Computer Architecture Project

A Study on the Effect of Vector Length on the Performance of Vectorized Code and Scalar Code

---

Prepared By

Ahmed Osama Ibrahim

Moamen Mohamed Ahmed

Yousef Taha Saad

Zeyad Hashem Mohamed

Ziad Ahmed Mohamed

Made under the supervision of

Professor Mai Mohamed

Abstract

This report investigates the performance implications of increasing vector length in both vector and scalar (normal) architectures, with a specific focus on execution time measured in clock cycles. While both architectures exhibit an increase in the number of clock cycles as vector length grows, our experiments reveal that the rate of increase is significantly lower for the vectorized implementation. In other words, as the workload scales, the vector architecture handles larger data sizes more efficiently, leading to a lower multiplication factor in clock cycles compared to the scalar counterpart. This efficiency becomes particularly pronounced at higher vector lengths, where the performance gap between the two architectures widens considerably. The vector architecture demonstrates a substantial performance advantage for large vector sizes, showcasing its ability to exploit data-level parallelism effectively. These results underline the scalability and suitability of vector architectures for high-throughput, data-parallel applications, especially as problem sizes grow.

I. Introduction

The rapid advancement of modern computing systems has brought about increasingly complex computational workloads that demand higher performance and efficiency. In response to these demands, processor architects have developed specialized designs aimed at accelerating execution through parallelism. Among these designs, vector architectures have emerged as a powerful alternative to traditional scalar architectures for many data-intensive tasks.

The primary objective of this project is to perform a detailed comparative analysis of vector and scalar architectures, with a specific focus on how vector length—the number of data elements that can be processed simultaneously—impacts performance. Scalar architectures operate on a single data element at a time, executing instructions sequentially. This traditional model is simple and well-suited for control-heavy or irregular workloads, but it often struggles to deliver high throughput for data-parallel operations commonly found in scientific computing, image processing, and large-scale simulations.

In contrast, vector architectures are designed to exploit data-level parallelism by executing a single instruction across multiple data elements in parallel. This model, referred to as SIMD (Single Instruction, Multiple Data), enables significant speedup for operations involving large arrays or vectors of data, provided that the workload exhibits sufficient regularity and parallelism. A key parameter in this architecture is the vector length, which determines the number of elements processed in one vector instruction. The performance gains achieved by a vector processor often depend heavily on this vector length, as well as how well the underlying hardware and memory subsystems can support such parallelism.

In this study, we investigate how varying the vector length affects overall execution time, throughput, and resource utilization when compared to scalar processing. We analyze both synthetic benchmarks and representative real-world workloads to understand how each architecture performs under different scenarios. By systematically adjusting the vector length, we aim to uncover performance trends and identify bottlenecks or diminishing returns that occur as the vector width increases.

Our analysis focuses mainly on the number of clock cycles elapsed during execution. These aspects are increasingly relevant in the context of modern computing. The insights gained from this

comparison are intended to inform decisions about processor design, compiler optimizations, and application development strategies.

Ultimately, by comparing scalar and vector architectures through the lens of vector length, we aim to provide a comprehensive understanding of the trade-offs involved in vectorization. This will help clarify where and when vector architectures provide significant advantages, and in what contexts scalar processing may still be preferable.

II. Methodology

2.1 Tools Used in the Study

To carry out the performance analysis of vector and scalar architectures, we utilized a set of tools specifically designed for the RISC-V ecosystem. These tools enabled us to write, compile, simulate, and run RISC-V programs in a controlled environment, allowing for accurate measurement of performance metrics such as clock cycles. The primary tools used in this study are the **RISC-V GNU Toolchain**, the **Spike RISC-V Simulator**, and the **Proxy Kernel (PK)**. Each plays a crucial role in the workflow, as described below.

2.1.1 RISC-V GNU Toolchain

The RISC-V GNU Toolchain is a collection of open-source tools based on the GNU Compiler Collection (GCC) that supports the RISC-V instruction set architecture (ISA). It includes the riscv64-unknown-elf-gcc compiler, assembler, linker, and other essential development utilities. In our study, this toolchain was used to write and compile C and assembly programs targeting both scalar and vector implementations. We compiled our source code into RISC-V ELF (Executable and Linkable Format) binaries that are compatible with RISC-V simulation tools. Importantly, this toolchain supports RISC-V vector extensions, allowing us to write vectorized code and leverage vector instructions for performance comparison.

