Vectorizing Memory Access on HammerBlade Architecture

Sahil Athrij

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2024

Committee:

Michael McCourt

Michael Taylor

Nafiul Siddique

Program Authorized to Offer Degree:
Electrical and Computer Engineering

University of Washington

Abstract

Vectorizing Memory Access on HammerBlade Architecture

Saahil Athrij

Chair of the Supervisory Committee:

Michael McCourt

Electrical and Computer Engineering

The Reduced Instruction Set Computer(RISC)-V architecture, celebrated for its open-source flexibility and modular design, is a cornerstone for modern computing innovations. However, RISC-V [99] faces a significant challenge toward widespread adoption because of the sparse availability of high-performance RISC-V processors. HammerBlade[32] is one of the few high-performance RISC-v processors that fills this gap. One of the most significant advances in parallel computer architecture has been the introduction of multi-core systems and Single Instruction Multiple Data (SIMD) processors; HammerBlade uses a multi-core system (2048 cores) to achieve this high level of parallel computing. However, HammerBlade does not currently perform any data-level parallelism.

The overarching objective of this thesis is to enhance the processing capabilities of the HammerBlade chip by extending the vanilla RISC-V core[32]. The extension supports 128-bit (16-byte-wide) loads, replacing the 32-bit (4-byte-wide) loads. This advancement enables multiprocessing efficiency in a large-scale, 2048-core system. This objective encompasses a twofold strategy. The first aspect involves a hardware extension of the RISC-V architecture, focusing on implementing 16 Byte Wide Load instructions to enhance data throughput and processing efficiency in a densely populated multi-core environment. The second aspect of the objective revolves around the critical refinement of the GCC RISC-V compiler [26]. This thesis thoroughly analyzes compiler optimization techniques, particularly emphasizing vectorization[16] strategies that can exploit the newly introduced 16 Byte Wide Load.

The goal is to develop a compiler that not only utilizes the architectural nuances of the enhanced HammerBlade chip but also optimizes code for this unique hardware. By synergizing these hardware and software advancements, the research aims to break new ground in how multiprocessing tasks are managed and executed, setting a precedent for future innovations in the field of computer architecture and compiler design.

Acknowledgement

I would like to thank my advisors, Professor Michael Taylor and Professor Michael McCourt, who gave me their maximum support.

Declaration

I, Mr. Sahil Athrij hereby declare that this thesis is the record of authentic work carried out by us during the academic year 2022 - 2024 and has not been submitted to any other University or Institute towards the award of any degree.

Contents

List of Figures

Chapter 1

Introduction

The HammerBlade manycore [25] Accelerator-Network is an open source mesh-based On-Chip-Network [66].The HammerBlade software paradigm is Single-Program Multiple Data [12](SPMD). Abstractly, the user writes a number of application kernels, and allocates tiles to execute those kernels using the HammerBlade CUDA-Lite Runtime[32]. The back end of this CUDA-lite runtime is the GNU GCC compiler that compiles the CUDA-Lite to RISC-V executable binary.

A HammerBlade node consists of a network of tiles organized in a 2-D mesh configuration, known as the manycore accelerator network. This network, which may also be called an on-chip network, NOC, OCN, or simply "the network," connects the tiles. The node is equipped with a flexible high-bandwidth memory system and an I/O system. Within a HammerBlade node, tiles serve as processing units, memory, or I/O interfaces.[91]

The GNU Compiler Collection (GCC) is a set of compilers and development tools that support various programming languages. It's a key component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux [26]. It is an open source compiler that is the standard compiler for multiple linux and GNU projects.

1.1 Thesis Summary

My contribution to this field to introduce a compiler design that identifies opportunities for vectorization in code written for the HammerBlade architecture, automatically transforming scalar operations into vector operations where beneficial.

The structure of this thesis is designed to provide a comprehensive exploration of the problem space, the proposed solutions, and the potential future directions of this research. The chapters are organized as follows:

Chapter 2. Literature Review and Background

This chapter sets the stage for our work by reviewing existing literature in the fields of compiler design, vectorization techniques, and the specifics of HammerBlade architecture. It provides a historical context for our research, highlighting the evolution of vectorization and memory optimization techniques, and identifies the gap our work seeks to fill.

Chapter 3. System Design and Additional Instructions

Here, we delve into the design of the Vectorizing Memory Access Compiler, detailing the architectural decisions, algorithms, and heuristics used to enable effective vectorization and optimized memory access. This chapter also introduces any additional instructions or architectural features proposed to enhance the HammerBlade architecture's handling of vectorized operations and memory access patterns.

Chapter 4. GNU Toolchain Setup and Analysis

Here, we delve into the setup of the GNU toolchain and how to get started with modifying GDB and GCC. Additionally this chapter also deals with all the prerequistes required to understand GCC register allocation and GCC passes required for vectorization.

Chapter 5. GCC Vectorization

Focusing on the practical implementation, this chapter discusses how the GNU Compiler Collection (GCC) has been extended and adapted to support our vectorization and memory access strategies on the HammerBlade architecture. It covers the modifications made to GCC, the challenges encountered, and how they were overcome, providing insights into the integration of new compiler techniques into existing frameworks.

Chapter 5. Testing

This chapter focuses on the different testing results from the testing benchark suite outlined in chapter 3.in this chapter we aim to quantify the performance gains and highlight the potential of these advancements in the realm of high-performance computing.

Chapter 7. Future Work

The final chapter outlines the potential future directions for this research. It reflects on the limitations of the current implementation and proposes areas for further exploration, including more sophisticated vectorization strategies, dynamic memory access optimizations, and broader architectural support. This chapter aims to pave the way for ongoing advancements in compiler technology for parallel architectures like HammerBlade.

Appendix: Writing Vectorized Code

This short guide provides recommendations for future users of the HammerBlade vectorized architecture and compiler to write easily vectorizable code.

Chapter 2

Literature Review and Background

2.1 Survey on the Hammer blade Architecture

The HammerBlade [32] architecture is scalable, versatile data-processing framework designed to accommodate the computational demands of supercomputers. Its foundational building block, the HammerBlade node, integrates as a single System-on-Chip (SoC), demonstrating the architecture's modular design which facilitates the assembly of multiple nodes via high-bandwidth links exceeding 100Gbps. This review delves into the structural, operational, and application-specific facets of the HammerBlade architecture, underlining its innovative approach to addressing the contemporary and future challenges in high-performance computing (HPC) and beyond.

2.1.1 Architectural Design and Network

At the core of the HammerBlade architecture is an array of tiles interconnected by a 2-D mesh network, known as the manycore accelerator network or the on-chip network (NOC) [66], with each tile serving a distinct function—either as a processing element, memory, or I/O interface. This segmentation aligns with the architecture's objective to optimize data processing across varying computational needs.[91]

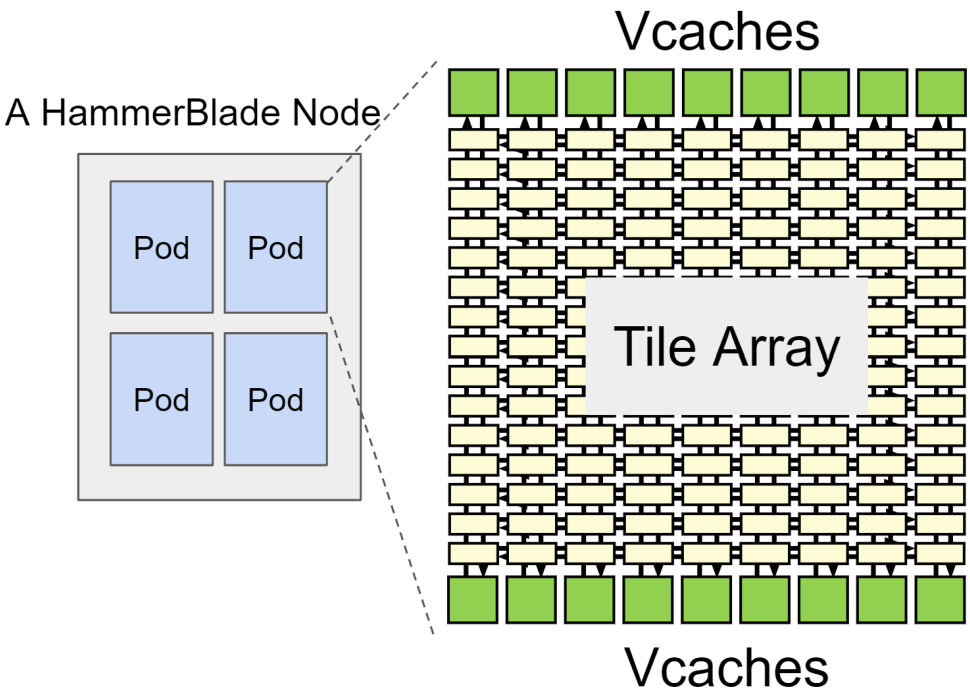

Figure 2.1: The Tile Structure of the Hammer Blade Architecture

2.1.2 Processing Elements

Three categories classify the processing tiles within a HammerBlade node: Throughput-optimized RISC-V tiles, Linux-capable RISC-V cores, and specialized accelerators. The throughput-optimized cores, called Vanilla-5 (V5) cores [99], are designed for floating-point operations and feature a scratchpad for local data storage alongside an instruction cache. There are also Linux-capable cores that leverage the open-source 64-bit BlackParrot RISC-V cores.

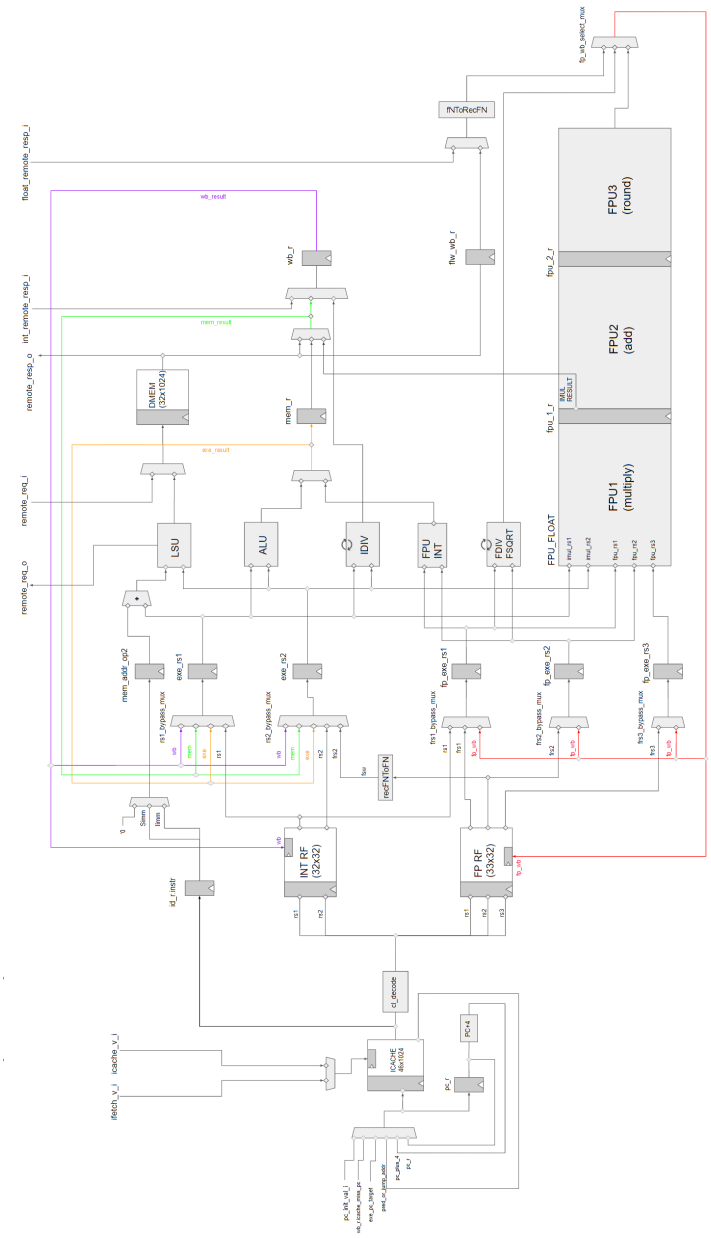

Figure 2.2: The Architecture of RISC-V Vanilla Core

2.1.3 Memory System and Caching

The HammerBlade architecture introduces a re-configurable and introspective cache system, facilitating efficient external memory interfacing through column-cache tiles. These tiles, positioned at the peripheral edges of the tile array[14], connect to memory controllers that manage multiple parallel memory channels across various memory types, including high bandwidth memory (HBM) [45], DDR4, and GDDR5 [58].

2.1.4 Communication Hierarchy and Address Space

A novel aspect of the HammerBlade architecture is its hierarchical communication framework, such as Pods. Including multiple Pods within each node facilitates a structured approach to managing communication latency and bandwidth. A non-uniform, globally addressable memory space complements this hierarchical design, enabling efficient data exchange and memory access patterns across the architecture. Such a configuration facilitates the development of sophisticated software abstractions for optimized data communication.

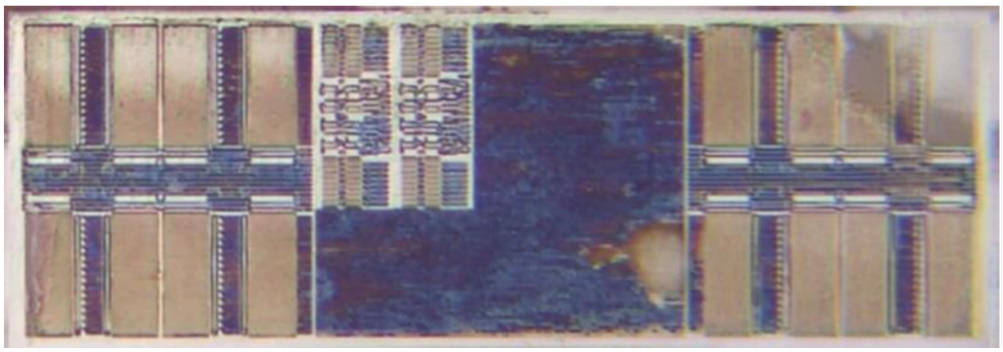

Figure 2.3: The Die Structure of the Hammer Blade Architecture

2.2 Survey on the Hammer blade Software

The HammerBlade architecture implements Single-Program Multiple Data (SPMD) programming model, facilitated by the HammerBlade CUDA-Lite[59] Runtime with the GNU toolchain at it's core.

