<td>li t1, 0</td>
</tr>
<tr>
<td></td>
<td>li t2, N</td>
</tr>
<tr>
<td></td>
<td>read_loop:</td>
</tr>
<tr>
<td></td>
<td>lw t3, 0(t0)</td>
</tr>
<tr>
<td></td>
<td>bne t3, t1, fail</td>
</tr>
<tr>
<td></td>
<td>addi t1, t1, 4</td>
</tr>
<tr>
<td></td>
<td>addi t0, t0, 16</td>
</tr>
<tr>
<td></td>
<td>bne t1, t2, read_loop</td>
</tr>
</tbody>
</table>
This code modification using the new instructions allows the 4 word-sized load/store to be replaced with a singular SIMD load/store. The speedup via VCS Simulation report is seen below.

<table>
<thead>
<tr>
<th></th>
<th>Standard Core</th>
<th>SIMD Core</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>10781000ps</td>
<td>5420000ps</td>
<td>1.989</td>
</tr>
</tbody>
</table>

Table 4.2: asm_dmem_test report

The calculations seen below approximate the change in loop instructions in the SIMD core. Via these calculations a speedup of 2.117 is seen, in-line with what is expected.

SIMD:
256 loops × 11 instructions + 256 loops × 6 instructions = 4352 total loop instructions (4.1)

Standard:
1024 loops × 4 instructions + 1024 loops × 5 instructions = 9216 total loop instructions (4.2)

### 4.2.2 asm_memcpy_test

With asm_dmem_test, the reads and writes are from DMEM, and as a result, asm_dmem_test can use SIMD operations. Conversely, asm_memcpy_test also reads and writes from DRAM and in these cases SIMD operations cannot be used. Read and writes from DRAM are what the Vanilla Core considers a “remote” operation and it receives the data from the network rather than the local scratchpad DMEM. Below is an example of converting the dmem_to_dram loop in asm_memcpy_dram where 4 FLW instructions can be replaced with a single FLS instruction. Note that the subsequent 32 fsw instructions are not viable for SIMD due to the fact they are storing to off-chip DRAM.
### Listing 4.3: Standard loop

**dmem_to_dram_loop:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>flw f0, 0(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f1, 4(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f2, 8(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f3, 12(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f4, 16(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f5, 20(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f6, 24(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f7, 28(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f8, 32(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f9, 36(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f10, 40(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f11, 44(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f12, 48(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f13, 52(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f14, 56(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f15, 60(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f16, 64(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f17, 68(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f18, 72(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f19, 76(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f20, 80(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f21, 84(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f22, 88(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f23, 92(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f24, 96(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f25, 100(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f26, 104(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f27, 108(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f28, 112(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f29, 116(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f30, 120(t0)</td>
<td></td>
</tr>
<tr>
<td>flw f31, 124(t0)</td>
<td></td>
</tr>
<tr>
<td>fsw f0, 0(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f1, 4(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f2, 8(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f3, 12(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f4, 16(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f5, 20(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f6, 24(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f7, 28(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f8, 32(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f9, 36(t1)</td>
<td></td>
</tr>
</tbody>
</table>

### Listing 4.4: SIMD loop

**dmem_to_dram_loop_SIMD:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>fls f0, 0(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f4, 16(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f8, 32(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f12, 48(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f16, 64(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f20, 80(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f24, 96(t0)</td>
<td></td>
</tr>
<tr>
<td>fls f28, 112(t0)</td>
<td></td>
</tr>
<tr>
<td>fsw f0, 0(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f1, 4(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f2, 8(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f3, 12(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f4, 16(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f5, 20(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f6, 24(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f7, 28(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f8, 32(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f9, 36(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f10, 40(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f11, 44(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f12, 48(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f13, 52(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f14, 56(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f15, 60(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f16, 64(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f17, 68(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f18, 72(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f19, 76(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f20, 80(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f21, 84(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f22, 88(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f23, 92(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f24, 96(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f25, 100(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f26, 104(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f27, 108(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f28, 112(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f29, 116(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f30, 120(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f31, 124(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f24, 96(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f25, 100(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f26, 104(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f27, 108(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f28, 112(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f29, 116(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f30, 120(t1)</td>
<td></td>
</tr>
<tr>
<td>fsw f31, 124(t1)</td>
<td></td>
</tr>
</tbody>
</table>
Listing 4.5: Standard loop cont.

```
fsw f10, 40(t1)
fsw f11, 44(t1)
fsw f12, 48(t1)
fsw f13, 52(t1)
fsw f14, 56(t1)
fsw f15, 60(t1)
fsw f16, 64(t1)
fsw f17, 68(t1)
fsw f18, 72(t1)
fsw f19, 76(t1)
fsw f20, 80(t1)
fsw f21, 84(t1)
fsw f22, 88(t1)
fsw f23, 92(t1)
fsw f24, 96(t1)
fsw f25, 100(t1)
fsw f26, 104(t1)
fsw f27, 108(t1)
fsw f28, 112(t1)
fsw f29, 116(t1)
fsw f30, 120(t1)
fsw f31, 124(t1)
```