2.1.2 Spike RISC-V Simulator

Spike is the official RISC-V ISA simulator and serves as a functional reference implementation of the RISC-V specification. It simulates the execution of RISC-V binaries, including those containing vector instructions, and provides a detailed, cycle-accurate view of program behavior. In our workflow, Spike was used to run the compiled programs and collect performance data, particularly the number of clock cycles required for each execution. This allowed us to compare how the scalar and vector versions of the same program behaved under varying vector lengths. Spike also integrates with additional tools to provide debug and profiling information, making it a critical component of our performance evaluation.

2.1.3 Proxy Kernel (PK)

The Proxy Kernel is a lightweight runtime environment designed to provide basic operating system functionality for RISC-V programs running under Spike. It handles tasks such as memory allocation, I/O, and system call translation, which are necessary for running user-level applications in the simulated environment. In our study, the Proxy Kernel was used to bootstrap and manage

the execution of our test programs on Spike. It provides the runtime support needed to execute C programs compiled with the RISC-V toolchain, ensuring that the test environment is consistent and behaves similarly to an actual RISC-V platform.

The integration of these tools follows a structured workflow:

- Code Development: Programs were written in C with optional use of vector intrinsics or inline assembly for vector operations.
- Compilation: The RISC-V GNU Toolchain compiled the programs into RISC-V ELF binaries, targeting the appropriate ISA extensions (scalar or vector).
- Execution: These binaries were run on the Spike simulator with the Proxy Kernel providing runtime support.
- Data Collection: Spike's performance counters were used to record execution metrics, including the number of clock cycles, allowing for quantitative comparison between scalar and vector implementations across varying vector lengths.

Together, these tools provided a robust and flexible platform for exploring architectural behavior, enabling us to isolate the effect of vector length on performance in a reproducible and accurate manner.

2.2 Test Plan

The core objective of our testing methodology was to evaluate and compare the performance of vectorized and non-vectorized (scalar) implementations of the same computational workload, specifically focusing on how execution time scales with increasing vector length. The primary metric used for comparison was the number of clock cycles required to complete execution in each case. Although precise, hardware-level timing measurements were not feasible within the constraints of our simulation environment, we were able to obtain estimated clock cycle counts using tools available in the RISC-V software stack.

2.2.1 Test Design

The test programs were designed to perform a simple, repetitive vector operation—such as vector addition or multiplication—on arrays of increasing lengths. Two versions of each test case were implemented:

- Scalar Implementation: Operated element-by-element using standard loops and scalar instructions.
- Vectorized Implementation: Used RISC-V vector intrinsics or assembly-level vector instructions to perform the same operation across multiple data elements in parallel, depending on the current vector length.

These implementations allowed us to observe how both scalar and vector architectures scale in performance as the problem size grows.

2.2.2 Compilation and Execution

Both implementations were compiled using the RISC-V GNU Toolchain, with appropriate flags to enable vector instruction support for the vectorized version. The resulting ELF binaries were then executed using the Spike RISC-V Simulator, with the Proxy Kernel (PK) providing a minimal runtime environment.

We used the –log-commits and –isa options in Spike where applicable, and more importantly, enabled performance logging to retrieve estimated cycle counts. These logs provided a rough approximation of the number of clock cycles required to complete each test case.

2.2.3 Measuring Clock Cycles

The most important part of the testing process was measuring the estimated number of clock cycles consumed during the execution of each version of the program. This measurement allowed us to compare the relative efficiency of scalar and vector approaches under identical conditions. Due to the simulated nature of the testing environment and lack of access to physical hardware counters, precise and cycle-accurate measurement was not achievable. Instead, we relied on the Spike simulator's internal instruction-level performance estimation, which provides a useful—albeit approximate—indicator of execution cost in terms of clock cycles.

Although these values are not exact reflections of real hardware performance, they are consistent and reliable enough to highlight relative trends and scaling behavior between the two architectures. This made it possible to identify how each architecture responds to increasing vector lengths and to determine the point at which vectorization begins to show significant advantages.