2.2.1 SPMD Model

In SPMD[67] parallel execution, multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep that Single instruction Multiple Data (SIMD) or Single Instruction Multiple Threads(SIMT)[59] imposes on different data. In the SPMD model, a single program is executed simultaneously on multiple data elements. Here, each processing element (PEs) runs the same program but with different data elements, allowing for parallel processing and increased efficiency.

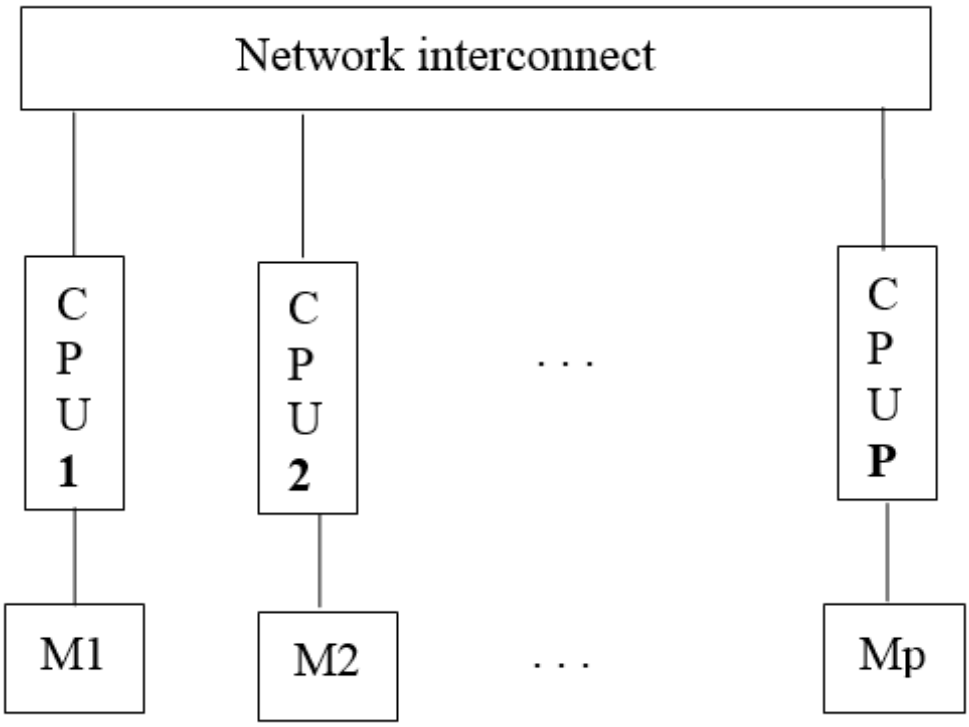

Figure 2.4: Programming Model of a SPMD chip

2.2.2 Tile Groups

The fundamental work-unit in the HammerBlade CUDA-Lite paradigm is the Tile Group, a two-dimensional collection of tiles that executes a single program concurrently. This structure allows for a flexible and efficient distribution of computational tasks across the architecture's tiles. The tile group can only be a uniform set of adjacent tiles. The maximum number of tiles in the tile group only limited by the number number of uniform adjacent tiles available.

Independent tile groups can run in parallel on different parts of the array.

Tile groups represent a collection of dynamically grouped tiles that characterize them based on shared values in their tile group origin registers. The software configures these registers. The primary attributes of tile groups include:

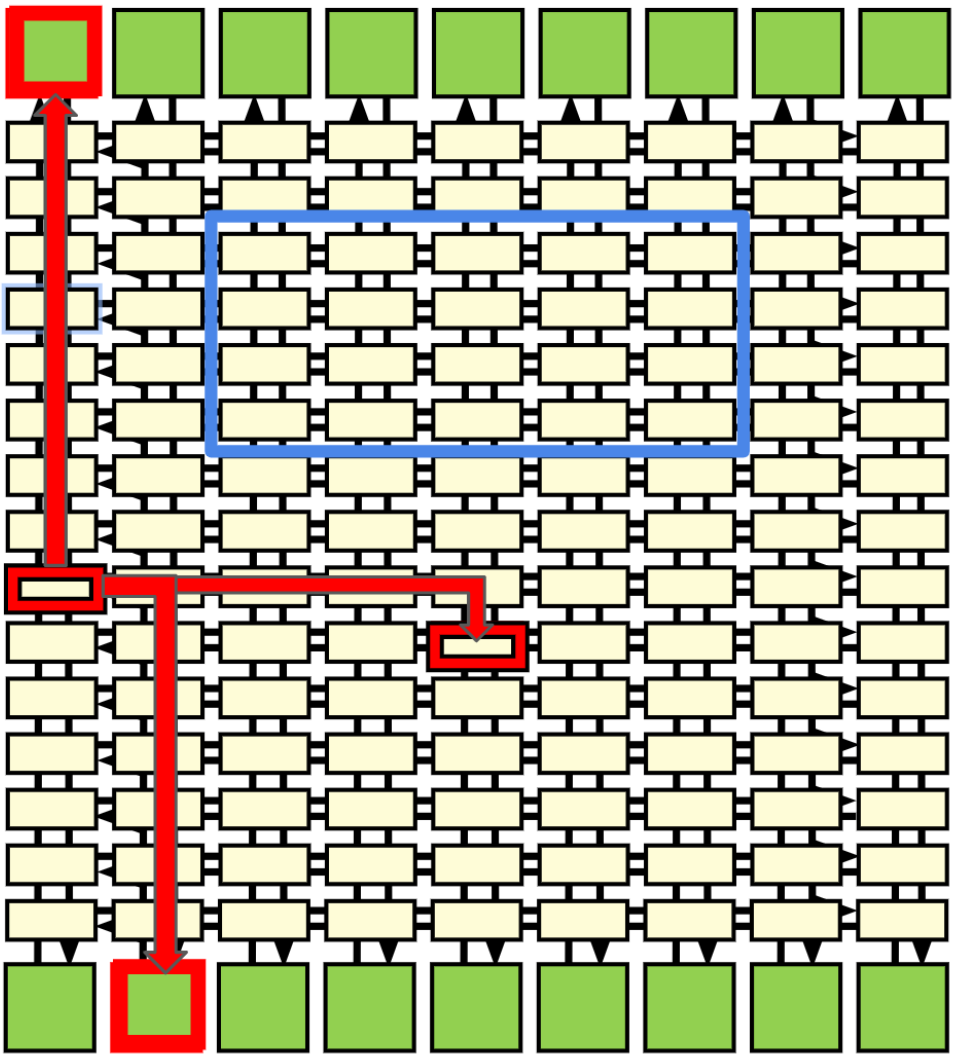

Figure 2.5: Tile Group Architecture of HammerBlade Kernel Dispatch

- **Unified Local Memory Sharing**: Tiles within the same group can share local memory resources.
- **Synchronization Barrier**: This barrier ensures that all tiles within a group reach a certain point in the execution process before proceeding further.

2.2.3 Grids

To facilitate the computation of complete results over extensive datasets, the CUDA-Lite Runtime employs Grids. Grids specify iterative invocations of Tile Groups, thereby extending the computing capabilities beyond the limitations of a single Tile Group. In essence, a Grid is a collection of tile groups that is used to dispatch a kernel. A grid can contain multiple tilegroup as it is a software construct.

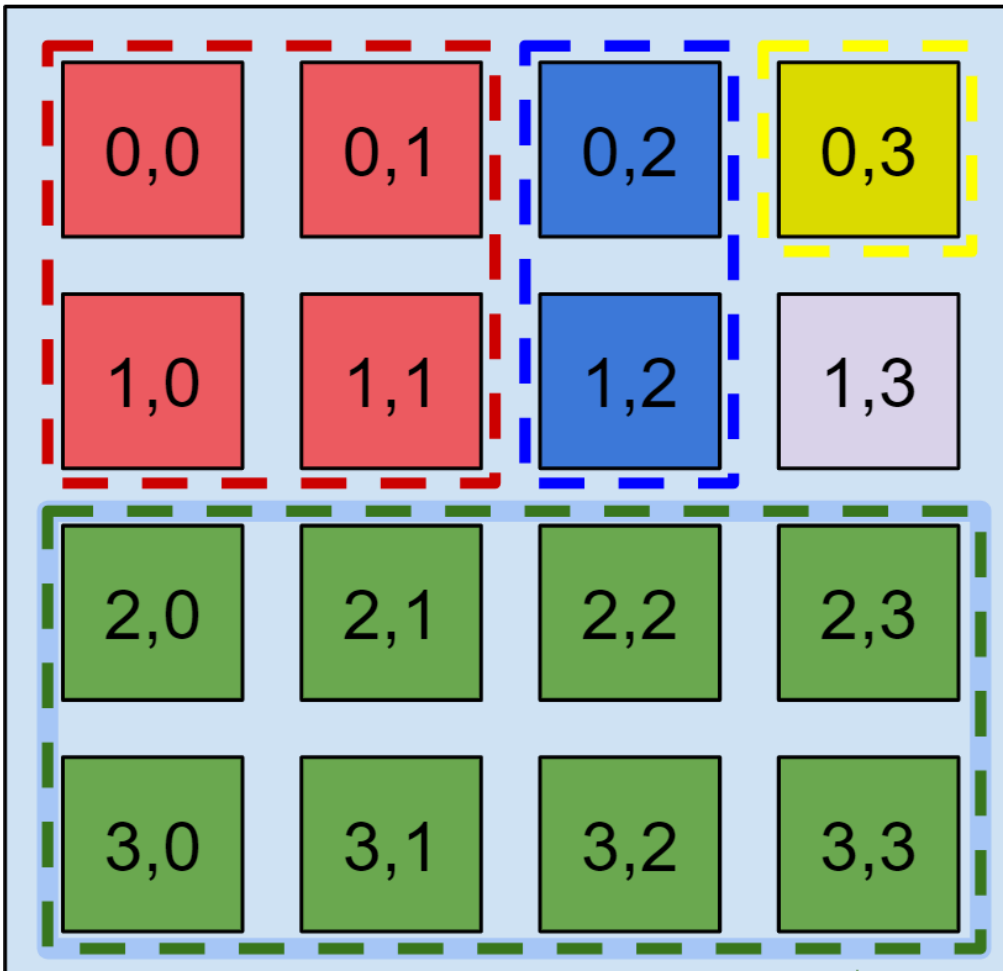

Figure 2.6: Grid Architecture of HammerBlade Kernel Dispatch

2.3 Survey on the GNU Toolchain and GCC

The GNU toolchain , a set of tools to develop software on Linux having the GNU Compiler Collection (GCC)[26] at its core. This comprehensive suite of programming tools provides a robust framework for compiling, debugging, and linking applications across various platforms. Key components include: GCC, GDB and the GNU Build System

2.3.1 GCC

The GNU compiler Collection(GCC) compiles C, C++, Objective-C, Fortran, Ada, and Go, among other programming languages; GNU Binutils handles binary files, object files, libraries, and executable files. GCC's extensible design allows for integrating third-party backends and frontends, facilitating support for new languages and architectures, which is critical in our quest to add auto-vectorization.

2.3.2 GDB

The GNU Debugger (GDB) allows developers to see what is happening inside another program while it executes or what another program was doing at the moment it crashed. GDB can be used to debug programs written in C, C++, and other languages by allowing the user to perform the following actions:

- GDB enables the user to start your programs with specific conditions.
- Set breakpoints in the program.
- Step through the program step by step.
- Inspect the state of the program.

2.3.3 GNU Build System

The GNU Build System, commonly referred to as Autotools, is a suite of programming tools designed to assist in making software packages portable across various Unix-like systems. Autotools include Autoconf, Automake, and Libtool. Here's an overview of each component and its purpose within the GNU Build System:

- **Autoconf**: Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of Unix-like systems. The primary goal of Autoconf is to create a script (typically named configure) that can tailor the software package to the local system.
- **Automake**: Automake complements Autoconf by generating portable Makefiles from a simpler template file (Makefile.am).Automake ensures that the Makefiles it generates

comply with the GNU Coding Standards, facilitating consistency and reliability across software projects.

- **Libtool**: Libtool is a programming tool that abstracts the process of using shared libraries away from developers. It provides a consistent, portable interface for creating static and shared libraries across various systems.

2.4 Survey on GPUs

The emergence of Graphics Processing Units (GPUs) has profoundly transformed the landscape of parallel computing, marking a significant departure from traditional, predominantly CPU-based computing paradigms. This evolution has underscored the substantial performance advantages of Single Instruction, Multiple Data (SIMD) [15] computation, which GPUs exploit to achieve remarkable efficiency and speed in processing parallel tasks.

GPUs are designed around a massively parallel architecture, where hundreds or thousands of smaller, efficient cores operate concurrently to perform computations. This design is inherently suited to SIMD, a parallel computing model in which the same operation is applied simultaneously to multiple data points. This approach is particularly effective for tasks that can be divided into parallel workloads, such as image processing, scientific simulations, and deep learning algorithms.

The performance uplift associated with SIMD computation on GPUs is a result of several factors. First, the parallel nature of SIMD allows for the efficient execution of operations over large data sets, drastically reducing the time required for computations that would take significantly longer on traditional CPUs. Second, GPUs are optimized for high throughput, meaning they can process large blocks of data in parallel, taking full advantage of their SIMD architecture. This makes them particularly well-suited for applications in machine learning, scientific computing, and real-time graphics rendering, where speed and efficiency are paramount.

Moreover, the continuous advancements in GPU technology, driven by companies like Nvidia and AMD, have further enhanced the capabilities of SIMD computation. These advancements include improvements in memory bandwidth, processing power, and energy efficiency, enabling more complex and demanding computational tasks to be performed more quickly and with greater precision. There are two major programming platforms for GPUs, they are CUDA from Nvidia and ROCm[49] from AMD.

