Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica - Scienza e Ingegneria
Master's Degree in Computer Engineering

Accelerating Computational Fluid Dynamics on RISC-V: Vectorization of OpenFOAM's Multigrid Solver

Master's Thesis in Computer Architecture M

Supervisor:
Prof.
Andrea Bartolini

Presented by:
Gabriele Ceccolini

Co-supervisors:
Dr.
Federico Ficarelli
Filippo Barbari
Dr.
Simone Bnà

March 2026 session
Academic Year 2024/2025

A mio nonno Ferdinando ...

Abstract

Computational Fluid Dynamics (CFD) applications rely heavily on memory-bound sparse linear algebra kernels like Sparse Matrix-Vector multiplication (SpMV). Legacy frameworks such as OpenFOAM exploit MPI parallelism but lack effective support for modern vector architectures, largely due to internal matrix formats that prevent contiguous memory accesses.

With the emergence of the RISC-V Vector Extension (RVV), vector architectures offer a compelling alternative to traditional SIMD and GPUs. This thesis investigates this potential by optimizing a state-of-the-art OpenFOAM multigrid solver across two contrasting RISC-V architectures: the long-vector EPAC accelerator prototype and the commercial short-vector Sophon SG2044 processor.

To overcome native memory bottlenecks, the main contributions of this work include: i) vectorizing the SpMV kernel using RVV intrinsics and vector-friendly sparse formats; ii) developing a custom smoother plugin for OpenFOAM that performs runtime data conversion; and iii) a comprehensive speedup and scalability evaluation scaling the problem size for the isolated SpMV kernel, the custom smoother, and the end-to-end CFD simulation.

Experimental results demonstrate a 6x speedup for the custom smoother on the EPAC test chip and a 1.5x speedup on the SG2044. Furthermore, end-to-end CFD simulation evaluations reveal overall performance improvements of 1.62x and 1.16x on the EPAC and SG2044 architectures, respectively. Ultimately, this work proves that legacy CFD codes can be successfully modernized and accelerated using emerging RISC-V hardware.

Contents

List of Figures **v**

1 Introduction **1**

1.1 Problem introduction and motivation 1
1.2 Vector architectures 3
1.3 OpenFOAM framework 4
1.4 Contributions 5

2 SIMD, Vector processors and EPAC accelerator **7**

2.1 Vector Processors 9
2.1.1 Key Architectural Features 9
2.1.2 Multi-lane Implementation 10
2.1.3 Comparison with Other Data-Level Parallelism (DLP) Paradigms 10
2.2 RISC-V and the Vector Extension 12
2.3 Long Vector Architectures and the European Processor Accelerator (EPAC) Accelerator 19
2.3.1 Motivation for long vector architectures 19
2.4 EPAC 1.5 Accelerator Architecture 22
2.4.1 The VEC Tile Architecture 23

3 Numerical Foundations of Computational Fluid Dynamics **27**

3.1 Problem discretization 27
3.2 Iterative solvers 30

3.2.1 Direct vs iterative methods 30
3.2.2 The Conjugate Gradient Method 31
3.3 Multigrid method 34
3.4 Introduction to OpenFOAM framework 36

4 Sparse matrix formats and Sparse Matrix-Vector multiplication (SpMV) vectorization 39
4.1 Basic sparse matrix formats 40
4.1.1 Coordinate List (COO) 40
4.1.2 Compressed Sparse Row (CSR) 41
4.2 Vector friendly sparse matrix formats 43
4.2.1 ELLPACK (ELL) 43
4.2.2 SELL-C-σ (SELL-C-σ) 44
4.3 SpMV Evaluation: Long-Vector vs. Short-Vector Architectures 48

5 End-to-End Acceleration of CFD Simulations: Smoother Integration and Evaluation 53
5.1 Simulation set-up and profiling of scalar execution 54
5.1.1 Anatomy of the Richardson smoother 56
5.2 Runtime format conversion and plugin integration 58
5.3 Final profiling and application speedup 59
5.3.1 Smoother standalone performance 59
5.3.2 Amdahl's Law and theoretical speedup bounds 61
5.3.3 End-to-end simulation profiling 63

6 Conclusions and Future Work 67

Bibliography 73

List of Figures

1.1 Arithmetic intensity of some common computational kernels [14]. 2
1.2 EPAC1.0 Test Chip in GF22 Technology [9] 4
1.3 OpenFOAM framework Graphical User Interface (GUI) . . . 5