2.2.4 Vector Length Variation

To investigate the influence of vector length on performance, we systematically increased the size of the input vectors across a range of test runs. At each step, we recorded the estimated clock cycle count for both scalar and vectorized versions. This allowed us to analyze how execution time scales and to observe the efficiency of vectorized computation in handling larger workloads.

The following snippet of code is taken from the source code file and this function is the one used to measure the elapsed clock cycles, the chosen performance metric for this study.

```
uint64_t measure_time_clock(void (*func)(size_t, const float, const float*, float*),
size_t n, const float a, const float* x, float* y){
    clock_t start,end;
    start = clock();
    func(n, a, x, y);
    end = clock();
    elapsed_time = ((double)(end - start)) * CLOCKS_PER_SEC;
    return elapsed_time;
}
```

The code was given both SAXPY functions, they implement the vectorized code and the scalar code, and then returned the approximate clock cycles elapsed during the execution of both functions with the same input arguments. The result was displayed in the `main()` function later.

2.3 Coded Benchmark

Evaluating the performance of computer architectures—especially when comparing fundamentally different approaches such as scalar and vector processing—requires consistent, well-defined workloads that can produce meaningful and reproducible results. This is where benchmarks play a critical role. Benchmarks are standardized programs or computational patterns designed to simulate real-world workloads, allowing researchers and developers to measure and compare performance across architectures, compilers, and systems in a fair and systematic way.

Benchmarks are essential for several reasons:

- Consistency: They provide a repeatable and uniform basis for testing across different platforms and configurations.
- Representativeness: Good benchmarks mimic the behavior of real applications, offering insights into how architectures will perform in practical scenarios.

- Comparability: They enable side-by-side performance evaluations of different systems under identical computational tasks.
- Scalability Testing: Benchmarks help in observing how performance evolves with increasing input sizes or workloads.

In this study, we selected the SAXPY benchmark as our primary test workload to evaluate the performance of scalar and vector architectures. SAXPY stands for Single-Precision A·X Plus Y, and it is a well-known vector operation used frequently in linear algebra, scientific computing, and graphics applications.

The SAXPY operation is defined as:

$$Y = aX + Y$$

or the result of the operation could be stored somewhere other than the `Y` vector, but this is not really important for defining the benchmark. To explain the equation:

- `a` is the scalar input.
- `X` and `Y` are vectors.

This simple yet representative operation is ideal for evaluating vector architectures because:

- It involves both scalar and vector operations, making it suitable for contrasting scalar versus vector execution.
- It is memory-intensive and compute-light, stressing memory access patterns and instruction-level parallelism.
- It maps naturally to vector hardware using SIMD instructions, making it a classic example to showcase the potential benefits of vectorization.
- It scales well with vector length, allowing us to test how performance evolves with increasing data sizes.

By using the SAXPY benchmark, we were able to simulate a realistic, data-parallel workload that is sensitive to architectural differences in how data is fetched, processed, and stored. This made it an excellent candidate for our study of the effect of vector length on performance. The benchmark provided a meaningful and controlled context in which to measure the estimated number of clock cycles required by both scalar and vector implementations, thus clearly demonstrating the performance trends and advantages of vector processing as the workload size increased.

The following code snippet is the one used to represent the vectorized code that performs the benchmark operation SAXPY:

```
void saxpy_vec(size_t n, const float a, const float *x, float *y) {
    for (size_t vl; n > 0; n -= vl, x += vl, y += vl) {
        vl = __riscv_vsetvl_e32m8(n);
        vfloat32m8_t vx = __riscv_vle32_v_f32m8(x, vl);
        vfloat32m8_t vy = __riscv_vle32_v_f32m8(y, vl);
        __riscv_vse32_v_f32m8(y, __riscv_vfmacc_vf_f32m8(vy, a, vx, vl), vl);
    }
}
```