2.4.1 CUDA

The CUDA [59] (Compute Unified Device Architecture) platform, developed by Nvidia, is a proprietary computing platform and programming model that allows direct access to Nvidia's

GPU's virtual instruction set and parallel computational elements. CUDA is designed explicitly for Nvidia GPUs. CUDA code is not natively compatible with GPUs from other manufacturers, such as AMD or Intel. The architecture is highly optimized for Nvidia's hardware, enabling efficient execution of parallel computations. CUDA Primarily supports programming in C, C++, and Fortran. CUDA also provides extensions for Python, allowing easy integration with popular scientific and machine-learning frameworks.

Figure 2.7: The architecture of the H100

2.4.2 ROCm

Developed by AMD, Radeon Open Compute (ROCm) is an open-source platform designed to enable GPU acceleration for computing. AMD's initiative is to provide an open-source foundation for high-performance computing (HPC) on GPUs. ROCm aims to be more open and versatile, supporting not just AMD GPUs but also designed to be portable across different architectures. ROCm includes support for other hardware through HIP (Heterogeneous-compute Interface for Portability), a tool that converts CUDA code to be compatible with AMD's platform. ROCm Supports a range of programming languages, including C, C++, and Fortran, through HIP and OpenCL. HIP allows developers to write code in a single source file that can run on AMD and Nvidia GPUs, enhancing portability and flexibility.

2.5 System Analysis

2.5.1 Existing System

The HammerBlade[32] architecture has computing setup across its 2048 cores, each powered by RISC-V vanilla cores. Despite its robust framework, the HammerBlade architecture encounters a pivotal limitation — its inherent lack of native vector load capabilities. This constraint restricts the architecture's efficiency in executing vector operations, which are integral to various computational tasks, particularly those involving linear algebra, signal processing, and machine learning algorithms. The absence of optimized vector processing mechanisms underscores a critical gap, necessitating the proposed enhancements in this research to unlock the architecture's computational prowess fully. This gap highlights the need for the proposed enhancements in this research.

RISC-V architecture has witnessed substantial progress, notably with compilers supporting the V (vector) [40] and P [39](packed SIMD instructions) extensions. These advancements represent a concerted effort to optimize code execution by leveraging the advanced instruction sets provided by the RISC-V architecture. Despite these strides, there exists a pronounced opportunity for further refinement of these compilers, especially in the context of burgeoning hardware capabilities. This research attempts to bridge the gap between the theoretical benefits of the RISC-V extensions and their practical implementation, thereby catalyzing tangible performance enhancements in many real-world applications.

Contrastingly, GPU computing, spearheaded by Nvidia [73], showcases unmatched performance benchmarks. However, this sector is markedly characterized by its closed-source nature, with the architectural intricacies remaining largely proprietary and shrouded in secrecy. This closed ecosystem starkly contrasts with the open-source ethos prevalent in CPU architecture development, where many open-source architectures have been made available for study and innovation. Despite this disparity, the GPU sector unequivocally demonstrates the substantial performance uplifts achievable through SIMD (Single Instruction, Multiple Data) architecture. This observation highlights the inherent value of SIMD paradigms in enhancing computational efficiency and underscores the potential benefits that could be realized within the HammerBlade architecture through the integration of advanced vector processing capabilities.

2.5.2 Proposed System

This thesis aims to comprehensively analyze the HammerBlade architecture, identifying inherent limitations while proposing strategic enhancements to harness the full spectrum of parallel computational possibilities. By drawing parallels with the advancements in GPU computing and the evolving compiler support for RISC-V extensions, this research endeavors to pave the way for significant performance improvements, transforming the theoretical capabilities of the HammerBlade architecture into a practical and efficient computing reality. Through this analytical lens, the thesis will explore the integration of native vector load capabilities and the refinement of compilers, setting a new benchmark in using RISC-V architecture for high-performance computing applications.

2.5.3 Problem Solving Approach

The principal objective of this research endeavor is to architecturally innovate within the HammerBlade software stack by seamlessly integrating components of Single Instruction Multiple Data (SIMD)[15] processing. The focal point of this architectural enhancement revolves around the evolution of memory access operations—transitioning from the conventional 32-bit load and store instructions to more expansive 128-bit, 16-byte wide loads and stores. This modification is designed to leverage the Packed SIMD model.

2.5.4 Hardware Extension of RISC-V Architecture

The implementation of a 16 Byte Wide Load in the HammerBlade chip's RISC-V core begins with an enhancement to support these larger instructions, modifying the existing architecture to manage larger data chunks efficiently. This improvement aims to increase data throughput and reduce bottlenecks in loading data, particularly in high-performance, multi-core systems. Furthermore, these hardware modifications can be later aligned the capabilities of the P extensions.

2.5.5 Refinement of the GCC RISC-V Compiler

Extensive refinements will be undertaken to optimize the GCC compiler for the revamped HammerBlade architecture, with the vectorized loads and stores. Modifying the compiler will involve the development of new optimization algorithms and strategies specifically designed to take advantage of the 16-byte wide load and enhanced vectorization capabilities. A key area of focus will be advancing the compiler's ability to vectorize code effectively. Vectorization includes devising novel vectorization techniques capable of automatically converting scalar operations into vectorized instructions, thereby fully harnessing the parallel processing power of the upgraded hardware.

2.5.6 Hardware and Software Co Design

Adopting a co-design approach, the project aims to develop hardware modifications and compiler optimizations simultaneously, ensuring a close alignment between the two. This strategy is crucial for maximizing the overall system performance, as it allows for the harmonization of hardware and software advancements. To facilitate this, the project will involve a cross-disciplinary collaboration within the Bespoke Sillicon Group (BSG) [31] team, bringing together experts in computer architecture and compiler design. This multidisciplinary team will collaboratively tackle the unique challenges that emerge from integrating advanced hardware modifications with sophisticated compiler technologies. This integrated approach is pivotal in ensuring that both the hardware and software components complement and enhance each other, leading to a more efficient and powerful system overall.

Chapter 3

System Design

3.1 Introduction

Designing a complex system, particularly one as intricate as a computing architecture, necessitates a strategic approach from system designers. The BSG Team undertakes this design work for the hardware development, and this thesis proposal delves into the compiler design. However, hardware design considerations are essential when designing a compiler. The system design process encompasses the deliberate planning and orchestration of various components and subsystems to achieve predefined goals efficiently. This task is compounded by several critical constraints that must be balanced to ensure the design's success, notably design cost, complexity, optimization opportunities, and die size considerations. Each of these factors plays a vital role in the overall feasibility and performance of the designed system. These design challenges are:

3.1.1 Design Cost and Complexity

The relationship between the complexity of an architecture and its design cost is almost directly proportional; as the complexity increases, so does the cost. This escalation is due to several factors, including increased development time, and the higher likelihood of requiring specialized expertise. Complex architectures might offer advanced features and capabilities, but they also present significant challenges in terms of verification, validation, and debugging, further contributing to the overall cost.

3.1.2 Missed Optimization Opportunities

Each decision made during the design phase can potentially lock in certain efficiencies or inefficiencies. The architecture's ability to handle parallel processing, power consumption, and heat dissipation are just a few areas where optimization can significantly impact over-

all system performance. Designers must employ a forward-thinking approach, anticipating future needs and potential bottlenecks, to integrate flexibility and scalability into the system.

3.1.3 Die Size Considerations

The size of the die is a pivotal factor in the cost and manufacturability of a chip. Increasing the die size not only directly impacts material costs but also affects yield rates during manufacturing, with larger dies being more susceptible to defects.[56] . We expect 25% increase in die size and it would be acceptable for a 4x performance uplift.

3.2 Vectorization Design Limitation

Vectorization represents a pivotal optimization in modern compilers, aiming to leverage the parallel processing capabilities of SIMD (Single Instruction, Multiple Data) architectures. By transforming scalar operations into vector operations, compilers can significantly enhance the execution speed of specific loops. The compiler will combine multiple scalar operations into a single vector operation. However, not all code is amenable to vectorization due to inherent limitations in its structure or logic. Understanding these limitations is crucial for developers aiming to optimize their code for SIMD execution. Below, we explore the primary constraints that impede the compiler's ability to vectorize code.

3.2.1 Countability

For a loop to be vectorized, its trip count, or the number of times the loop will execute, must be determinable at the entry to the loop during runtime. This requirement does not necessitate knowledge of the exact count at compile time, allowing for dynamic calculations as the program runs. However, if the compiler or the runtime system cannot ascertain the loop's trip count before its execution, vectorization cannot proceed. This limitation ensures that the vectorized operations can be safely and effectively applied across the loop iterations without the risk of overstepping the bounds of data structures or violating execution logic.

3.2.2 Single Entry Single Exit

A fundamental prerequisite for loop vectorization is a singular, predictable flow into and out of the loop. If a loop contains more than one exit point that depends on the data processed, it complicates the vectorization process. Such data-dependent exits introduce uncertainty in control flow, making it challenging for the compiler to guarantee that vectorized operations will maintain the program's semantic integrity. Consequently, loops with multiple data-dependent exit points are typically not considered for vectorization. i.e., there should be no break statements.

3.2.3 Straight-line code

The SIMD model thrives on executing the same operation across multiple data elements in parallel. This model imposes a restriction on the control flow within the loop; specifically, all iterations of the loop must follow an identical execution path. This uniformity ensures that each SIMD instruction operates correctly across all elements. Control flow divergences, such as those introduced by switch statements or if-conditions leading to different branches within the loop, disrupt this uniformity and, by extension, the applicability of SIMD vectorization. The requirement for straight-line code within the loop iterations is thus a critical consideration for vectorization. i.e., there should not be any switch statements.

3.2.4 The innermost loop of a nested Loop

Under standard circumstances, only the innermost loop is eligible for vectorization in nested loops. This limitation stems from the complexity introduced by multiple levels of looping, where outer loops may influence the conditions or data of the inner loops in ways that are incompatible with SIMD's parallel processing model. However, specific compiler optimizations can mitigate this restriction, such as loop unrolling or loop interchange techniques that transform the nested loops' structure, potentially allowing for vectorization at a different level or improving the efficiency of the innermost loop's vectorization.

3.3 Hardware Design Limitations

In computing hardware, the pursuit of integrating Single Instruction, Multiple Data (SIMD) capabilities within a constrained die area presents a series of design challenges. Balancing the implementation of SIMD functionalities against cost and die space limitations necessitates strategic design choices. This section outlines the limitations and design decisions undertaken to maximize SIMD potential while adhering to the constraints of lower cost and minimized die area. [56]

3.3.1 Load and Store

One of the pivotal enhancements in our hardware design involves the revamping of load and store operations to accommodate wider, 16-byte loads and stores. This decision is rooted in the understanding that, although load and store instructions constitute only about 20% of a program's instructions, they account for approximately 80% of execution time. By quadrupling the memory access rate, we anticipate a significant improvement in program execution times, addressing one of the primary bottlenecks in computing performance[17]. There is a question of why only 16 Byte parallelism? The number of wires for the ruche network[48] required increases exponentially with number of bits. this 16 byte change in the die structure would the router size by 4x , which takes up 4% die space currently, the new router would take up 16% die space.

3.3.2 Striding

Striding [75]refers to the gap between successive memory accesses measured in bytes. While the architecture could support strides larger than the register size, indicating additional space between elements, it has been decided not to implement striding capabilities to conserve die space. This decision reflects a trade-off between flexibility in memory access patterns and the imperative to minimize hardware complexity and cost.

Figure 3.1: Non Unit Strided Memory Access, with stride of two

3.3.3 Registers

The vectorized load and store however will use same a0-a15 registers as the normal load stores as in packed SIMD.

3.3.4 Register Set

For vectorized load and store operations, the decision has been made to utilize the existing a0-a15 registers, aligning with packed SIMD paradigms. This choice ensures that the extension of SIMD capabilities does not necessitate a wholesale redesign of the register architecture, thereby conserving resources and simplifying the implementation.

In this model we have a choice to either enforce a register set, in this case:

- a0-a3
- a4-a7
- a8-a11
- a12-a15

Or we can choose a sliding window where any contiguous set of available registers can be used as vector loads. In this case :

- a0-a3
- a1-a4
- a2-a5
- a3-a6

- a4-a7
- a5-a8
- a6-a9
- a7-a10
- a8-a11
- a9-a12
- a10-a13
- a11-a14
- a12-a15

Despite the apparent flexibility offered by the sliding window method, it was ultimately decided to adhere to fixed register groups. This decision was made to avoid the significant expansion of the crossbar size that would be required to support the sliding window approach, which would not proportionally benefit performance given the constraints. The new crossbar will increase die size by approximately 10%.

3.3.5 ALU

In alignment with minimizing die size, there will be no vectorization of the Arithmetic Logic Unit (ALU) or the register file. This decision underlines a strategic prioritization of SIMD enhancements that can be realized within the existing spatial and cost constraints, focusing on areas where the most substantial performance gains can be achieved without a corresponding increase in hardware complexity. Almost all ALU operations take one cycle; we do not need to vectorize ALU operations now. However, future implementations will also aim to vectorize the ALU.

3.4 Benchmark Selection

We will select memcpy, addarray, and matrix multiply as benchmarks for evaluating the impact of vectorized loads and stores on the HammerBlade architecture, a comprehensive approach to demonstrate the potential performance gains from SIMD vectorization. These benchmarks cover a broad spectrum of common computational patterns, from simple memory operations to more complex arithmetic and matrix operations, providing a nuanced understanding of how vectorization enhancements affect overall system performance[31]. The benchmarking details are as follows:

3.4.1 Memcpy Benchmark

- **Objective**: To measure the performance improvement in memory copying operations due to vectorized load and store instructions. Memcpy operations are essential for a wide range of applications, making this benchmark a crucial indicator of the efficiency of memory bandwidth utilization.

- **Methodology**: Execution times for copying varying sizes of data blocks will be recorded and compared before and after the implementation of vectorized instructions. This comparison will highlight the direct impact of SIMD enhancements on memory transfer rates.

3.4.2 AddArray Benchmark

- **Objective**: To evaluate the performance gains in simple arithmetic operations, specifically array addition, which can benefit from parallel execution through SIMD instructions. This benchmark will showcase the improvements in computational throughput for basic arithmetic tasks.
- **Methodology**: Two large arrays will be added element-wise, with execution times measured before and after vectorization. This benchmark will particularly benefit from the vectorized load and store operations to process multiple data points simultaneously.