2.1 The four categories of Flynn's computer architecture taxonomy. 7
2.2 Single lane vs multi lane vector processor [14] 11
2.3 RV64V Architecture [14] 13
2.4 Structure of an RISC-V Vector Extension (RVV) Register [10] 14
2.5 Register grouping using the Length Multiplier (LMUL) parameter [10] 15
2.6 Code comparison: DAXPY implementation. Targets: RISC-V Scalar (GCC -O2 `-fno-tree-vectorize`), RVV (Clang -O2 `-mllvm -prefer-predicate-over-epilogue=predicate-dont-vectorize`), x86-64 Advanced Vector Extensions (AVX) (GCC -O3 `-mavx2`). Note how RVV elegantly handles arbitrary memory lengths within a single compact loop via `vsetvli`, whereas AVX requires fixed-width loop increments and a separate scalar tail loop. The AVX assembly is simplified for readability. 17
2.7 Comparison between RVV, AVX-512, Arm Scalable Vector Extension (SVE), NEC SX ISA [10] 19
2.8 Short vectors vs Long vectors [6]. 20

2.9 Performance analysis of the SpMV kernel under various memory latency and bandwidth constraints [32]. 21
2.10 EPAC 1.5 accelerator tape-out with multi-tile structure. [20] . 22
2.11 EPAC accelerator multi-tile structure with focus on the VEC tile architecture. [20]. 23
2.12 Vitruvius+ architecture [21] 24
2.13 EPAC VPU VL_{max} versus other different architectures [20] . . 25

3.1 Example of an unstructured computational mesh for Finite Volume Method (FVM) discretization [13]. 28
3.2 From transport equation to sparse linear system [13]. 30
3.3 Steepest descent versus Conjugate Gradient (CG) trajectory [15]. 32
3.4 Multigrid (MG) method [13]. 34
3.5 Open Field Operation and Manipulation (OpenFOAM) framework structure [19]. 36
3.6 Comparison between OpenFOAM syntax (left) and the continuous Navier-Stokes equation (right). 37

4.1 Arithmetic Intensity of main High-Performance Computing (HPC) kernels [4]. 40
4.2 COO sparse matrix format [23]. 41
4.3 CSR sparse matrix format [23]. 42
4.4 CSR and ELL sparse matrix formats. Arrows indicate the order of elements in memory.[18]. 44
4.5 SpMV operation of ELL format [18]. 45
4.6 SELL-C-σ sparse matrix format [18]. 45
4.7 Overview of the Computational Fluid Dynamics (CFD) test case setup, showing the solid cylinder geometry (a) and the corresponding mesh resolution blocks (b).[24] 48
4.8 Non-zero distribution on the $cylinder_2k$ matrix. 49

4.9 SpMV speedup and percentage of Vector Processing Unit (VPU) stalls on EPAC. 50
4.10 SpMV speedup and cache miss rates on SG2044. 51

5.1 ParaView[3] visualization of the velocity magnitude ($|U|$) for the flow past a cylinder test case in OpenFOAM. 55
5.2 EPAC scalar application profiling 56
5.3 Execution trace from Paraver[5], a performance analysis and visualization tool, highlighting the execution time breakdown of the smoother's internal sections: SpMV, vector updates, and initialization. 57
5.4 Paraver execution trace showing the time distribution of the Geometric-Algebraic MultiGrid (GAMG) solver phases. The setup phase overhead is safely amortized over the repeated smoother executions. 58
5.5 Performance profiling of the vectorized Richardson smoother on the EPAC architecture (128k cells problem). 60
5.6 Performance profiling of the vectorized Richardson smoother on the SG2044 architecture (128k cells problem). 61
5.7 Theoretical maximum speedup according to Amdahl's Law compared to the realized application speedup. 63
5.8 Application profiling on EPAC 64
5.9 Application profiling on SG2044 64

Chapter 1

Introduction

1.1 Problem introduction and motivation

A significant portion of HPC applications involves the simulation of physical systems, such as fluid dynamics. These applications have historically been valuable for both scientific and engineering purposes, allowing within certain limits for the prediction of a fluid's evolution (often air) and its properties in a much more economical and rapid manner compared to physical experiments, such as wind tunnels.

Due to the intrinsic nature of the phenomena they simulate, these simulations are computationally highly demanding. Consequently, they require HPC systems to simulate scenarios of real-world interest effectively.

From a computational perspective, these applications differ fundamentally from other prevalent HPC workloads. Unlike Artificial Intelligence (AI) applications, which predominantly rely on dense linear algebra and often leverage reduced numerical precision, CFD simulations depend heavily on sparse linear algebra requiring high numerical precision (64-bit Floating-Point (FP64)). Consequently, these workloads exert significantly higher pressure on system memory bandwidth compared to dense algebra applications, often shifting the performance bottleneck from raw computation to data movement.