And the following code snippet is the one used to represent the scalar code that performs the benchmark operation SAXPY:

```
void saxpy_normal(size_t n, const float a, const float *x, float *y) {
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

We chose to implement and run the performance tests in C rather than writing directly in RISC-V assembly because C provides a higher-level, more readable, and maintainable programming model

while still offering the ability to control low-level behavior when needed—especially through the use of compiler intrinsics or inline assembly for vector operations. Writing in C significantly reduces development time and complexity, allows for easier modification and scaling of test cases, and ensures better portability across different platforms and toolchains. Additionally, modern compilers for RISC-V are capable of generating efficient machine code from C, especially when targeting specific ISA extensions such as vector instructions, making C a practical and effective choice for performance benchmarking while still preserving the ability to compare low-level architectural behavior.

III. Results

This section presents the results of our experimental analysis comparing the performance of scalar and vector implementations of the SAXPY benchmark across varying vector lengths. The primary metric used to evaluate performance was the estimated number of clock cycles required to complete the computation in each case. By systematically increasing the vector length and recording the corresponding clock cycle counts for both architectures, we observed clear trends in how each approach scales. The table below summarizes the collected data, highlighting the relative efficiency and growing performance advantage of the vector architecture as the workload size increases.

Vector Length	Cycles for Vecorized Code	Cycles for Scalar Code
100	$\tilde{1}000000$	$\tilde{2}000000$
200	$\tilde{1}000000$	$\tilde{5}000000$
300	$\tilde{1}000000$	$\tilde{7}000000$
400	$\tilde{2}000000$	$\tilde{1}0000000$
500	$\tilde{2}000000$	$\tilde{1}2000000$
600	$\tilde{3}000000$	$\tilde{1}5000000$
700	$\tilde{4}000000$	$\tilde{1}7000000$
800	$\tilde{4}000000$	$\tilde{2}0000000$
900	$\tilde{5}000000$	$\tilde{2}2000000$
1000	$\tilde{5}000000$	$\tilde{2}4000000$

These results provide a foundational comparison between the scalar and vector implementations across different vector lengths. While clear performance patterns begin to emerge, a more detailed analysis of these trends—including their implications and underlying causes—will be discussed in the following section.

IV. Analysis

In the context of vector architectures, vector length refers to the number of data elements that a single vector instruction can operate on simultaneously. This is a key characteristic of vector processors, as it determines the degree of parallelism that can be exploited during computation.

For example, if a vector processor supports a vector length of 4, it can process 4 data elements in parallel with each vector instruction. As the vector length increases, more data elements can be handled in parallel, leading to a potential reduction in execution time for certain types of data-parallel tasks.

However, the maximum vector length is a separate, hardware-specific limitation. It refers to the largest number of elements that can be processed simultaneously by the vector processor. This value is constrained by several factors, such as the width of the vector registers and the underlying hardware architecture. For example, a vector processor may have a maximum vector length of 128, meaning that no more than 128 elements can be processed in a single vector operation, regardless of how large the input data is.

The key difference between vector length and maximum vector length is that the vector length is a variable that can be adjusted based on the workload and the specific instruction being executed, while the maximum vector length is a fixed limit imposed by the hardware. The vector length can be chosen to balance between parallelism and resource utilization, but it will always be constrained by the hardware's maximum vector length. As the vector length increases, the benefits of vectorization become more apparent, but the performance gains will level off once the maximum vector length is reached, beyond which the processor cannot execute any more elements in parallel.

Understanding both the vector length and the maximum vector length is essential in optimizing vectorized applications, as choosing an optimal vector length can significantly impact performance, especially in terms of reducing execution time and maximizing throughput for large data sets.

4.1 Effect of Changing Vector Length

1. **Effect on Parallelism and Throughput**
   - **Increasing `vl`:**