3.4.3 Matrix Multiply Benchmark

- **Objective**: To assess the enhancements in executing more complex arithmetic operations, such as matrix multiplication, which is a fundamental operation in numerous scientific and machine learning applications. Matrix multiplication can significantly benefit from SIMD optimization due to its repetitive nature and the high volume of arithmetic operations.
- **Methodology**: Execution times for multiplying matrices of various sizes will be compared pre and post-vectorization. This benchmark will test the ability of the architecture to leverage SIMD for both loading/storing large data sets and performing parallel operations efficiently.

3.4.4 HammerBench Benchmark

- **Objective**: To provide a comprehensive performance analysis across a wide range of computational tasks and scenarios beyond the three primary benchmarks. This suite will help identify other areas where SIMD vectorization may contribute to performance improvements.
- **Methodology**: The suite will be executed in its entirety, with performance metrics collected and analyzed against baseline measurements from the architecture prior to SIMD enhancements. This broad evaluation will help quantify the overall impact of vectorization on system performance.

3.5 Instructions And Uses

This study embarks on an exploration of augmenting the existing RISC-V Vanilla cores within the HammerBlade architecture, which traditionally employ the RV32I[63] instruction set, with a custom extension specifically designed to enhance vectorized load and store operations. The initiative to integrate these advanced vectorization capabilities represents a significant architectural overhaul, aimed at substantially improving the efficiency and performance of data handling and computational tasks. Spearheaded by the architecture researchers at the Bespoke Silicon Group (BSG Team).[99]

3.5.1 Data transfer Instruction

We will add the following instructions to the RV32 Vanilla cores to enable SIMD load and store.

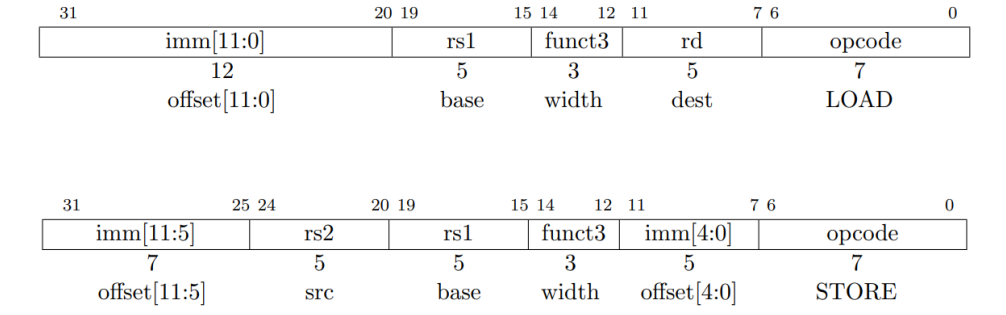

Figure 3.2: the Architecture Memory Instruction in RV32I [99]

Load

vle register_set, address

The vectorized load instruction will load 16 bytes from memory simultaneously into the 4 registers in the register set.

Description of the Vectorized Load Instruction

- **Operation**: The vectorized load instruction fetches a contiguous block of 16 bytes from a specified memory location. It then distributes this block across four registers, with each register receiving a 4-byte segment of the total data simultaneously.
- **Data Ordering**: The instruction adheres to little-endian data ordering, meaning that the least significant byte of the data block is stored in the lowest memory address (and consequently loaded into the lowest byte of the first register in the set), progressing to the most significant byte at the highest address (loaded into the highest byte of the last register in the set).

Consider a scenario where a vectorized load instruction is issued to load 16 bytes from a specific memory location 0x2000 into the registers a0 to a3, with each register set to hold 4 bytes of data. If the memory block starting at the address contains the sequence 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 (where 01 is the least significant byte and 10 is the most significant byte), the data would be loaded into the registers as follows:

vle a0, 2000
This instruction will now load memory locations

- 2000-2003 into a0 and would receive 01 02 03 04.
- 2004-2007 into a1 and would receive 05 06 07 08.
- 2008-200B into a2 and would receive 09 0A 0B 0C.
- 200C-200F into a3 and would receive 0D 0E 0F 10.

Store

vse register_set, address

The vetorized store instruction will store 16 bytes simultaneously from the chosen register set into 16 bytes in memory starting at the specified location and incrementing from there The load will be in little-endian order.

Description of the Vectorized Store Instruction

- **Operation**: Upon execution, the vectorized store instruction sequentially transfers the contents of four selected registers, each holding 4 bytes of data, into a 16-byte memory block. This operation enables the simultaneous storage of multiple data elements, leveraging the SIMD capabilities to enhance the throughput and efficiency of data storage operations.
- **Data Ordering**: Consistent with the vectorized load instruction, the vectorized store instruction adheres to little-endian data ordering. This means that the least significant byte (LSB) of the first register in the set is stored in the lowest memory address, and the most significant byte (MSB) of the last register is stored in the highest address of the 16-byte block. This ordering facilitates compatibility with little-endian systems, ensuring that data is stored in a manner that matches the system's native data representation.

Assuming a scenario where the registers a0 to a3 are used to store 16 bytes of data into memory, with the registers containing the following data:

- Register a0: 01 02 03 04
- Register a1: 05 06 07 08
- Register a2: 09 0A 0B 0C
- Register a3: 0D 0E 0F 10

vse a0, 2000

This instruction will now load memory locations

- a0 will be stored in 2000-2003
- a1 will be stored in 2004-2007
- a2 will be stored in 2008-200B
- a3 will be stored in 200C-200F

The vectorized store instruction would write this data to a specified memory location 0x2000 in little-endian order, resulting in a memory block being populated as follows: 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10, with 01 being stored at the lowest specified memory address and 10 at the highest.

Chapter 4

GNU Toolchain Setup and Analysis

This chapter aims to demystify the process of setting up the GNU Toolchain, namely GDB and GCC, offering step-by-step guidance. It will cover the installation of the toolchain components, the configuration of the development environment, and the integration of additional utilities that enhance the toolchain's functionality. This chapter will also discuss GCC in detail, especially the register allocation process and the files to be modified to add vectorization support. Understanding these details is essential in understanding the modifications done in the next chapter to support vectorization and to support future modification.

4.1 Setup

- Clone the tool chain repository,the official RISC-V GNU-toolchain repo.

```
git clone https://github.com/riscv-collab/riscv-gnu-toolchain.git
```

- Clone the sub modules into the respective folders.

```
git clone https://github.com/linsinan1995/riscv-gcc.git gcc
git clone https://github.com/linsinan1995/riscv-binutils-gdb.git binutils
```

- Configure the project and build.

```
./configure --prefix=/opt/riscv --enable-multilib
sudo make linux
```

- The executable for gcc will now be placed in the prefix directory.

4.2 Understanding the Tool chain

For SIMD Vectorization there are two major parts: The compiler that converts scalar accesses to vector accesses and the assembler that converts the assembly language to machine code.

4.2.1 Assembler

The assembler is a one to one translation of opcodes to machine code. The major parts of the assembler are the match and mask used to match the opcode produced by the compiler.

4.2.2 Compiler

The compiler is one of the most complex software in the GNU toolchain and the main focus of this research work. The front end generates a target-independent representation (GIMPLE)[26] that is used when optimizing the code, and the result is passed to the back end that converts it to its own representation (RTL) and generates the code for the program.

The back end's internal representation for the code consists of a ldoubly inked list of objects called "insns". An insn corresponds roughly to an assembly instruction, but there are insns representing labels, dispatch tables for switch statements, etc. Each insnsn is constructed as a tree of expressions
For example,

(reg:m n)
is an expression representing a register access, and

(plus:m x y)
represents adding the expressions x and y. An insn adding two registers may combine these as

(set (reg:m r0) (plus:m (reg:m r1) (reg:m r2)))
The m in the expressions denotes a machine mode that defines the size and representation of the data object or operation in the expression.

Backend Processing

The backend starts by converting GIMPLE to RTL,
for the following small GIMPLE function:

```
add (long long unsigned int * a, long long unsigned int * b, long long unsigned int * c)
{
```

```
    register volatile unsigned int p1 __asm__ (*a7);
    p1 = 5;
    {
        short int i;
        i = 0;
        goto <D.1868>;
        <D.1867>:
        _1 = (unsigned int) i;
        _2 = _1 * 8;
        _3 = a + _2;
        _4 = *_3;
        _5 = (unsigned int) i;
        _6 = _5 * 8;
        _7 = b + _6;
        _8 = *_7;
        _9 = (unsigned int) i;
        _10 = _9 * 8;
        _11 = c + _10;
        _12 = _4 + _8;
        *_11 = _12;
        p1.0_13 = p1;
        _14 = p1.0_13 + 1;
        p1 = _14;
        i.1_15 = i;
        i.2_16 = (unsigned short) i.1_15;
        _17 = i.2_16 + 1;
        i = (short int) _17;
        <D.1868>:
        if (i <= 31) goto <D.1867>; else goto <D.1869>;
        <D.1869>:
    }
}
```

The backend of the compiler produces an initial unoptimized RTL code. Then runs multiple passes through the RTL code and generates a final optimized RTL code.

```
(note 1 0 7 NOTE_INSN_DELETED)
(note 7 1 43 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 43 7 5 NOTE_INSN_PROLOGUE_END)
(note 5 43 6 NOTE_INSN_FUNCTION_BEG)
(insn 6 5 23 (set (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80])
    (const_int 0 [0])) "main.c":4:40 136 {*movsi_internal}
    (expr_list:REG_EQUAL (const_int 0 [0])
    (nil)))
(insn 23 6 22 (set (reg:SI 28 t3 [90])
    (const_int 256 [0x100])) "main.c":6:6 136 {*movsi_internal}
    (expr_list:REG_EQUIV (const_int 256 [0x100])
    (nil)))
(code_label 22 23 10 2 (nil) [1 uses])
(note 10 22 13 [bb 3] NOTE_INSN_BASIC_BLOCK)
(insn 13 10 12 (set (reg:SI 14 a4 [86])
    (plus:SI (reg/v/f:SI 11 a1 [orig:82 b ] [82])
    (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80]))) "main.c":7:22 3 {addsi3}
    (nil))
```

```
(insn 12 13 37 (set (reg:SI 13 a3 [85])
    (plus:SI (reg/v/f:SI 10 a0 [orig:81 a ] [81])
    (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80]))) "main.c":7:17 3 {addsi3}
    (nil))
(insn 37 12 38 (set (reg:SI 16 a6 [orig:88 MEM[base: _31, offset: 0B] ] [88])
    (mem:SI (reg:SI 13 a3 [85]) [1 MEM[base: _31, offset: 0B]+0 S4 A64])) "main.c":7:20 136
        {*movsi_internal}
    (nil))
(insn 38 37 39 (set (reg:SI 17 a7 [ MEM[base: _31, offset: 0B]+4 ])
    (mem:SI (plus:SI (reg:SI 13 a3 [85])
    (const_int 4 [0x4])) [1 MEM[base: _31, offset: 0B]+4 S4 A32])) "main.c":7:20 136
        {*movsi_internal}
    (expr_list:REG_DEAD (reg:SI 13 a3 [85])
    (nil)))
(insn 39 38 40 (set (reg:SI 6 t1 [orig:89 MEM[base: _30, offset: 0B] ] [89])
    (mem:SI (reg:SI 14 a4 [86]) [1 MEM[base: _30, offset: 0B]+0 S4 A64])) "main.c":7:20 136
        {*movsi_internal}
    (nil))
(insn 40 39 11 (set (reg:SI 7 t2 [ MEM[base: _30, offset: 0B]+4 ])
    (mem:SI (plus:SI (reg:SI 14 a4 [86])
    (const_int 4 [0x4])) [1 MEM[base: _30, offset: 0B]+4 S4 A32])) "main.c":7:20 136
        {*movsi_internal}
    (expr_list:REG_DEAD (reg:SI 14 a4 [86])
    (nil)))
(insn 11 40 21 (set (reg:SI 14 a4 [84])
    (plus:SI (reg/v/f:SI 12 a2 [orig:83 c ] [83])
    (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80]))) "main.c":7:14 3 {addsi3}
    (nil))
(insn 21 11 16 (set (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80])
    (plus:SI (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80])
    (const_int 8 [0x8]))) "main.c":6:6 3 {addsi3}
    (nil))
(insn 16 21 41 (set (reg:DI 16 a6 [87])
    (plus:DI (reg:DI 16 a6 [orig:88 MEM[base: _31, offset: 0B] ] [88])
    (reg:DI 6 t1 [orig:89 MEM[base: _30, offset: 0B] ] [89]))) "main.c":7:20 340 {*add64_rvp}
    (expr_list:REG_DEAD (reg:DI 6 t1 [orig:89 MEM[base: _30, offset: 0B] ] [89])
    (expr_list:REG_EQUIV (mem:DI (reg:SI 14 a4 [84]) [1 MEM[base: _29, offset: 0B]+0 S8 A64])
    (nil))))
(insn 41 16 42 (set (mem:SI (reg:SI 14 a4 [84]) [1 MEM[base: _29, offset: 0B]+0 S4 A64])
    (reg:SI 16 a6 [87])) "main.c":7:14 136 {*movsi_internal}
    (expr_list:REG_DEAD (reg:SI 16 a6 [87])
    (nil)))
(insn 42 41 24 (set (mem:SI (plus:SI (reg:SI 14 a4 [84])
    (const_int 4 [0x4])) [1 MEM[base: _29, offset: 0B]+4 S4 A32])
    (reg:SI 17 a7 [+4 ])) "main.c":7:14 136 {*movsi_internal}
    (expr_list:REG_DEAD (reg:SI 17 a7 [+4 ])
    (expr_list:REG_DEAD (reg:SI 14 a4 [84])
    (nil))))
(jump_insn 24 42 46 (set (pc)
    (if_then_else (ne (reg:SI 15 a5 [orig:80 ivtmp.16 ] [80])
    (reg:SI 28 t3 [90]))
    (label_ref:SI 22)
    (pc))) "main.c":6:6 181 {*branchsi}
(int_list:REG_BR_PROB 1040190860 (nil)) -> 22)
(note 46 24 44 [bb 4] NOTE_INSN_BASIC_BLOCK)
(note 44 46 45 NOTE_INSN_EPILOGUE_BEG)
(jump_insn 45 44 47 (simple_return) "main.c":12:1 257 {simple_return}
(nil) -> simple_return)
(barrier 47 45 35)
```

```
(note 35 47 36 NOTE_INSN_DELETED)
(note 36 35 0 NOTE_INSN_DELETED)
```

Then the backend uses the target architecture's instructions that are described in riscv.md and rvp.md using instruction patterns to convert the RTL code to assembly language, which is a direct conversion from insn to target instruction.