In `asm_memcpy_dram` there are 19798 total instructions and 2048 of them are local loads/store that are converted to SIMD load/store instructions. The 2048 FLW/FSW instructions are converted into 512 FLS/FSS instructions. Using Amdahl’s law and assuming the speedup is 4x, the expected speedup is 1.081. However the table below shows a different story.

<table>
<thead>
<tr>
<th>Standard Core</th>
<th>SIMD Core</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>17603000ps</td>
<td>16493000ps</td>
<td>1.06730127933</td>
</tr>
</tbody>
</table>

Table 4.3: `asm_memcpy_dram` results

Almost 2% of the overall speedup is missing. The Vanilla Core’s handling of remote loads must be further understood to find the unexpected slowdown. The semantics of a remote load is that the core sends out a remote request, and “scores” the desired register address of remote load data. The SIMD Vanilla Core supports non-blocking loads but incoming remote loads are not promised to be in order via the on-chip response network. Consequently, there are situations where SIMD store instructions are waiting for an out of order remote load. This will stall the upcoming SIMD store instruction until the remote load is completed. Figure 4.2 shows a sequence of 8 SIMD stores after a series of remote loads. The float dependency wire indicates that for the next cycle the core must stall due to the fact the data to be stored is not ready yet via remote load.
The added stalls incurred by remote loads means that the speedup of SIMD stores after remote loads is 2x rather than 4x (and can be even lower depending on order of remote loads). The average speedup of SIMD instructions will now be 3x due to the fact SIMD loads are 4x and SIMD stores are 2x. Using these values, Amdahl’s law results in a speedup of 1.0714 which is in line with the speedup measured.
4.2.3 SGEMM

The Single precision General Matrix Multiplication (SGEMM) is a common benchmark that is implemented in the hb_hammerbench suite by The Bespoke Silicon Group. It simulates HammerBlade architecture using the CUDA-lite programming model previously described. This benchmark relies heavily on local FLW and FSW instructions making it an attractive recipient of a SIMD makeover. SGEMM can be ran using a variety of settings, but all benchmarking was done using SGEMM 512,2 (512x512 matrix multiplication, 2 passes).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Standard Core</th>
<th>SIMD Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>local flw</td>
<td>151519232</td>
<td>67108864</td>
</tr>
<tr>
<td>local fsw</td>
<td>50331648</td>
<td>0</td>
</tr>
<tr>
<td>local fls</td>
<td>0</td>
<td>21102592</td>
</tr>
<tr>
<td>local fss</td>
<td>0</td>
<td>12582912</td>
</tr>
</tbody>
</table>

Table 4.4: Instruction count for SGEMM

One difficulty with the SGEMM benchmark (and any hb_hammerbench benchmark) is that the CUDA-lite model is explicitly written in C/C++. While it was trivial to alter the assembly (asm) benchmarks, the CUDA-lite code required a separate avenue of attack. Due to the current compilers limitations within the current HammerBlade toolchain, inline assembly would be troublesome with how registers are allocated. The most efficient way to add SIMD instructions to this benchmark is by object dumping the existing kernel (code that runs on each tile) and rewriting portions of it in separate assembly functions and linking the functions in C. Below is an example of a rewritten portion of the SGEMM Kernel.