     - Higher parallelism: More data elements are processed per instruction, improving throughput for vectorizable workloads.
     - Better utilization of vector units: Reduces the number of instructions needed for large datasets, increasing IPC (Instructions Per Cycle).
   - **Decreasing `vl`:**
     - Lower parallelism: Fewer elements are processed per instruction, potentially increasing loop overhead and reducing throughput.
     - May improve granularity: For irregular workloads, shorter vectors can reduce wasted computation (e.g., with masking).
2. **Effect on Memory Bandwidth Utilization**
   - **Increasing `vl`:**
     - Higher bandwidth demand: More data is fetched/stored per instruction, which can saturate memory bandwidth faster.
     - Improved spatial locality: Longer contiguous accesses may leverage burst transfers or prefetching more effectively.
   - **Decreasing `vl`:**
     - Lower bandwidth pressure: Reduces the risk of memory bottlenecks but may lead to smaller, less efficient accesses.
     - Potential for more scatter/gather overhead: Shorter vectors may increase address generation overhead.

3. **Effect on Power Consumption and Energy Efficiency**

   - **Increasing `vl`:**
     - Higher dynamic power: More lanes/units are active per cycle, increasing power draw.
     - Better energy efficiency (if utilized fully): Fewer instructions for the same work reduce control overhead, improving energy per operation.
   - **Decreasing `vl`:**
     - Lower dynamic power: Fewer active lanes reduce power consumption.
     - Potential inefficiency: More instructions and stalls may increase energy per task if parallelism is underutilized.

4. **Effect on Software Compatibility and Portability**

   - **Increasing `vl`:**
     - May break compatibility: Code tuned for shorter vectors might assume smaller registers (e.g., via hardcoded strip-mining).
     - Requires adaptive algorithms: Software should dynamically adjust to the available `vl` (e.g., using runtime checks).
   - **Decreasing `vl`:**
     - Easier portability: Shorter vectors are more likely to match legacy code assumptions.
     - May limit performance: Code optimized for longer vectors may underperform.

5. **Effect on Latency and Instruction Overhead**

   - **Increasing `vl`:**
     - Lower instruction overhead: Fewer decode/issue cycles for the same amount of work.
     - Higher tail effect latency: Partial vector fills (e.g., when data size $\neq$ multiple of `vl`) may waste cycles.
   - **Decreasing `vl`:**
     - Higher instruction overhead: More loops/strips needed, increasing loop control latency.
     - Finer-grained control: May reduce tail effect waste for small datasets.

6. **Effect on Hardware Complexity and Cost**

   - **Increasing `vl`:**
     - Higher complexity: Wider ALUs, more register file ports, and larger crossbars increase area and cost.
     - More aggressive memory subsystem: Needs higher bandwidth caches/interconnects.
   - **Decreasing `vl`:**
     - Simpler hardware: Narrower datapaths reduce area and cost.
     - May limit scalability: Less competitive for HPC/AI workloads requiring long vectors.

4.2 Impact of Changing the Maximum Vector Length

Changing the MVL implies a change in the hardware design or configuration of the vector unit in the RV64V processor. This could mean increasing or decreasing the width of vector registers or adjusting the supported data types. Below are the detailed effects on system performance:

1. **Effect on Parallelism and Throughput**
   - **Increase in MVL:**
     - A larger MVL means that more data elements can be processed in parallel during a single vector instruction. This can significantly improve throughput for data-intensive workloads, as fewer instructions are needed to process the same amount of data.
     - **Example:** If MVL increases from 8 to 16 for 32-bit elements, a single vector instruction can process twice as many elements, potentially halving the number of iterations or instructions required for a loop.
     - **Performance Gain:** Applications with high data parallelism (e.g., matrix operations, image processing) will see a performance boost due to reduced instruction overhead and better utilization of the vector unit.
   - **Decrease in MVL:**
     - A smaller MVL reduces the number of elements processed per instruction, leading to more instructions and iterations to handle the same workload.
     - **Performance Loss:** This can degrade performance, especially for workloads that are heavily vectorized, as the processor will need more cycles to complete the same task.
     - **Mitigation:** Software optimizations, such as loop unrolling or strip mining (breaking loops into manageable chunks based on the smaller MVL), can partially offset the performance loss.
2. **Effect on Memory Bandwidth Utilization**
   - **Increase in MVL:**