4.3 Register Allocation

The Integrated Register Allocator (IRA) is a system designed to manage and optimize use the registers. It works by organizing these registers into regions and then using a method called graph coloring[61], starting from the top and moving down through these regions. This method is based on a specific algorithm known as the Chaitin-Briggs algorithm[9].

The IRA handles multiple tasks in register allocation:

- Combining registers when possible[24] (register coalescing).
- Splitting the use of registers to make them more efficient (register live range splitting).
- Selecting the best physical register available for the task (choosing a better hard register).

Additionally, the IRA makes decisions about which physical register to use, trying to use the ones that will cost less in terms of computing power.[26]

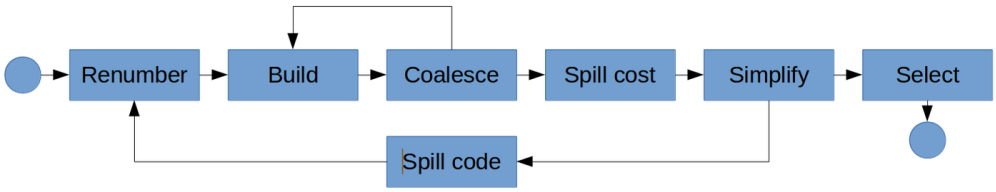

Figure 4.1: The register allocation process [26]

4.3.1 Major IRA notions

- **Region**: This refers to a specific part of the Control Flow Graph (CFG) where the IRA applies graph coloring, a technique for register allocation based on the Chaitin-Briggs algorithm. IRA can handle any arrangement of nested CFG regions, which collectively form a hierarchical structure, typically with the entire function serving as the root region and natural loops as subordinate regions. These regions are represented by a data structure known as a loop_tree_node.

- **Allocno**: Representing the lifespan of a pseudo-register within a region, an allocno's characteristics include its corresponding pseudo-register number, allocno class, and its interactions with other allocnos and hard registers. Key attributes of an allocno for understanding the allocation algorithm include:
  - **Live Ranges**: Lists of program points indicating where the allocno is active, crucial for identifying conflicts and transforming IRA's representation for efficient allocation.
  - **Hard-Register Costs**: A matrix indicating the cost of assigning each hard register to the allocno, factoring in callee-clobbered register costs and move instruction efficiencies to determine the most cost-effective assignments.A callee-clobbered register refers to a type of CPU register whose value is not guaranteed to be preserved across function calls within a program.
  - **Conflict Hard-Register Costs**: Similar to hard-register costs, this matrix is used to refine the final cost calculation for unassigned allocnos, indicating preferences based on potential move instruction eliminations.

- **Allocno Class**: Each register, or allocno, is allocated based on a specific register class, meaning only hard registers from this class are assignable to it. However, often only a smaller, more profitable subset of these hard registers is actually considered for assignment. This selection is further refined by our adaptation of the Chaitin-Briggs algorithm, ensuring the sets of assignable hard registers form a non-overlapping hierarchy.

- **Pressure Class**: Within the set of all register classes, a pressure class includes those hard registers available for allocation. This set is predefined in the machine-description file based on specific criteria. The concept of register pressure, which influences IRA decisions, is calculated exclusively for these pressure classes, affecting how allocation regions are formed.

- **Cap**: For pseudo-registers not active in a region but present in a nested one, IRA introduces a cap allocno in the outer region to facilitate allocation.

- **Copy**: Connections between allocnos, primarily established through move instructions, influence hard-register cost adjustments to favor uniform hard-register assignment across connected allocnos. IRA employs copies not just for coalescing registers but also for aligning operands to machine description constraints and optimizing instruction inputs and outputs.

4.3.2 IRA Major Passes

Building IRA Internal Representation

- **Region and Allocno Construction**: Initially, IRA delineates regions and generates allocnos, initializing most of their characteristics as detailed in the file ira-build.cc.

- **Allocno Class Identification and Cost Calculation**: For each allocno, IRA determines its class and computes the initial costs associated with memory and each hard register within its class. This step is crucial for establishing the foundation for register allocation decisions found in ira-cost.c.

- **Live Range and Register Pressure Analysis**: IRA calculates the live ranges for each allocno, assesses register pressure across each pressure class within all regions, and identifies hard registers in conflict for each allocno. Additionally, it records information regarding calls that occur within an allocno's lifespan (in ira-lives.cc).

- **Optimization for Speed**: To enhance IRA's efficiency, regions experiencing low register pressure are excluded from further consideration (ira-build.cc).

- **Information Propagation**: Accumulated information from allocnos in lower regions is propagated to corresponding allocnos in upper regions, ensuring that allocno attributes reflect comprehensive, integrated data across nested regions (ira-build.cc).

- **Cap Creation**: Special allocnos, known as caps, are created for pseudo-registers that are inactive in a region but active in nested regions. This step facilitates the allocation process across different levels of the region hierarchy (ira-build.cc).

- **Conflict and Copy Management**: With the live ranges of allocnos and their classes established, IRA identifies allocnos that conflict with each other. Allocnos that are alive at the same time (i.e., their live ranges overlap) are said to conflict with each other because they cannot be assigned to the same register without causing data corruption. These conflicting allocnos are organized either as a bit vector or an array of pointers, based on which approach is more efficient. At this stage, IRA also generates copies between allocnos to optimize register usage by minimizing move instructions and aligning allocno preferences (ira-conflicts.cc).

Graph Coloring

Now IRA has all necessary info to start graph coloring process. It is done in each region on top-down traverse of the region tree (file ira-color.cc).

- **Identification of Profitable Hard Registers**: Initially, IRA determines the most beneficial hard registers for each allocno based on its class. Often, callee-saved hard registers are deemed advantageous, especially for allocnos that persist through calls. If the set of profitable hard registers doesn't naturally form a hierarchical tree structure, IRA employs approximations to establish this hierarchy, aiding in the assessment of straightforward allocno colorability. Although this approximation scenario is infrequent, it's crucial for streamlining the coloring process.

- **Allocno Stacking for Coloring**: Leveraging Briggs' optimistic coloring approach, which marks a significant advancement over Chaitin's method, IRA strategically stacks allocnos without necessitating immediate spilling. The order in which allocnos are stacked can influence the allocation outcome, guided by heuristics designed to optimize this sequence. A notable tactic involves grouping allocnos into "threads" — sets of non-conflicting, colorable allocnos linked by copies — and stacking these threads sequentially to enhance the probability of eliminating copies by assigning identical hard registers to connected allocnos.

- **Handling Intersected Register Classes**: IRA incorporates a modified Chaitin-Briggs algorithm capable of addressing allocnos with intersecting register classes, utilizing the hard register set hierarchy to deduce allocno colorability. This modification allows for a nuanced understanding of allocno conflicts and potential register assignments, even in complex scenarios like those involving general and specific hard register assignments.

- **Allocno Assignment and Spilling**: As allocnos are popped from the stack, IRA tries to assign them appropriate hard registers. If direct assignment proves challenging or

if separating coalesced allocnos is more advantageous, IRA adjusts its strategy accordingly, aiming for the most cost-effective allocation. This includes spilling allocnos when necessary, prioritizing the minimization of overall allocation costs.

- **Optimization Post-Coloring**: Following the initial coloring and allocno assignment, IRA seeks to refine the allocation from a cost perspective. This may involve reallocating hard registers by spilling certain allocnos to free up resources for others, thus reducing the total allocation cost.

- **Cost Adjustments for Subregions**: After completing allocno assignments within a region, IRA updates the hard register and memory costs for allocnos in nested subregions. This adjustment accounts for the potential costs associated with data transfers at the boundaries between regions and subregions. When employing the default regional allocation algorithm, IRA extends these assignments to allocnos in regions with lower register pressure, ensuring an efficient distribution of available hard registers.

Spill/restore code moving

During the allocation process, the Integrated Register Allocator (IRA) traverses the region tree in a top-down manner. This approach, while systematic, occasionally overlooks opportunities for more optimal allocations due to the IRA's lack of foresight into the lower levels of the region tree. To address this, IRA incorporates a straightforward optimization strategy aimed at enhancing allocation within a region that includes subregions and is itself nested within another region.

This optimization specifically targets scenarios where allocnos in a subregion are spilled. If spilling the corresponding allocno in the overarching region proves to be profitable, the IRA proceeds with this action. The rationale behind this strategy is to perform adjustments that improve the overall allocation by reducing the need for spill and restore operations, which can be costly in terms of execution time.

The optimization operates through a simple iterative algorithm that repeatedly seeks out and executes profitable transformations as long as they remain viable.

Code Change

After the graph coloring phase, situations may arise where two allocnos representing the same pseudo-register— one located outside a region and the other inside— are assigned to different locations, either in hard-registers or memory. To address this discrepancy, the

Integrated Register Allocator (IRA) implements a code modification strategy during its top-down traversal of regions, as detailed in the ira-emit.cc file. This strategy involves creating a new pseudo-register within the region and generating additional code to facilitate the transfer of allocno values across the region's boundaries.

In more complex scenarios, such as when a swap between values stored in two hard-registers is necessary, IRA may introduce a new allocno specifically for transferring allocno values. This new allocno is initially marked as spilled, indicating that its value needs to be moved to or from memory during execution. Despite these conditions, IRA proceeds to generate the necessary pseudo-register and the corresponding move instructions at the region borders, even if both allocnos were originally assigned to the same hard-register. This approach is taken to mitigate the potential impact of the reload pass, which might spill a pseudo-register for various reasons. By maintaining one allocno in a hard-register, the adverse effects of spilling are reduced, as it is generally more advantageous than spilling both allocnos.

Should the reload phase not alter the allocation of these two pseudo-registers, any superfluous move operations introduced by IRA are likely to be eliminated during post-reload optimizations, ensuring that only necessary code adjustments are retained. Additionally, IRA refrains from generating move instructions for allocnos assigned to the same hard register under specific conditions—namely, when employing the default regional allocation algorithm and the register pressure within the region for the corresponding pressure class is below the threshold of available hard registers for that class.

Moreover, IRA implements further optimizations aimed at eliminating unnecessary store operations and minimizing code duplication at the region borders.

IR Flattening

Following the code modification phase, the Integrated Register Allocator (IRA) proceeds to consolidate its internal representation of multiple regions into a singular region framework, a process documented within the ira-build.cc file and referred to as IR flattening. This procedure is designed to streamline the internal structure of IRA, making it more efficient for subsequent operations. Although this IR flattening process is inherently more complex than simply rebuilding the internal representation from scratch, it offers significant speed advantages, making it a preferred approach within the IRA's optimization strategy.

Once the internal representation has been flattened, the IRA shifts its focus towards reallocating hard registers to allocnos that were previously spilled. This reallocation effort is facilitated by a straightforward and efficient priority coloring algorithm, as detailed in the function ira_reassign_conflict_allocnos within the ira-color.cc file. During this phase, new allocnos that emerged as a result of the earlier code change pass are also considered for hard register assignments. This step is critical as it allows for the optimization of register usage,

potentially reducing the need for spilling and restoring operations and thus enhancing the overall execution efficiency of the compiled code.

4.3.3 Reload Pass

In the final stage of the optimization process, the Integrated Register Allocator (IRA) hands over control to the reload pass, a crucial phase that fine-tunes the register allocation by addressing any remaining issues with spilled pseudo-registers. The reload pass engages with IRA to enhance its decision-making, utilizing several functions detailed in the ira-color.cc file. This focuses on three main areas:

- **Stack Slot Sharing for Spilled Pseudo-Register**: Utilizing the detailed conflict information provided by IRA regarding pseudo-registers, the reload pass can optimize memory usage by sharing stack slots among spilled pseudo-registers. This strategy is predicated on understanding which pseudo-registers conflict with each other, allowing the reload pass to allocate stack space more efficiently by avoiding conflicts in stack slot assignments.

- **Reassigning Hard Registers to Spilled Pseudo-Register**: At the conclusion of each iteration within the reload phase, an effort is made to reassign hard registers to all spilled pseudo-registers. This process is informed by IRA's comprehensive analysis of register usage, aiming to minimize the number of pseudo-registers that must be spilled to memory. By dynamically reassessing the allocation of hard registers, the reload pass seeks to optimize the use of available registers, potentially reducing execution time and improving overall program performance.

- **Optimal Hard Register Selection for Spilling**: When deciding which hard register to spill, the reload pass benefits from IRA's insights into pseudo-register live ranges and the current register pressure in relevant sections of the code. This information allows the reload pass to make more informed decisions about which hard registers to spill, prioritizing the spilling of registers that will have the least impact on the program's execution efficiency.

Chapter 5

GNU Vectorization Modifications

The chapter explores a contribution of this thesis project to the enhancements and modifications within the GNU toolchain for optimizing vectorization. The first half of the chapter addresses the changes made to Binutils, focusing on converting opcodes to machine code. The second half discusses the modifications implemented to support auto-vectorization.