**kernel.cpp:**

```c
// load the psum
bsg_unroll(SUB_DIM)
for (int py = 0; py < SUB_DIM; py++) {
    bsg_unroll(SUB_DIM)
    for (int px = 0; px < SUB_DIM; px++) {
        psum[py][px] = sub_block_out[(py*BLOCK_DIM)+px];
    }
}
```

**SIMDkernel.cpp:**

```c
// load the psum
load_psum_s(sub_block_out);
```

The disassembly from these code snippets is as follows:
There are 5 separate linked assembly functions in the SIMD SGEMM kernel, allowing 84410368 FLW instructions to be converted into 21102592 FLS instructions, and 50331648 FSW instructions into 12582912 FSS instructions. This accounts for 23.53% of all instructions. Using Amdahl’s law and understanding that not all of the 4x speedup will be experienced from stores following remote loads, a speedup of 1.13 is achieved.

<table>
<thead>
<tr>
<th>Standard Core</th>
<th>SIMD Core</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6613918 cycles</td>
<td>6337718 cycles</td>
<td>1.04358035495</td>
</tr>
</tbody>
</table>

Table 4.5: SGEMM Speedup
In the core utilization statistics Table 4.6 there is suddenly an increase in ADDI, JAL, and JALR instructions. This is because when the assembly functions are called, the compiler must first calculate each operand, then JAL to the function location embedded into the assembly, and finally JALR back to the code from the function call. This accounts for 7.75% lost speedup. If this “lost” speedup is added to the simulation speedup the speedup is 1.12, which is what is expected.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SIMD Count</th>
<th>SIMD Percent</th>
<th>Standard Count</th>
<th>Standard Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>local flw</td>
<td>67108864</td>
<td>13.10%</td>
<td>151519232</td>
<td>26.47 %</td>
</tr>
<tr>
<td>local fsw</td>
<td>0</td>
<td>0.00 %</td>
<td>50331648</td>
<td>8.79 %</td>
</tr>
<tr>
<td>local ffs</td>
<td>21102592</td>
<td>4.12%</td>
<td>0</td>
<td>0.00 %</td>
</tr>
<tr>
<td>local fss</td>
<td>12582912</td>
<td>2.46%</td>
<td>0</td>
<td>0.00 %</td>
</tr>
<tr>
<td>addi</td>
<td>69826664</td>
<td>13.63%</td>
<td>55443879</td>
<td>9.68 %</td>
</tr>
<tr>
<td>jal</td>
<td>11550720</td>
<td>2.25%</td>
<td>0</td>
<td>0.00 %</td>
</tr>
<tr>
<td>jalr</td>
<td>11550720</td>
<td>2.25%</td>
<td>0</td>
<td>0.00 %</td>
</tr>
<tr>
<td>fnadd</td>
<td>268435456</td>
<td>52.39%</td>
<td>268435456</td>
<td>46.89 %</td>
</tr>
<tr>
<td>local ld</td>
<td>682791</td>
<td>0.13%</td>
<td>18113</td>
<td>0.00 %</td>
</tr>
<tr>
<td>local st</td>
<td>690345</td>
<td>0.13%</td>
<td>530476</td>
<td>0.09 %</td>
</tr>
<tr>
<td>bne</td>
<td>10862434</td>
<td>2.12%</td>
<td>10862420</td>
<td>1.90 %</td>
</tr>
<tr>
<td>blt</td>
<td>1043</td>
<td>0.00%</td>
<td>1048</td>
<td>0.00 %</td>
</tr>
<tr>
<td>bge</td>
<td>3072</td>
<td>0.00%</td>
<td>3072</td>
<td>0.00 %</td>
</tr>
<tr>
<td>slli</td>
<td>1050149</td>
<td>0.20%</td>
<td>2501</td>
<td>0.00 %</td>
</tr>
<tr>
<td>add</td>
<td>2256511</td>
<td>0.44%</td>
<td>1271473</td>
<td>0.22 %</td>
</tr>
<tr>
<td>sub</td>
<td>1158</td>
<td>0.00%</td>
<td>2048</td>
<td>0.00 %</td>
</tr>
<tr>
<td>lui</td>
<td>595265</td>
<td>0.12%</td>
<td>7242</td>
<td>0.00 %</td>
</tr>
<tr>
<td>or</td>
<td>384</td>
<td>0.00%</td>
<td>384</td>
<td>0.00 %</td>
</tr>
<tr>
<td>and</td>
<td>384</td>
<td>0.00%</td>
<td>384</td>
<td>0.00 %</td>
</tr>
<tr>
<td>fence</td>
<td>128</td>
<td>0.00%</td>
<td>128</td>
<td>0.00 %</td>
</tr>
<tr>
<td>barsend</td>
<td>128</td>
<td>0.00%</td>
<td>128</td>
<td>0.00 %</td>
</tr>
<tr>
<td>barrecv</td>
<td>128</td>
<td>0.00%</td>
<td>128</td>
<td>0.00 %</td>
</tr>
<tr>
<td>remote st global</td>
<td>128</td>
<td>0.00%</td>
<td>128</td>
<td>0.00 %</td>
</tr>
<tr>
<td>remote fsw dram</td>
<td>2097152</td>
<td>0.41%</td>
<td>2097152</td>
<td>0.37 %</td>
</tr>
<tr>
<td>remote seq flw dram</td>
<td>31457280</td>
<td>6.14%</td>
<td>31457280</td>
<td>5.49 %</td>
</tr>
<tr>
<td>remote fsw dram</td>
<td>524288</td>
<td>0.10%</td>
<td>524288</td>
<td>0.09 %</td>
</tr>
</tbody>
</table>