     - A larger MVL allows the processor to fetch and process more data in a single operation, improving memory bandwidth utilization if the memory subsystem can keep up with the demand.
     - **Potential Bottleneck:** If the memory system (e.g., cache or DRAM) cannot supply data fast enough, the performance gains from a larger MVL may be limited by memory latency or bandwidth constraints. This can lead to underutilization of the vector unit, as it spends more time waiting for data.
   - **Decrease in MVL:**
     - A smaller MVL reduces the data processed per operation, which may lower the demand on memory bandwidth. This could be beneficial in systems with constrained memory bandwidth, as it reduces contention and potential stalls.
     - **Trade-off:** While memory pressure is reduced, the overall computation time may increase due to the need for more instructions.
3. **Effect on Power Consumption and Energy Efficiency**
   - **Increase in MVL:**
     - Larger vector registers and wider data paths associated with a higher MVL typically consume more power due to increased hardware complexity and activity.
     - **Energy Efficiency:** If the workload is highly vectorized and the increased MVL leads to significantly fewer instructions, the energy per operation may improve (better performance per watt). However, if memory or other bottlenecks prevent full utilization of the vector unit, energy efficiency may degrade due to idle cycles.

- **Decrease in MVL:**
  - Reducing MVL lowers the power consumption of the vector unit, as smaller registers and narrower data paths are less power-hungry.
  - **Energy Efficiency:** This can improve energy efficiency for workloads that don't require high vector lengths, but it may increase total energy consumption for highly parallel tasks due to longer execution times.

4. **Effect on Software Compatibility and Portability**
   - **RISC-V Vector Extension Design:** One of the strengths of RVV is its "scalable vector architecture," which allows software to adapt to different MVLs without needing recompilation. The software queries the hardware's MVL using instructions like `vsetvl` and adjusts the vector length dynamically.
   - **Changing MVL:** While changing MVL doesn't break software compatibility (due to RVV's design), it can affect performance portability. Code optimized for a specific MVL may perform suboptimally on hardware with a different MVL.
   - **Example:** A loop optimized for an MVL of 16 may leave the vector unit underutilized if run on hardware with an MVL of 32, or it may require more iterations if run on hardware with an MVL of 8.
5. **Effect on Latency and Instruction Overhead**
   - **Increase in MVL:**
     - A larger MVL can reduce the number of instructions needed to process a dataset, lowering the instruction fetch and decode overhead. This can also reduce loop control overhead in vectorized loops.
     - **Latency Impact:** However, if individual vector operations have higher latency due to wider data paths or contention in shared resources (e.g., functional units), the per-instruction latency might increase.
   - **Decrease in MVL:**
     - A smaller MVL increases the number of instructions and loop iterations, raising the overhead of instruction fetching, decoding, and loop control.
     - **Latency Impact:** Smaller vector operations may have lower per-instruction latency, but the overall workload latency increases due to the higher instruction count.
6. **Effect on Hardware Complexity and Cost**
   - **Increase in MVL:**
     - Supporting a larger MVL requires wider vector registers, broader data paths, and potentially more complex control logic, increasing the hardware cost and area on the chip.
     - **Performance Trade-off:** While this can improve performance for vectorized workloads, it may come at the expense of resources that could be used for other features (e.g., more cores or larger caches).
   - **Decrease in MVL:**
     - Reducing MVL simplifies the hardware design, potentially freeing up resources for other components or reducing manufacturing costs.
     - **Performance Trade-off:** The performance for vectorized workloads may suffer, making the system less competitive for applications requiring high parallelism.

4.3 Stride and its Effects

Strides refer to the step size between elements when accessing data in memory, particularly during vector or SIMD (Single Instruction, Multiple Data) operations. Instead of always reading contiguous elements (as in regular arrays), strides allow skipping over a fixed number of memory locations

between accesses. Mathematically, if the base address is A, and the stride is S, then the addresses accessed in a vector operation are: [A, A + S, A + 2S, A + 3S, ...] This concept is critical in vector processors, GPUs, and SIMD units where multiple data points are processed in parallel.

4.3.1 Importance of Strides