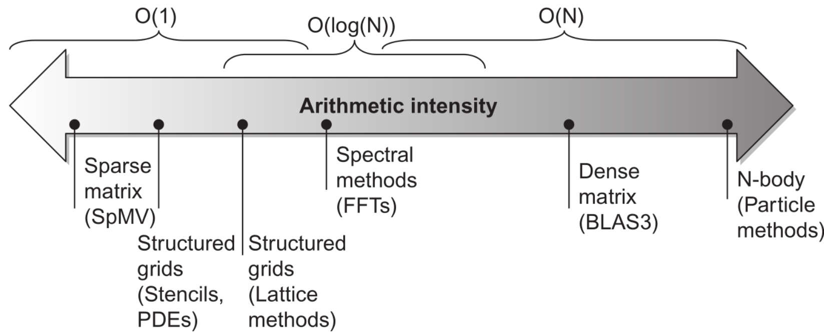


Figure 1.1: Arithmetic intensity of some common computational kernels [14].

Mathematically, these problems are initially formalized by partial differential equations related to fluid physics (such as the Navier-Stokes equations). Through numerical methods, they are discretized into simple linear systems. Consequently, the problem reduces to the resolution of these linear systems, which are typically very large and sparse. To solve these systems, the most efficient approach is not found in classical direct methods (e.g., Lower-Upper (LU) decomposition, Gaussian elimination, etc.) commonly used for dense matrices but rather iterative methods [27]. These methods are termed "iterative" because they involve executing a sequence of consecutive iterations such that, in the limit, the solution vector converges to the true solution vector. From a computational perspective, a crucial aspect is that these methods rely heavily on SpMV. It is precisely this specific kernel that must be targeted to accelerate this type of application.

CFD simulations, therefore, serve as an excellent benchmark for evaluating parallel computing systems in low arithmetic intensity regimes. This is exemplified by the High Performance Conjugate Gradients (HPCG) [8] benchmark, which is designed to simulate precisely these workloads, in contrast to its dual, the more famous High-Performance Linpack (HPL) [26], which evaluates dense algebra workloads characterized by high arithmetic intensity.

1.2 Vector architectures

Within the parallel computing landscape, various programming paradigms are adopted depending on the architectural nature of the underlying system. Currently, the standard approach in HPC involves a combination of distinct parallelism levels: distributed parallelism via Message Passing Interface (MPI) across multiple physical nodes, fixed-length Single Instruction, Multiple Data (SIMD) parallelism (e.g., Intel AVX [16], Arm Neon [2]), and Graphics Processing Unit (GPU)-based acceleration (e.g., Compute Unified Device Architecture (CUDA) [22], Single-source C++ Heterogeneous Programming (SYCL) [17]).

This highly heterogeneous technology stack presents significant challenges in software development. GPU programming typically relies on decomposing workloads into thousands of threads managed by complex hardware scheduling, often requiring the management of disjoint memory spaces. Conversely, fixed-length SIMD technologies, such as AVX, suffer from inherent rigidity. By exposing the vector length directly within the instruction set, they lack versatility; any change in the hardware vector width necessitates code recompilation to maintain efficiency.

To address these limitations, the HPC landscape is currently witnessing a renaissance of Vector Length Agnostic (VLA) architectures. This model relies on the integration of dedicated vector registers within the processor, paired with vector functional units capable of natively performing operations on vector operands. A crucial feature of this paradigm is that the number of elements processed in a given operation is determined at runtime, rather than being fixed in the instruction set.

Historically rooted in the seminal Cray supercomputers, this paradigm has been revitalized by modern systems such as the Fujitsu A64FX (which powers the Fugaku supercomputer [28]), NEC SX-Aurora TSUBASA [30], and EPAC.

The primary architectural target selected for this thesis is EPAC, developed within the European Processor Initiative (EPI). Built upon RVV,

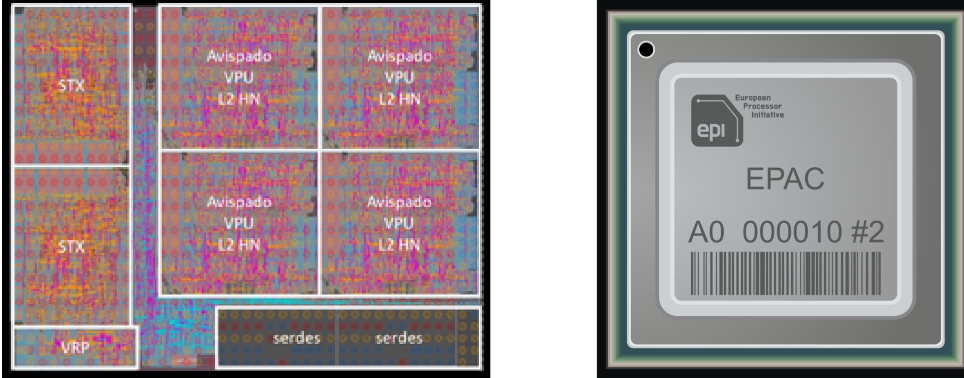


Figure 1.2: EPAC1.0 Test Chip in GF22 Technology [9]

EPAC implements a highly aggressive long-vector architecture.