Table 4.6: SGEMM Core Utilization
Chapter 5

Cost Analysis

To fully understand the impact of the Vanilla Core proposed modifications, the power and energy characteristics must also be analyzed. As seen in the Tile Floor Plan in TSMC 16nm [2,7] the IMEM, DMEM, and register file take up 62% of the overall tile area [15]. Consequently, the IMEM and DMEM will be the primary focus regarding the area and power changes.

5.1 Area

Increasing the DMEM width while decreasing depth creates a 27% increase in overall DMEM Area [38]. Considering that the DMEM is one of the largest contributors to overall tile area, this is somewhat undesirable. Not only does the overall area increase, but the increased width causes the Y length to be over 10 times longer than the X length which will be a limiter, especially if looking to optimize the tile for area.

<table>
<thead>
<tr>
<th></th>
<th>depth:256, width:128</th>
<th>depth:1024, width:32</th>
</tr>
</thead>
<tbody>
<tr>
<td>X length (microns)</td>
<td>20.789</td>
<td>53.045</td>
</tr>
<tr>
<td>Y length (microns)</td>
<td>266.496</td>
<td>82.176</td>
</tr>
<tr>
<td>Area (microns²)</td>
<td>5540.185344</td>
<td>4359.02592</td>
</tr>
</tbody>
</table>

Table 5.1: DMEM Area
5.2 Power

The second aspect of the cost analysis is to understand the change in power after implementing SIMD modifications. To do this, two primary power sources will be considered: The IMEM and DMEM. The IMEM is the 4KB instruction cache and the DMEM is the local 4KB scratchpad used for all local loads/stores.

<table>
<thead>
<tr>
<th></th>
<th>SIMD DMEM</th>
<th>Standard DMEM</th>
<th>IMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Power(mA)</td>
<td>5.66E-03</td>
<td>2.21E-03</td>
<td>2.83E-03</td>
</tr>
<tr>
<td>Write Power(mA)</td>
<td>6.08E-03</td>
<td>2.82E-03</td>
<td>3.58E-03</td>
</tr>
<tr>
<td>Read Power per bit(mA)</td>
<td>4.42E-05</td>
<td>6.92E-05</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 5.2: DMEM & IMEM read/write power

While the overall power per read and write is significantly higher in the SIMD DMEM, it is actually more efficient in reading per bit. Both the IMEM and DMEM are static random-access memory (SRAM). The SRAM energy in the HammerBlade tile takes about 35% of the total energy to execute one scalar load/store\(^{[56]}\), which implies that a decrease or increase in power will heavily affect the power characteristics of a full HammerBlade array. Given this knowledge, there will be a “breakeven point” where if a program can convert a percentage of its load/stores into SIMD loads/stores it will save energy rather than use more energy.