Strides are useful because they enable efficient vector operations on non-contiguous memory patterns. For example:

- Accessing columns in row-major 2D matrices (common in C and C++), or rows in column-major matrices (used in Fortran, MATLAB).
- Subsampling or processing every Nth element (e.g., down sampling audio or images).
- Efficiently handling structured data layouts like structs or interleaved RGB data.

Strides also help maximize parallelism in vector machines while reducing instruction overhead and enabling better utilization of hardware memory bandwidth — provided the strides are chosen carefully to avoid bank conflicts and cache inefficiencies.

4.3.2 Bank Conflicts and Strid Pitfalls

When strides are powers of 2 and the number of banks is also a power of 2, they can align poorly, causing multiple accesses to hit the same bank — leading to serialized, slow memory operations according to this formula:

$$No.Banks/GCD(No.Banks, stride) < BankBusyTime$$

If the L.H.S, which correlates to the effective number of banks is less than the busy time then conflicts will appear, so in general we want the number of banks and the stride to be coprime with each other meaning GCD(stride, No. banks) = 1.

4.3.3 Performance Impact of Large Strides

- Large strides (like 32 or more) cause: Cache line underutilization (fetching 64-byte cache lines but using only a small part).
- Memory bandwidth waste: more loads for fewer useful data points.
- Bank conflicts: if the stride and bank count are not coprime.
- Latency increases: more frequent cache misses, longer access times.

4.4 Results vs. Expectations

The results of our study show a clear alignment with theoretical expectations regarding the impact of vector length on performance, particularly when using the SAXPY benchmark as the test operation.

From a theoretical perspective, increasing the vector length (VL) in a vector architecture should allow more data elements to be processed in parallel per instruction. This higher degree of parallelism is expected to improve throughput and reduce the total number of instructions needed to complete the task. As a result, the execution time—and by extension, the number of clock cycles—should grow more slowly compared to scalar (normal) architectures as the input size increases.

Our experimental findings support this behavior. While the number of clock cycles increases with larger input sizes for both scalar and vector architectures, the rate of increase is significantly lower

in the vector implementation. This confirms the theoretical advantage of vectorization: the ability to process larger chunks of data with fewer instructions, leading to better utilization of hardware and reduced control overhead.

In the case of the SAXPY benchmark—which is inherently data-parallel and well-suited to vector execution—this effect is especially pronounced. At larger vector lengths, the vector architecture demonstrates a significant performance advantage over the scalar counterpart, validating our expectation that such workloads benefit most from vector processing.

These results confirm that vector length is a critical factor in leveraging the performance potential of vector architectures, particularly for operations with high data-level parallelism like SAXPY.

V. Conclusion

In conclusion, vector architecture offers clear performance advantages over scalar architecture, particularly in workloads that involve repetitive operations on large datasets. Unlike scalar processors, which execute one instruction on one piece of data at a time, vector processors can apply a single instruction to multiple data elements simultaneously (SIMD – Single Instruction, Multiple Data). This parallelism significantly improves execution speed and efficiency, especially in fields such as scientific computing, signal processing, and artificial intelligence, where large-scale vector and matrix operations are common.

One key factor that further enhances vector performance is the vector length — the number of elements that can be processed in parallel. As the vector length increases, the processor can operate on more data elements with a single instruction, reducing loop iterations and instruction overhead. This leads to better utilization of processing resources, higher throughput, and more efficient execution of data-parallel workloads. Longer vector lengths also minimize control logic overhead, making execution more streamlined and predictable.

Moreover, vector architectures are better at exploiting memory bandwidth and reducing instruction fetch and decode operations, leading to improved energy efficiency and faster data processing. While scalar processors may still be suitable for general-purpose tasks or operations with low data parallelism, vector processors clearly outperform them in high-throughput, data-intensive applications.