5.1 Target

The target for our current GCC vectorization program is to convert 4 Scalar loads/Stores to a vector load/Store

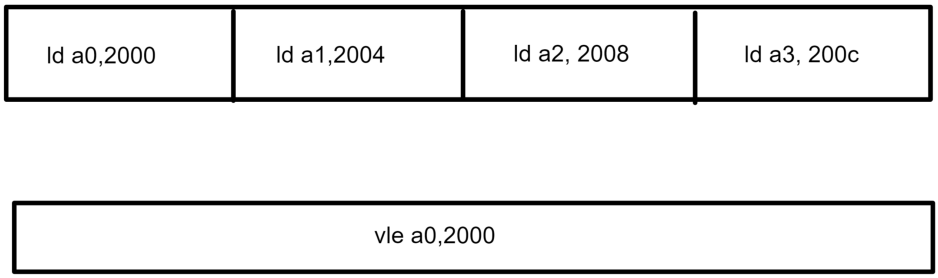

Figure 5.1: Target Vectorization

5.2 Binutils Modifications

Before we are able to modify the GCC compiler we must first add support for the two Vector load and store instructions to binutils. The following instruction pattern were added to binutils.

```
vle     rd rs1      imm12 14..12=7 6..2=0x00 1..0=3
vst    imm12hi rs1 rs2 imm12lo 14..12=7 6..2=0x08 1..0=3
```

5.2.1 Define Opcode Mapping

The riscv-opc.h file in the $GNUHOME/binutils/include/opcode directory is for defining opcode mappings for the RISC-V architecture. The modifications introduce new #define directives for MATCH_vle and MATCH_vst, along with their corresponding masks MASK_vle and MASK_vst. These directives establish the binary patterns that the assembler will use to recognize and encode the vle (vector load) and vst (vector store) instructions.

```
DECLARE_INSN(vst, MATCH_vst, MASK_vst)
DECLARE_INSN(vle, MATCH_vle, MASK_vle)

#define MATCH_vle 0x7003
#define MASK_vle 0x707f

#define MATCH_vst 0x7023
#define MASK_vst 0x707f
```

implement Opcode

The riscv-opc.c file contains the implementation details for opcode handling, including instruction classes, syntax, and binary encoding rules. Modifying this file to include the vle and vse instruction pattern is essential for enabling the assembler to process these instructions correctly. The following modifications were made to the file riscv-opc.c found in $GNUHOME/binutils/include/opcode.

```
    {"vle", 32, INSN_CLASS_I, "d,o(s)", MATCH_vle, MASK_vle, match_opcode, INSN_DREF|INSN_4_BYTE },
```

Compilation of binutils with the modifications will yeild an assembler capable of translating vle and vse into machine code that would run on the machine once it is built by the BSG team.

5.3 GCC Vectorization

GCC vectorization can be done at multiple passes.

- Early Passes: Concentrate on more aggressive optimizations that may depend on specific details of the program's structure but are still generally independent of the target architecture. Advanced vectorization strategies, loop transformations for better vectorization, and data locality improvements are typical in this phase.
- Late Passes: Target low-level optimizations that are closely tied to the target architecture, including instruction selection, scheduling, and final adjustments for vectorized code to align with the specifics of the SIMD instructions supported by the target processor.

5.3.1 Peephole Optimization

Peephole optimization operates on a small set of instructions within a sliding window. It is a late pass optimization. Peephole optimization is applied towards the end of the compilation process, after the initial code generation phase. Because it operates on a small window of instructions at a time, peephole optimization is relatively simple and fast. Due to this,

it is easy to confirm program correctness. The peephole optimization given in the figure transform 4 sequential set register from memory operand into a single nested set register from memory operand.

This research uses peephole optimization to catch stray loads and stores and convert them to vectorized loads.

peephole.md

Add the peephole definition to peephole.md.

```
(define_peephole2
  [(set (match_operand:SI 0 "nonimmediate_operand")
        (match_operand:SI 1 "move_operand"))
   (set (match_operand:SI 2 "nonimmediate_operand")
        (match_operand:SI 3 "move_operand"))
   (set (match_operand:SI 4 "nonimmediate_operand")
        (match_operand:SI 5 "move_operand"))
   (set (match_operand:SI 6 "nonimmediate_operand")
        (match_operand:SI 7 "move_operand"))]
  ""
    [(set (match_dup 0)
        (plus:SI (match_dup 1)
            (plus:SI (match_dup 2)
                (plus:SI (match_dup 3)
                    (plus:SI (match_dup 4)
                        (plus:SI (match_dup 5)
                            (plus:SI (match_dup 6) (match_dup 7))
                        )
                    )
                )
            )
        )
    )]
)
```

The above peephole patterns matches 4 loads that occur together and outputs a insn that is a nested assign of 4 registers.

Defining the Vle Instruction

Add a matching rule in the machine description riscv.md file so that the compiler can replace the new insn.

```
(define_insn "*vle"
  [(set (match_operand:SI 0 "" "")
        (plus:SI (match_operand:SI 1 "" "")
                 (plus:SI (match_operand:SI 2 "" "")
                          (plus:SI (match_operand:SI 3 "" "")
                                   (plus:SI (match_operand:SI 4 "" "")
                                            (plus:SI (match_operand:SI 5 "" "")
                                                     (plus:SI (match_operand:SI 6 "" "")
                                                              (match_operand:SI 7 "" ""))))))))]
  ""
  { return riscv_output_vector_move (operands[0], operands[1]); }
)
```

Then define the output_vector_move in the riscv.c file.

```
const char *
riscv_output_vector_move (rtx dest, rtx src)
{
    enum rtx_code dest_code, src_code;

    dest_code = GET_CODE (dest);
    src_code = GET_CODE (src);

    if (dest_code == REG && src_code == MEM)
        return "vle \t%0,%1";
    else if (src_code == REG && dest_code == MEM)
        return "vse \t%z1,%0";

    gcc_unreachable ();
}
```

We also need to add a check for unaligned loads in riscv.c.

```
const int check_if_same (rtx op1, rtx op2, rtx op3, rtx op4)
{
    if(REGNO(op1) == REGNO (op2) && REGNO (op1) == REGNO (op3) && REGNO (op1) == REGNO (op4))
        return true;

    unsigned int base1 = REGNO (XEXP (op1, 0)); // Get the base register of op1
    unsigned int base2 = REGNO (XEXP (op2, 0)); // Get the base register of op2
    unsigned int base3 = REGNO (XEXP (op3, 0)); // Get the base register of op3
    unsigned int base4 = REGNO (XEXP (op4, 0)); // Get the base register of op4

    if (base1 == base2 && base1 == base3 && base1 == base4) {
        return true;
    }

    return false; // Different base registers
}
```

5.3.2 Custom Pass

While the peephole optimization takes care of any stray loads and stores, stray loads and stores are loads and stores that are not a part of a loop. We need to perform a custom pass vectorization The pass should identify 4 sets of load/store operations and emit corresponding vle and vse while also ensuring proper register allocation. We will be creating a pass name hoist_loads

Define a pass_data struct

```
 const pass_data pass_data_hoist_loads =
       {
              RTL_PASS, /* type */
              "hoist_loads", /* name */
              OPTGROUP_NONE, /* optinfo_flags */
              TV_HOIST_LOADS, /* tv_id */
```

```
            0, /* properties_required */
            0, /* properties_provided */
            0, /* properties_destroyed */
            0, /* todo_flags_start */
            TODO_df_finish, /* todo_flags_finish */
      };
```

Define a pass class, extending rtl_opt_pass class

```
    class pass_hoist_loads : public rtl_opt_pass
   {
   public:
       pass_hoist_loads (gcc::context *ctxt)
              : rtl_opt_pass (pass_data_hoist_loads, ctxt){}

       /* opt_pass methods: */
       /* The epiphany backend creates a second instance of this pass, so we need
          a clone method. */
       opt_pass * clone () { return new pass_hoist_loads (m_ctxt); }
       virtual bool gate (function *) { return true; }
       virtual unsigned int execute (function *)
       {
           return hoist_loads ();
       }

   };
```

Currently, we are enabling the function without any gating, if required some gating logic can be added to conditionally run this pass. However as this is a "free optimization" we ses no reason to omit this vectorization.

Define the function to handle the core logic of the pass

```
static void hoist_loads()
{
  basic_block bb;
  rtx_insn *insn;
  rtx_insn *ld[40];
  int ld_count = 0;

  rtx register_blocks[][4]= {
      { gen_rtx_REG (SImode, 10), gen_rtx_REG (SImode, 11), gen_rtx_REG (SImode, 12), gen_rtx_REG
          (SImode, 13)},
      { gen_rtx_REG (SImode, 14), gen_rtx_REG (SImode, 15), gen_rtx_REG (SImode, 16), gen_rtx_REG
          (SImode, 17)}
  };

  FOR_EACH_BB_FN (bb, cfun)
  {
    FOR_BB_INSNS (bb, insn)
    {
      if (load_insn_p (insn))
        ld[ld_count++] = insn;
    }
```

```
  qsort(ld, ld_count, sizeof(rtx_insn *), compare_mem_addresses);

  for(int i=0; i<ld_count; i++)
    dprintf( "ld[%d] = %d\n", i, INSN_UID(ld[i]));

  bool reg_block_taken[2] = {false, false};

  for (int i=0; i<2; i++)
    for(int j=0; j <4; j++)
      for(int k=0; k < ld_count; k++)
        if (REGNO(register_blocks[i][j]) == get_reg_no(SET_SRC(PATTERN(ld[k])))) {
          reg_block_taken[i] = true;
          break;
        }

  for (int j = 0; j < 2; j++) // Assuming 2 register blocks {a0, a1, a2, a3} and {a4, a5, a6, a7}
  {
    if (!reg_block_taken[j] && is_register_block_available(register_blocks[j], bb))
    {
      rtx reg_operands[] = {NULL, NULL, NULL, NULL};
      rtx src = NULL;

      for (int i = 0; i < ld_count - 3; i++)
      {
        rtx mem0 = SET_SRC(PATTERN(ld[i]));
        rtx mem1 = SET_SRC(PATTERN(ld[i + 1]));
        rtx mem2 = SET_SRC(PATTERN(ld[i + 2]));
        rtx mem3 = SET_SRC(PATTERN(ld[i + 3]));

        int diff0 = INTVAL(XEXP(XEXP(mem1, 0), 1)) - INTVAL(XEXP(XEXP(mem0, 0), 1));
        int diff1 = INTVAL(XEXP(XEXP(mem2, 0), 1)) - INTVAL(XEXP(XEXP(mem1, 0), 1));
        int diff2 = INTVAL(XEXP(XEXP(mem3, 0), 1)) - INTVAL(XEXP(XEXP(mem2, 0), 1));

        if (diff0 == 4 && diff1 == 4 && diff2 == 4) // Assuming 4 bytes apart for 32-bit loads
        {
          dprintf( "%d-%d) Found 4 loads from consecutive memory locations\n",j, i);
          // Found 4 loads from consecutive memory locations
          for (int k = 0; k < 4; k++) {
            reg_operands[k] = SET_DEST(PATTERN(ld[k + i]));
            del_insn(ld[k + i]);
            ld[k] = ld[k + 4];
          }

          ld_count -= 4;

          // Set src to the first memory location of the load with the lowest address
          src = mem0;

          // Call the functions
          replace_with_new_registers(register_blocks[j], reg_operands, bb);
          emit_vle_with_registers(register_blocks[j], src, bb);

          // Break out of the loop
          break;
        }
      }
    }
  }
}
```

```
}
```

Breaking down the function:

```
  rtx register_blocks[][4]= {
    { gen_rtx_REG (SImode, 0), gen_rtx_REG (SImode, 1), gen_rtx_REG (SImode, 2), gen_rtx_REG
        (SImode, 3)},
    { gen_rtx_REG (SImode, 4), gen_rtx_REG (SImode, 5), gen_rtx_REG (SImode, 6), gen_rtx_REG
        (SImode, 7)}
};
```

This declares the register blocks that can be used for vector operations. We are splitting the registers into blocks a0 to a3 and a4 to a7.

```
  FOR_BB_INSNS (bb, insn)
  {
    if (load_insn_p (insn))
      ld[ld_count++] = insn;
  }

    qsort(ld, ld_count, sizeof(rtx_insn *), compare_mem_addresses);

 bool reg_block_taken[2] = {false, false};

  for (int i=0; i<2; i++)
    for(int j=0; j <4; j++)
      for(int k=0; k < ld_count; k++)
        if (REGNO(register_blocks[i][j]) == get_reg_no(SET_SRC(PATTERN(ld[k])))) {
          reg_block_taken[i] = true;
          break;
        }
```

Here we are iterating through all the basic blocks and through each instruction inside a basic block. We are storing all the load instructions for later use. The sorting logic is described below. Once we have collected all the load instructions we mark register blocks that contain registers used as memory base registers for the load as unavailable for vectorization.

```
  for (int j = 0; j < 2; j++) // Assuming 2 register blocks {a0, a1, a2, a3} and {a4, a5, a6, a7}
  {
    if (!reg_block_taken[j] && is_register_block_available(register_blocks[j], bb))
    {
      rtx reg_operands[] = {NULL, NULL, NULL, NULL};
      rtx src = NULL;

      for (int i = 0; i < ld_count - 3; i++)
      {
        rtx mem0 = SET_SRC(PATTERN(ld[i]));
        rtx mem1 = SET_SRC(PATTERN(ld[i + 1]));
        rtx mem2 = SET_SRC(PATTERN(ld[i + 2]));
        rtx mem3 = SET_SRC(PATTERN(ld[i + 3]));

        int diff0 = INTVAL(XEXP(XEXP(mem1, 0), 1)) - INTVAL(XEXP(XEXP(mem0, 0), 1));
        int diff1 = INTVAL(XEXP(XEXP(mem2, 0), 1)) - INTVAL(XEXP(XEXP(mem1, 0), 1));
```

```
        int diff2 = INTVAL(XEXP(XEXP(mem3, 0), 1)) - INTVAL(XEXP(XEXP(mem2, 0), 1));

        if (diff0 == 4 && diff1 == 4 && diff2 == 4) // Assuming 4 bytes apart for 32-bit loads
        {
          dprintf( "%d-%d) Found 4 loads from consecutive memory locations\n",j, i);
          // Found 4 loads from consecutive memory locations
          for (int k = 0; k < 4; k++) {
            reg_operands[k] = SET_DEST(PATTERN(ld[k + i]));
            del_insn(ld[k + i]);
            ld[k] = ld[k + 4];
          }

          ld_count -= 4;

          // Set src to the first memory location of the load with the lowest address
          src = mem0;

          // Call the functions
          replace_with_new_registers(register_blocks[j], reg_operands, bb);
          emit_vle_with_registers(register_blocks[j], src, bb);

          // Break out of the loop
          break;
        }
      }
    }
  }
}
```

This is the core logic of the pass. Here we look for sets of 4 loads from consecutive memory locations and replace them with vle instruction.

- **is_register_block_available** function checks if the given register is alive outside the current basic block. We consider registers live only within the current basic block as available as reallocating registers within a basic block can be done without introducing much complexity.
- **replace_with_new_registers** function replaces register operands from all the instructions in the current basic block according to how registers were reallocated for vle instruction.
- **emit_vle_with_registers** function creates and emits vle instruction to the current basic block's RTL chain.