To accurately calculate this, the total power used by single-word load/stores and SIMD load/stores must be understood. This is done by summing up the read and write power from IMEM, and read power for loads or write power for stores. Shown in Table 5.3 are the total power values for individual load/stores and 4x load/stores.

<table>
<thead>
<tr>
<th></th>
<th>SIMD Core</th>
<th>Standard Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLW(mA)</td>
<td>12.07E-03</td>
<td>8.62E-03</td>
</tr>
<tr>
<td>FSW(mA)</td>
<td>12.49E-03</td>
<td>9.23E-03</td>
</tr>
<tr>
<td>4xFLW(FLS) (mA)</td>
<td>12.07E-03</td>
<td>34.48E-03</td>
</tr>
<tr>
<td>4xFSW(FSS) (mA)</td>
<td>12.49E-03</td>
<td>36.92E-03</td>
</tr>
</tbody>
</table>

Table 5.3: Instruction Power from IMEM DMEM

Using these values, it can be computed that as long as 38.18% of load instructions and 34.86% of store instructions are able to be converted to SIMD, power will reach net zero. Note that these are load/store exclusive, meaning that if 38.18% of load instructions are converted that would cover the power cost of all remaining load instructions to a net zero; the same is true for stores. The related code for calculating this can be found in the appendix. Using SGEMM 512 as an example, out of the 201850886 total load/store instructions 134742022 were able to be converted to SIMD instructions-66.75%. Using the data from the Table 5.3, 5.487E05 mA of power is saved.
5.2.1 Banked DMEM alternative

To decrease the cost of single word load and store instructions, the SIMD DMEM can be "banked". The DMEM is split into two 64 wide 256 deep identical DMEM units rather than the single 128 wide 256 deep DMEM unit. This is an alternative approach to the existing SIMD DMEM and is not supported by the current Vanilla Core SIMD modifications.

<table>
<thead>
<tr>
<th></th>
<th>SIMD DMEM</th>
<th>Standard DMEM</th>
<th>Banked DMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Power(mA)</td>
<td>5.66E-03</td>
<td>2.21E-03</td>
<td>2.95E-03</td>
</tr>
<tr>
<td>Write Power(mA)</td>
<td>6.08E-03</td>
<td>2.82E-03</td>
<td>3.21E-03</td>
</tr>
<tr>
<td>Read Power per bit(mA)</td>
<td>4.42E-05</td>
<td>6.92E-05</td>
<td>4.61E-05</td>
</tr>
</tbody>
</table>

Table 5.4: Various DMEM configuration read/write power

The banked DMEM would only turn on the respective unit that contains the location required for a single word load/store. For SIMD instructions, both DMEM units would be turned on. As seen in Table 5.4, our single word instructions would incur a power cost that is significantly lower than the larger single-unit SIMD DMEM. However, the power cost of SIMD instructions would be substantially larger than the single-unit SIMD DMEM due to the need to power on both DMEM units.

<table>
<thead>
<tr>
<th></th>
<th>SIMD Core</th>
<th>Standard Core</th>
<th>Banked DMEM Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLW(mA)</td>
<td>12.07E-03</td>
<td>8.62E-03</td>
<td>9.36E-03</td>
</tr>
<tr>
<td>FSW(mA)</td>
<td>12.49E-03</td>
<td>9.23E-03</td>
<td>9.62E-03</td>
</tr>
<tr>
<td>4xFLW(FLS) (mA)</td>
<td>12.07E-03</td>
<td>34.48E-03</td>
<td>18.72E-03</td>
</tr>
<tr>
<td>4xFSW(FSS) (mA)</td>
<td>12.49E-03</td>
<td>36.92E-03</td>
<td>19.24E-03</td>
</tr>
</tbody>
</table>

Table 5.5: Instruction Power from IMEM DMEM

Using a banked DMEM, we would only need to convert 15.82% of load instructions and 8.11% of store instructions to reach our power breakpoint. In SGEMM 512, the banked DMEM would save 5.05382E05 mA of power. This is less power saved than with the single-unit SIMD DMEM but is more robust against increased power consumption from single word instructions. The two main downsides to using a banked DMEM are that the DMEM address logic will need to be changed to support SIMD load/stores accessing both units and that there is a slight area increase compared to the single-unit SIMD DMEM (5971 and 5540 microns$^2$ respectively).
Chapter 6