Therefore, for scenarios that involve heavy vector or array operations, such as multimedia processing, scientific simulations, and machine learning — vector architecture is not only more efficient, but also more scalable and better aligned with the future direction of high-performance computing.

VI. Recommendations

- **Memory Access Pattern and Stride Control**
  - **Exploit Strided Loads/Stores (`vlse`/`vsse`)**: When processing non-contiguous data (e.g., AoS layouts or down-sampled vectors), keep the maximum vector length (MVL)

constant but vary the stride so that each memory transaction still fills the vector registers fully.

- **Convert Wide Strides to Unit Stride by Blocking**: Apply loop interchange or tiling so that the inner loop accesses elements consecutively; this allows cheap unit-stride loads and gives the cache prefetcher a chance to work.
- **Align Data to VLEN / Cache-Line Boundaries**: Use `__attribute__((aligned(64)))` to prevent split cache-line fetches when VLEN bytes are accessed.
- **Segmented Loads for SoA Layouts**: Replace manual pointer arithmetic with RVV segment instructions (`vlseg*`, `vsseg*`) to fetch several independent arrays in one instruction.

- **Vector Control (`vsetvli`) Minimisation**
  - **Loop Strip-Mining**: Place a single `vsetvli` outside the strip-mined loop and iterate with a fixed VL until fewer than VL elements remain.
  - **Mask the Tail Instead of Re-programming VL**: Use `vsetvli` once, then employ a tail-mask on the last iteration; this removes one control instruction per loop trip.
  - **Fuse Independent Kernels Sharing the Same VL**: Combine successive vector operations on the same data window (e.g., load–add–multiply) before changing VL.

- **Arithmetic Kernel Refinement**
  - **Use Fused Instructions**: Replace separate `vfmul` followed by `vfadd` with `vfmacc` when implementing SAXPY or GEMM micro-kernels.
  - **Reduction Instructions**: Employ `vredsum`, `vredmax`, etc., rather than scalar fold-back loops for dot products or norms.
  - **Widening/Narrowing Operations**: If input data are `int8` or `bf16`, use `vwmacc` or `vwadd` to keep accumulation in a wider format and defer down-cast to the end.

- **Loop-Level Scheduling**
  - **Software Pipelining**: Overlap load–compute–store stages across iterations; in RVV this often means issuing the store for chunk i while computing chunk $i-1$ and loading chunk $i+1$.
  - **Unroll by an Integer Multiple of VL**: Manual unrolling hides `vsetvli` latency further and exposes more ILP to the back end of superscalar cores.
  - **Prefetch via Dummy Strided Loads**: Issue a dummy `vlse` with zero destination mask two iterations ahead to warm up caches when hardware prefetchers are weak.

- **Data Layout Transformations**
  - **AoS to SoA Conversion**: Convert `struct{float x; float y; float z;}` into separate `x[]`, `y[]`, `z[]` arrays so that each vector register receives homogeneous data.
  - **Interleave Reads, De-interleave Writes**: For bidirectional stencils (e.g., 2-D kernels), read with `vlseg`+`vlse`, compute, then write back using `vsseg` to maintain a compact layout in memory.

- **Tail Handling Without Scalar Fall-back**
  - **Mask the Tail**: Let the final vector instruction operate on the partial mask generated by `vmsne.vi` to avoid any scalar clean-up code.
  - **Backward Iteration for Unknown Lengths**: Run the loop from the end toward the start when the data size is not a multiple of VL. The first iteration (which touches the tail) is then the only one that needs a mask.

Applying sections A–C typically yields the most immediate speed-ups because they attack memory bandwidth and instruction overhead simultaneously. Sections D and E remove the remaining scalar corner-cases, enabling a fully vectorised execution path.

Bibliography

[1] Riscvarchive. (n.d.). *RISCV-V-spec/V-spec.adoc at master · riscvarchive/RISCV-V-spec*. GitHub. Retrieved from https://github.com/riscvarchive/riscv-v-spec/blob/master/v-spec.adoc#example-stripmine-sew

[2] TheBeard. (2020, May 3). *Vector Architecture*. The Beard Sage. Retrieved from https://thebeardsage-com.translate.goog/vector-architecture/?_x_tr_sch=http&_x_tr_sl=en&_x_tr_tl=ar&_x_tr_hl=ar&_x_tr_pto=tc

[3] *The RISC-V ISA Simulator (Spike)* — Chipyard HEAD documentation. (n.d.). Retrieved from https://chipyard.readthedocs.io/en/latest/Software/Spike.html

[4] Latif, M. (2024, September 9). *Mastering RISC-V: Setting up Spike Simulator for seamless Development*. Medium. Retrieved from https://medium.com/@latifbhatti012/mastering-risc-v-setting-up-spike-simulator-for-seamless-development-b54eceb25d12

[5] Hennessy, J. L., & Patterson, D. A. (n.d.). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publications.

[6] GeeksforGeeks. (2023, February 14). *Vector Processor vs Scalar Processor*. Retrieved from https://www.geeksforgeeks.org/vector-processor-vs-scalar-processor/