Function to check if a register block is alive outside a given basic block

```
   bool
is_live_outside(basic_block bb, unsigned regno)
{
  basic_block bb_iter;
  bitmap live_in, live_out;

  // Iterate through all basic blocks in the current function
```

```
  FOR_EACH_BB_FN(bb_iter, cfun)
  {
    // Skip the given basic block
    if (bb == bb_iter)
      continue;

    // Get the live-in and live-out sets for the current basic block
    live_in = DF_LR_IN(bb_iter);
    live_out = DF_LR_OUT(bb_iter);

    // Check if the register is live in or live out of the current basic block
    if ((live_in && bitmap_bit_p(live_in, regno)) || (live_out && bitmap_bit_p(live_out, regno)))
      return true; // The register is live outside the given basic block
  }

  return false; // The register is not live outside the given basic block
}

bool
is_register_block_available(rtx reg_block[], basic_block bb)
{
  // Check if any register from the block is live outside the current block
  for (int i = 0; i < 4; i++)
    if (is_live_outside(bb, REGNO(reg_block[i])))
      return false;

  return true;
}
```

Function to emit vle instruction to top of given basic block

```
void emit_vle_with_registers(rtx reg_block[], rtx mem_operand, basic_block bb)
{
  rtx_insn *vle_insn = as_a <rtx_insn *> (
      gen_vle (reg_block[0], mem_operand, reg_block[1], reg_block[2], reg_block[3]));
  emit_insn_after(vle_insn, BB_HEAD(bb));
}
```

Function to replace the occurrence of a register operand in a given instruction with a new register operand

```
   rtx replace_register_recursive(rtx x, rtx old_reg, rtx new_reg)
{
  int i, j;
  const char *fmt;

  // Base case: if x is NULL, return
  if (!x) return x;

  // If x is the old register, return the new register
```

```
  if (x == old_reg) return new_reg;

  // Otherwise, recursively process each sub-expression of x
  fmt = GET_RTX_FORMAT(GET_CODE(x));
  for (i = GET_RTX_LENGTH(GET_CODE(x)) - 1; i >= 0; i--)
  {
    if (fmt[i] == 'e')
    {
      XEXP(x, i) = replace_register_recursive(XEXP(x, i), old_reg, new_reg);
    }
    else if (fmt[i] == 'E')
    {
      for (j = XVECLEN(x, i) - 1; j >= 0; j--)
      {
        XVECEXP(x, i, j) = replace_register_recursive(XVECEXP(x, i, j), old_reg, new_reg);
      }
    }
  }

  return x;
}
```

Function to replace all corresponding pair of old registers with new registers in a basic block

```
void replace_with_new_registers(rtx reg_block[], rtx old_regs[], basic_block bb)
{
  rtx_insn *insn;

  // Iterate through all the instructions in the basic block
  FOR_BB_INSNS(bb, insn)
  {
    if(!insn || !INSN_P (insn))
      continue;

    // Check if the instruction uses any of the old registers
    for (int i = 0; i < 4; i++)
    {
      rtx new_pattern = replace_register_recursive(PATTERN(insn), old_regs[i], reg_block[i]);
      if (new_pattern != PATTERN(insn))
      {
        emit_insn_after(new_pattern, insn);
        del_insn(insn);
      }
    }
  }
}
```

Function to compare two load instructions first based on register number and then based on offset.

```
int compare_mem_addresses(const void *a, const void *b)
{
  rtx mem_a = SET_SRC(PATTERN(*(rtx_insn **)a));
  rtx mem_b = SET_SRC(PATTERN(*(rtx_insn **)b));

  int reg_num_a = get_reg_no(mem_a);
  int reg_num_b = get_reg_no(mem_b);

  // Compare by register number first
  if (reg_num_a != reg_num_b)
    return reg_num_a - reg_num_b;

  return get_offset(mem_a) - get_offset(mem_b);
}
```

With this the compiler is able to vectorize loads from 4 scalar loads to 1 vector load, similar modification is to be done to to add store support.

Chapter 6

Testing

6.1 Memcpy

We start testing by testing the program on a basic memcpy function that goes

```
void memcpy(unsigned long long *a,unsigned long long *b,){


for(short i=0;i<32;i+=2){
    b[i] = a[i];
    b[i+1] =a[i+1];
    }
}
```

Compiling without the vectorization optimizations we get the following.

```
    memcpy:
li a3,0
li t1,256
.L2:
add a4,a1,a3
add a2,a0,a3
lw a5,0(a4)
lw a6,4(a4)
lw a7,8(a4)
lw a8,12(a4)
sw a5,0(a2)
sw a6,4(a2)
sw a7,8(a2)
sw a8,12(a2)
addi a3,a3,8
bne a3,t1,.L2
```

Compiling with vectorization optimization gets us the following.

```
memcpy:
li a3,0
li t1,256
.L2:
add a4,a1,a3
add a2,a0,a3
```

```
vle a8,0(a4)
vst a8,0(a2)
addi a3,a3,8
bne a3,t1,.L2
```

From this we see that we see major improvement in the load and stores if they are well balanced. In theory ,we get a 4x reduction in loads and stores, decreasing our execution time from 448 cycles to 256 cycles given a perfectly cached memory and a load time of 1 cycle. However, with a more realworld example of 37 cycles per load [19] , we go from a cycle time of 9664 cycles to 2560 cycles which is close to a **3.7x** performance increase from not a large die size increase of about 26%.

6.2 addArray

We start testing by testing the program on a basic addArray function that goes

```
void addarray(unsigned long long *a,unsigned long long *b,unsigned long long *b){


for(short i=0;i<32;i+=2){
    c[i] = b[i] + a[i];
    c[i+1] =b[i+1]+ a[i+1];
    }
}
```

Compiling without the vectorization optimizations we get the following

```
addarray:
li a4,0
li t1,256
.L2:
add a6,a0,a4
add a7,a1,a4
lw a5,0(a6)
lw a3,0(a7)
lw a6,4(a6)
lw a7,4(a7)
add a3,a5,a3
add a6,a6,a7
add a5,a5,a6
add a7,a2,a4
sw a3,0(a7)
sw a5,4(a7)
addi a4,a4,8
bne a4,t1,.L2
```

Compiling with vectorization optimization gets us the following

```
addarray:
li a4,0
li t1,256
.L2:
add a6,a0,a4
add a7,a1,a4
vle a8,0(a6)
vle a12,0(a7)
```

```
add a8,a12,a8
add a9,a13,a9
add a10,a14,a10
add a11,a15,a11
vse a3,0(a5)
addi a4,a4,16
bne a4,t1,.L2
```

From this we see that we see major improvement in the load and stores however we still have the same amount of adds thus our performance gains will not be as good as memcpy. In theory ,we get a 4x reduction in loads and stores, decreasing our execution time from 576 cycles to 320 cycles given a perfectly cached memory and a load time of 1 cycle. However, with a more realworld example of 37 cycles per load [19] , we go from a cycle time of 14784 cycles to 4128 cycles which is close to a **3.5x** performance increase from not a large die size increase of 26%.

6.3 HammerBlade Modifications

Currently the HammerBlade chip is undergoing modifications to add support for VLE and VST instructions. Once they are completed the full suite of hammerbench benchmark suite can be tested and verified.

Chapter 7

Conclusion and Future work

In this thesis, we embarked on a comprehensive exploration of the integration between cutting-edge hardware architectures, exemplified by HammerBlade, and the advanced capabilities of vectorizing compilers within the realm of high-performance computing. This journey has not only highlighted the transformative potential of such integration for computational efficiency and performance but has also laid down a foundational blueprint for future innovations in processor architecture and compiler design. The focus on HammerBlade's architectural innovations, coupled with the in-depth analysis of vectorizing compilers, underscores a pivotal advancement towards optimizing computational tasks across various domains, including but not limited to big data analytics, machine learning, and scientific simulations.

The synergy between HammerBlade's architectural innovations and vectorizing compilers lies in their mutual goal: to maximize the computational power available to applications while minimizing resource consumption. For developers, this integration means the ability to write code that not only runs faster but also more efficiently, by automatically adapting to the underlying hardware's parallelism capabilities. This is particularly relevant in fields that demand substantial computational resources, such as machine learning, scientific simulation, and real-time data analysis.

However, realizing the full potential of this synergy requires overcoming challenges such as optimizing compiler algorithms to understand and exploit HammerBlade's unique architecture and ensuring that the programming models and tools available to developers are both powerful and accessible. As compilers become more adept at automatic vectorization for architectures like HammerBlade, the barrier to entry for writing highly efficient parallel code lowers, democratizing access to high-performance computing capabilities. There are multiple enhancements to this system that can be made to improve performance.

7.1 Summary of Contributions

My contributions through this thesis encompass several key areas:

- **Architectural Exploration**: Providing an in-depth analysis of HammerBlade's architecture created by the BSG Team and documenting the minimum modifications required for vectorization and emphasizing its potential to redefine performance benchmarks in open-source chip technology. This exploration sheds light on the architectural nuances that facilitate high-performance computing, offering valuable insights for future architectural designs.
- **Compiler Optimization**: Delving into the intricacies of vectorizing compilers, this thesis outlines strategies for harnessing compiler technologies to exploit the parallel processing capabilities of modern hardware effectively. The examination of automatic vectorization, loop unrolling, and other optimization techniques contributes to a deeper understanding of compiler efficiency in enhancing application performance. It also creates the first vectorizing compiler for the hammerblade chip.
- **Compiler Modification Guide**: Beyond theoretical contributions, this thesis serves as a practical guide for developers and researchers aiming to leverage HammerBlade's architecture and vectorizing compilers in their projects. Through detailed discussions on challenges, solutions, and potential enhancements, the document provides a roadmap for optimizing computing tasks and navigating the complexities of high-performance computing.

7.2 Conclusion and Future Implications

In conclusion, the integration of HammerBlade's innovative hardware with the evolving sophistication of vectorizing compilers represents a significant milestone in high-performance computing. This thesis contributes to the body of knowledge by offering a comprehensive analysis of this integration, highlighting its potential to significantly enhance computational efficiency and performance. As we stand on the brink of a new era in computing technology, the insights and contributions presented in this document for future projects, guiding researchers, developers, and enthusiasts to push the boundaries of computing capabilities.

Future enhancements to this system, aimed at bolstering performance, could include:

- The implementation of true SIMD capabilities and dedicated vector registers.
- Automated loop unrolling for multi-nested loops to enhance execution efficiency.
- Introduction of loop masking techniques for conditional statements within loops.
- Expansion to wider data buses, such as 256-bit or 512-bit configurations, akin to AVX-512, to increase data throughput.

These potential modifications represent just the tip of the iceberg in the vast sea of possibilities for vectorization and architectural optimization.

Chapter 8

Appendix: Writing Vectorizable Code

8.1 Vectorization Guidelines

To enhance the ability of compilers to vectorize code efficiently, thereby improving performance through parallel execution on modern hardware architectures, follow these guidelines

- **Simple "For" Loops**: Opt for countable, single entry, and single exit "for" loops. Complex termination conditions should be avoided; both the loop's lower and upper bounds should remain constant throughout its execution.
- **Write Linear Code**: Aim for straight-line code without branches. This means minimizing the use of switch statements, goto, return statements within loops, and most function calls.
- **Minimize Dependencies Between Iterations**: Vectorization thrives on independence between loop iterations. Specifically, read-after-write dependencies (where a loop iteration reads a value that was written in a previous iteration) should be avoided as they prevent parallel execution of the iterations.
- **Prefer Arrays Over Pointers**: When possible, use array notation rather than pointers. C programs offer considerable flexibility with pointers, leading to potential aliasing and unexpected dependencies that can confuse the compiler about the safety of vectorizing code with pointers.
- **Use of Loop Index in Array Subscripts**: Wherever possible, directly use the loop index for array subscripts. This is more straightforward than using a separate counter variable to track array positions, which can complicate the compiler's analysis for vectorization.
- **Unit Stride in Inner Loops**: Strive for memory access patterns that advance one element at a time (unit stride) in the innermost loops. This pattern is more vectorization-friendly as it aligns with how vector instructions load and store data.
- **Minimize Indirect Addressing**: Try to reduce the use of indirect addressing (accessing elements through pointer arrays or lookup tables), as it can introduce irregular memory access patterns that are difficult to vectorize.

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, and the DARPA/SRC JUMP ADA Center. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([8, 20, 103, 100, 62, 60, 105, 6, 97, 71, 28, 74, 4, 72, 27, 29, 50, 96]), ASIC Clouds ([93, 101, 90, 79, 52, 51, 55, 82]), open source hardware ([92, 81, 18]) RISC-V ([64, 70, 69, 13, 104, 2, 95, 46, 68, 57, 11]), Network-on-Chips ([47, 65, 106, 86, 53]), security ([10, 37, 36, 3]), benchmark suites ([94, 54, 5]), dark silicon ([78, 77, 78, 83, 7, 30, 27, 96]), multicore ([64, 34, 33, 35, 80, 86, 85, 89, 76, 53, 84, 87, 88, 98]), compiler tools ([41, 21, 42, 23, 44, 43, 22, 102, 1]) and FPGAs ([107, 38, 5]).

Bibliography

[1] A. Agarwal et al. "The RAW compiler project". In: *Proceedings of the Second SUIF Compiler Workshop*. 1997, pp. 21–23.

[2] Tutu Ajayi et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric". In: *HOTCHIPS*. Aug. 2017.

[3] Alric Althoff et al. "Hiding Intermittant Information Leakage with Architectural Support for Blinking". In: *International Symposium on Computer Architecture (ISCA)*. 2018.

[4] Manish Arora et al. "Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems". In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2011.

[5] Jonathan Babb et al. "The Raw Benchmark Suite: Computation Structures for General Purpose Computing". In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 1997.

[6] B. Beresini, S. Ricketts, and M.B. Taylor. "Unifying manycore and FPGA processing with the RUSH architecture". In: *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*. 2011, pp. 22–28.

[7] Vikram Bhatt et al. " Sichrome: Mobile web browsing in Hardware to save Energy ". In: *Dark Silicon Workshop, ISCA*. 2012.

[8] Ajay Brahmakshatriya et al. "Taming the Zoo: A Unified Graph Compiler Framework for Novel Architectures". In: *ISCA*. 2021.