Conclusion and Future Work

6.1 Conclusion

The HammerBlade manycore architecture relies on high-throughput workhorse compute tiles, primarily via the RISC-V Vanilla Core. This thesis proposes a low-weight extension to the Vanilla Core that can not only improve the throughput of the SIMD Core compute tile but also decrease its power profile. First, the hardware was extended to support two instructions: FLS and FSS. This was done by creating parallel lanes along the standard FLW and FSW instructions and required control logic to support the widened data bus. Next, GCC was altered to allow for the use of the FLS and FSS instructions through custom assembly files. Multiple SPMD and HammerBlade benchmarks were altered and analyzed to understand the performance gained by the SIMD extension. Lastly, the power and area of the proposed changes were analyzed to further understand the cost of this modification.

This modification adds increased throughput for applications with high data-level parallelism with the potential for large power savings depending on the percentage of the application that can be converted to SIMD instructions. The primary drawback of this design is the increased area. With that in mind this SIMD core may not become the standard compute tile for the HammerBlade architecture, but can instead blend in seamlessly as a “soft” accelerator depending on the intended use of the HammerBlade system.

6.2 Future Work

While largely successful, there are a number of directions this work can go in the future. Listed below are a few ways that this work can be furthered and refined.

- **SIMD core area and switching power**: One largely unexplored area of this thesis is the area and power cost of the proposed changes outside of the SRAM. This may impact the efficacy of the modified SIMD design.

- **Compiler support**: The benchmarks chosen gave promising results, but can be fully investigated once the compiler modification is complete.

- **Striding**: Most common algorithms, if not storing to adjacent memory addresses, are storing to common intervals throughout memory. With the current design this is not feasible, and likely would require multiple DMEMs on chip to support.
• **Place and route**: This design was not fully pushed through the RTL flow. Completing this would further the understanding of the power and area costs of the proposed design.

• **Vector register**: Many current SIMD processors contain a vector register that is solely responsible for vector instructions (FLS, FSS). This was not in the scope of this thesis, and I expect the area required for another vector register to outweigh the potential gain in a manycore system.

These potential modifications would further the understanding of how SIMD instructions could transform the HammerBlade Manycore architecture.
Chapter 7

Appendix: Python Code for Power Analysis

7.1 Python Code for Power Analysis

To properly calculate the ratio of instructions required to "break even" power-wise was calculated using this simple python script:

```python
from decimal import Decimal

def TestValues(totalInstructions):
    totalInstructions = int(totalInstructions)
    minStore = 0
    minLoad = 0
    calcStore = -1
    calcLoad = -1
    for num in range(totalInstructions):
        if minStore == 0:
            calcStore = CalcStore(num, totalInstructions - num)
        if calcStore > 0:
            minStore = num
        if minLoad == 0:
            calcLoad = CalcLoad(num, totalInstructions - num)
        if calcLoad > 0:
            minLoad = num

    print("-----")
    print("total instructions: " + str(totalInstructions))
    print("minimum simd stores for positive delta : " + str(format_percentage(minStore /
        totalInstructions)))
    print("minimum simd loads for positive delta : " + str(format_percentage(minLoad/
        totalInstructions)))
    print("-----")

def CalcStore(simd, normal):
    return Decimal(Decimal("5.6025") * simd - Decimal("3.45") * normal)

def CalcLoad(simd, normal):
    return Decimal(Decimal("6.1075") * simd - Decimal("3.26") * normal)

def format_percentage(decimal_number, decimal_places=2):
    percentage = '{:.{}%}'.format(decimal_number, decimal_places)
    return percentage
```

44
def main():
    choice = input("enter total instructions: ")
    TestValues(choice)

if __name__ == "__main__":
    main()
Bibliography


[38] Bespoke Silicon Group. GF14S RAMDATASHEET. 2024.


Donghwan Jeon et al. “Kremlin: Like gprof, but for Parallelization”. In: *Principles and Practice of Parallel Programming (PPoPP)*. 2011.


Tommy Jung. personal communication. 2024.


Michael Bedford Taylor. “Your agile open source HW stinks (because it is not a system)”. In: ICCAD. 2020.