[9] Preston Briggs, Keith D. Cooper, and Linda Torczon. "Improvements to graph coloring register allocation". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 428–455. ISSN: 0164-0925. DOI: 10.1145/177492.177575. URL: https://doi.org/10.1145/177492.177575.

[10] Sadullah Canakci et al. "DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing". In: *DAC*. 2021.

[11] Yuan-Mao Chueh. "A Complete Open Source Network Stack For BlackParrot". MA thesis. University of Washington, 2022.

[12] Frederica Darema. "The SPMD Model: Past, Present and Future". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Yiannis Cotronis and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–1. ISBN: 978-3-540-45417-5.

[13] Scott Davidson et al. "The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric". In: *Micro, IEEE* (Mar. 2018).

[14] Scott Davidson et al. "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips". In: *IEEE Micro* 38.2 (2018), pp. 30–41. DOI: 10.1109/MM.2018.022071133.

[15] Ramandeep Singh Dehal et al. "GPU Computing Revolution: CUDA". In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2018, pp. 197–201. DOI: 10.1109/ICACCCN.2018.8748495.

[16] Neil Dickson, Kamran Karimi, and Firas Hamze. "Importance of Explicit Vectorization for CPU and GPU Software Performance". In: *Journal of Computational Physics* 230 (Mar. 2010), pp. 5383–5398. DOI: 10.1016/j.jcp.2011.03.041.

[17] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. "A Survey of Different Approaches for Overcoming the Processor - Memory Bottleneck". In: *International Journal of Information Technology and Computer Science* 9 (Apr. 2017), p. 151. DOI: 10.5121/ijcsit.2017.9214.

[18] Hadi Esmaeilzadeh and Michael Bedford Taylor. "Open Source Hardware: Stone Soups and Not Stone Satues, Please". In: *SIGARCH Computer Architecture Today*. Dec. 2017.

[19] Si Five. *SiFive E7 Core*. 2017. URL: https://sifive-china.oss-cn-zhangjiakou.aliyuncs.com/Standard%20Core%20IP/e76mc_core_complex_manual_21G2.pdf.

[20] Emily Furst. "Code Generation and Optimization of Graph Programs on a Manycore Architecture". PhD thesis. University of Washington, 2021.

[21] S. Garcia et al. "The Kremlin Oracle for Sequential Code Parallelization". In: *Micro, IEEE* 32.4 (July 2012), pp. 42–53.

[22] Saturnino Garcia et al. "Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning". In: *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*. 2010.

[23] Saturnino Garcia et al. "Kremlin: Rethinking and Rebooting gprof for the Multicore Age". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2011.

[24] Lal George and Andrew W. Appel. "Iterated register coalescing". In: *ACM Trans. Program. Lang. Syst.* 18.3 (May 1996), pp. 300–324. ISSN: 0164-0925. DOI: 10.1145/229542.229546. URL: https://doi.org/10.1145/229542.229546.

[25] P. Gepner and M.F. Kowalik. "Multi-Core Processors: New Way to Achieve High System Performance". In: *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*. 2006, pp. 9–13. DOI: 10.1109/PARELEC.2006.54.

[26] GNU Project. *GCC: The GNU Compiler Collection*. https://gcc.gnu.org/. 1983.

[27] Nathan Goulding et al. "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon". In: *HOTCHIPS*. 2010.

[28] N. Goulding-Hotta et al. "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future". In: *Micro, IEEE* (Mar. 2011), pp. 86–95.

[29] Nathan Goulding-Hotta. "Specialization as a Candle in the Dark Silicon Regime". PhD thesis. University of California, San Diego, 2020.

[30] Nathan Goulding-Hotta et al. "GreenDroid: An Architecture for the Dark Silicon Age". In: *Asia and South Pacific Design Automation Conference (ASPDAC)*. 2012.

[31] Bespoke Sillicon Group. *Hammerblade Benchmark Suite*. 2017. URL: https://github.com/bespoke-silicon-group/hb_hammerbench.

[32] Bespoke Sillicon Group. *HammerBlade Manycore Overview*. 2020. URL: https://docs.google.com/document/d/1i62N72pfx2Cd_xKT3hiTuSilQnuC0ZOaSQMG8UPkto/edit#heading=h.uoulvwin7369.

[33] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache". In: *International Conference on Computer Design (ICCD)*. 2013.

[34] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "QualityTime: A Simple Online Technique for Quantifying Multicore Execution Efficiency". In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014.

[35] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking". In: *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*. 2013.

[36] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. "A Runtime Approach to Security and Privacy". In: *European Security and Privacy*. 2016.

[37] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. " BlackBox: Lightweight Security Monitoring for COTS Binaries". In: *Code Generation and Optimization*. 2016.

[38] Hu et al. "FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach ". In: *ICCD*. 2007.

[39] RISC-V International. *RISC-V P Extension*. 2020. URL: https://github.com/riscv/riscv-p-spec.

[40] RISC-V International. *RISC-V P Extension GCC compiler*. 2020. URL: https://github.com/linsinan1995/riscv-gcc/tree/riscv-gcc-experiment-p-ext.

[41] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. "Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis". In: *Conference on Code Generation and Optimization (CGO)*. 2013.

[42] Donghwan Jeon et al. "Kismet: Parallel Speedup Estimates for Serial Programs". In: *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*. 2011.

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

[44] Donghwan Jeon et al. "Parkour: Parallel Speedup Estimates from Serial Code". In: *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*. 2011.

[45] Hongshin Jun et al. "HBM (High Bandwidth Memory) DRAM Technology and Architecture". In: *2017 IEEE International Memory Workshop (IMW)*. 2017, pp. 1–4. DOI: 10.1109/IMW.2017.7939084.

[46] Dai Cheol Jung. "Caches for Complex Open Source System-on-Chip Designs". MA thesis. University of Washington, 2019.

[47] Dai Cheol Jung et al. "Ruche Networks: Wire-Maximal, No-Fuss NoCs". In: *NOCS*. 2020.

[48] Dai Cheol Jung et al. "Ruche Networks: Wire-Maximal, No-Fuss NoCs : Special Session Paper". In: *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. 2020, pp. 1–8. DOI: 10.1109/NOCS50636.2020.9241586.

[49] Jehandad Khan et al. *MIOpen: An Open Source Library For Deep Learning Primitives*. 2019. arXiv: 1910.00078 [cs.LG].

[50] Moein Khazraee. "Reducing the development cost of customized cloud infrastructure". PhD thesis. University of California, San Diego, 2020.

[51] Moein Khazraee et al. "Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.

[52] Moein Khazraee et al. "Specializing a Planet's Computation: ASIC Clouds". In: *IEEE Micro* (May 2017).

[53] Jason Kim et al. "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. Aug. 2003.

[54] Sravanthi Kota Venkata et al. "SD-VBS: The San Diego Vision Benchmark Suite". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2009.

[55] Ikuo Magaki et al. "ASIC Clouds: Specializing the Datacenter". In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[56] W. Maly. "Cost of Silicon Viewed from VLSI Design Perspective". In: *31st Design Automation Conference*. 1994, pp. 135–142. DOI: 10.1145/196244.196311.

[57] Sripathi Muralitharan. "TinyParrot: An Integration-Optimized Linux-Capable Host Multicore". MA thesis. University of Washington, 2021.

[58] Bhagwath Nitin et al. "DDR5 design challenges". In: *2018 IEEE 22nd Workshop on Signal and Power Integrity (SPI)*. 2018, pp. 1–4. DOI: 10.1109/SaPIW.2018.8401666.

[59] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[60] S. Pal et al. "A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm". In: *Symposium on VLSI Circuits*. 2019, pp. C150–C151.

[61] Panos M. Pardalos, Thelma Mavridou, and Jue Xue. "The Graph Coloring Problem: A Bibliographic Survey". In: *Handbook of Combinatorial Optimization: Volume1–3*. Ed. by Ding-Zhu Du and Panos M. Pardalos. Boston, MA: Springer US, 1998, pp. 1077–1141. ISBN: 978-1-4613-0303-9. DOI: 10.1007/978-1-4613-0303-9_16. URL: https://doi.org/10.1007/978-1-4613-0303-9_16.

[62] D. Park et al. "A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator". In: *IEEE Journal of Solid-State Circuits* (Apr. 2020), pp. 933–944.

[63] David Patterson and Andrew Waterman. *The RISC-V Reader: an Open Architecture Atlas*. San Francisco, CA, USA: Strawberry Canyon, 2017, pp. xiv + 180. ISBN: 0-9992491-1-8.

[64] D. Petrisko et al. "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs". In: *IEEE Micro* (July 2020), pp. 93–102.

[65] Daniel Petrisko et al. "NoC Symbiosis". In: *NOCS*. 2020.

[66] Uma R, Sarojadevi H., and Sanju V. "Network-On-Chip (NoC) - Routing Techniques: A Study and Analysis". In: *2019 Global Conference for Advancement in Technology (GCAT)*. 2019, pp. 1–6. DOI: 10.1109/GCAT47503.2019.8978403.

[67] Altair Radioss. *Single Program Multiple Data*. 2020. URL: https://2021.help.altair.com/2021/hwsolvers/rad/topics/solvers/rad/theory_radioss_parallelization_spmd_r.htm.

[68] Shashank Vijaya Ranga. "ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor". MA thesis. University of Washington, 2021.

[69] A. Rovinski et al. "A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS". In: *2019 Symposium on VLSI Circuits*. 2019, pp. C30–C31.

[70] A. Rovinski et al. "Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL". In: *IEEE Solid-State Circuits Letters* 2.12 (2019), pp. 289–292.

[71] Jack Sampson et al. "An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors". In: *Conference on Field Programmable Logic and Applications (FPL)*. 2011.

[72] Jack Sampson et al. "Efficient Complex Operators for Irregular Codes". In: *High Performance Computing Architecture (HPCA)*. 2011.

[73] Martin Svedin et al. *Benchmarking the Nvidia GPU Lineage: From Early K80 to Modern A100 with Asynchronous Memory Transfers*. 2021. eprint: arXiv:2106.04979.

[74] Steven Swanson and Michael Taylor. "GreenDroid: Exploring the next evolution for smartphone application processors". In: *IEEE Communications Magazine*. Mar. 2011.

[75] Jarmo Takala and Tuomas Jarvinen. "Stride Permutation Access In Interleaved Memory Systems". In: (Nov. 2003). DOI: 10.1201/9780203913185.ch4.

[76] MB Taylor et al. "The Raw processor-a scalable 32-bit fabric for embedded and general purpose computing". In: *Proceedings of Hot Chips XIII*. 2001.

[77] Michael Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *Micro, IEEE* (Sept. 2013).

[78] Michael Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *Design Automation and Test in Europe*. Apr. 2014.

[79] Michael Taylor. "The Evolution of Bitcoin Hardware". In: *Computer, IEEE* (Sept. 2017).

[80] Michael Taylor. "Tiled Microprocessors". PhD thesis. Massachusetts Institute of Technology, 2007.

[81] Michael B. Taylor. "BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design". In: *Design Automation Conference*. June 2018.

[82] Michael B. Taylor. "Bitcoin and the Age of Bespoke Silicon". In: *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 2013.

[83] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *Design Automation Conference (DAC)*. 2012.

[84] Michael B. Taylor et al. "A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network". In: *IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2003.

[85] Michael B. Taylor et al. "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams". In: *International Symposium on Computer Architecture (ISCA)*. June 2004.

[86] Michael B. Taylor et al. "Scalar Operand Networks". In: *IEEE Transactions on Parallel and Distributed Systems*. Feb. 2005.

[87] Michael B. Taylor et al. "Scalar Operand Networks". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. Feb. 2005.

[88] Michael B. Taylor et al. "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures". In: *International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2003.

[89] Michael B. Taylor et al. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs". In: *IEEE Micro*. Mar. 2002.

[90] Michael Bedford Taylor. "Geocomputers and the Commercial Borg". In: *SIGARCH Computer Architecture Today*. Dec. 2017.

[91] Michael Bedford Taylor. "Tiled Microprocessors". MA thesis. MIT, 2007. URL: `https://www.bsg.ai/papers/Tiled_Microprocessors_MBT_Thesis.pdf`.

[92] Michael Bedford Taylor. "Your agile open source HW stinks (because it is not a system)". In: *ICCAD*. 2020.

[93] Michael Bedford Taylor et al. "ASIC Clouds: Specializing the Datacenter for Planet-Scale Applications". In: *CACM* (2020), pp. 103–109.

[94] Shelby Thomas et al. "CortexSuite: A Synthetic Brain Benchmark Suite". In: *International Symposium on Workload Characterization (IISWC)*. Oct. 2014.

[95] Luis Vega and Michael Bedford Taylor. " RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization ". In: *CARRV*. 2017.

[96] Ganesh Venkatesh et al. "Conservation cores: reducing the energy of mature computations". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010.

[97] Ganesh Venkatesh et al. "QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner". In: *International Symposium on Microarchitecture (MICRO)*. 2011.

[98] Elliot Waingold et al. "Baring it all to Software: Raw Machines". In: *IEEE Computer*. Sept. 1997.

[99] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`.

[100] Chenhao Xie et al. "Q-VR: System-Level Design for Future Mobile Collaborative Virtual Reality". In: *ASPLOS*. 2021.

[101] Shaolin Xie et al. "Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds". In: *ACM Sigops Operating System Review*. 2018.

[102] Karen Zee et al. "Runtime Checking for Program Verification". In: *RV*. 2007.

[103] Xingyao Zhang et al. "η-LSTM: Co-Designing Highly-Efficient Large LSTM Training via Exploiting Memory-Saving and Architectural Design Opportunities". In: *ISCA*. 2021.

[104] Ritchie Zhao et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric". In: *7th RISC-V Workshop*. 2017.

[105] Qiaoshi Zheng et al. "Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon". In: *Transactions on Embedded Computing Systems (TECS)* (Mar. 2014).

[106] Yi Zhu et al. “Advancing supercomputer performance through interconnection topology synthesis”. In: *International Conference on Computer-Aided Design (ICCAD)*. 2008, pp. 555–558.

[107] Yi Zhu et al. “Energy and Switch Area Optimizations for FPGA Global Routing Architectures”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. Jan. 2009.