Complementing this, the research also targets the Monte Cimone RISC-V cluster, a multi-node infrastructure powered by commercial RISC-V processors featuring vector support.

1.3 OpenFOAM framework

The OpenFOAM framework was selected as the primary application target for this thesis. OpenFOAM is an open-source software for CFD, it is extensively utilized across both industrial and academic sectors. Currently, OpenFOAM scales from individual workstations to large-scale supercomputers, relying on MPI for distributed parallelism via domain decomposition.

However, hardware acceleration support remains limited. GPU acceleration is still in an experimental phase; while initiatives such as SPUMA [7] (developed by CINECA) exist, they have not yet been integrated into the standard, upstream release of the software. Similarly, support for fixed-length SIMD extensions (e.g., Intel AVX, Arm Neon) is constrained, relying primarily on compiler auto-vectorization rather than explicit optimization. Crucially, the framework's internal data structures lack specific memory layouts designed to facilitate vectorization. Consequently, the objective of implementing and exploring the vectorization of key framework kernels for

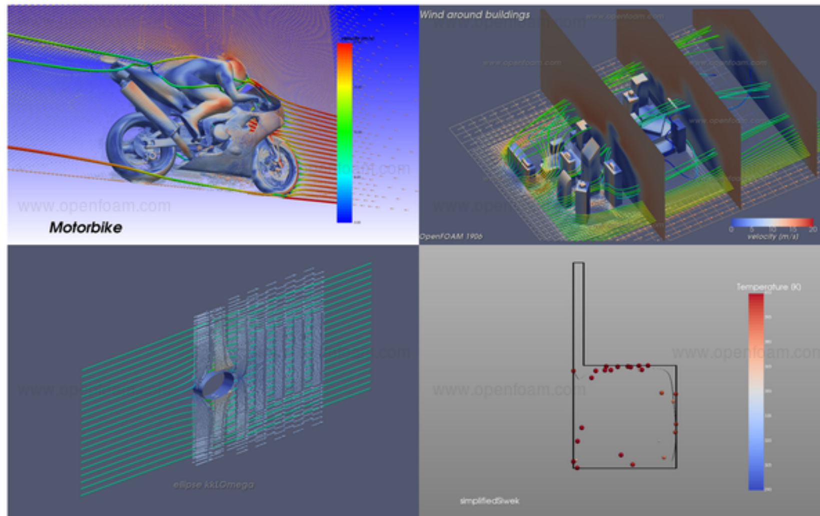


Figure 1.3: OpenFOAM framework GUI

RISC-V Vector architectures serves as a compelling case study and a significant technological demonstrator.

1.4 Contributions

The objective of this project was to evaluate the feasibility of accelerating OpenFOAM's multigrid solver, named GAMG, through vectorization based on RVV intrinsics.

The central hypothesis was that by transitioning matrix storage from the default format to a vector-friendly layout, and by isolating and vectorizing the most computationally significant sections using RVV intrinsics, a net performance gain could be achieved, outweighing the overhead introduced by format conversion.

The main contributions of this thesis are the following:

i) The vectorization of SpMV kernel in different storage formats;

ii) The integration of a fully vectorized smoother into OpenFOAM's multigrid solver;

iii) Performance benchmarking, microarchitectural profiling, and scalability study of both SpMV's microkernels and full-scale OpenFOAM simulation applications leveraging the optimized multigrid solver.

In terms of experimental results, our intrinsic-based vectorization approach yielded significant performance improvements. Specifically, for the isolated vectorized smoother, we achieved a 6x speedup on the EPAC processor and a 1.5x speedup on the SG2044 processor. When evaluating the complete end-to-end simulation, the overall speedup reached 1.62x on EPAC and 1.16x on SG2044, compared to the respective scalar baseline versions. This demonstrates that substantial performance gains can be realized by adopting alternative data formats compared to OpenFOAM's defaults and by leveraging intrinsic-based vectorization.

Chapter 2

SIMD, Vector processors and EPAC accelerator

As mentioned before, nowadays parallelism is exploited in current architectures in many forms. To systematize these diverse implementations, the possible parallelism paradigms are described by the taxonomy introduced by Michael Flynn in 1966 [11]. This classification categorizes computers based on two independent dimensions: parallelism in the *Instruction Stream* and parallelism in the *Data Stream*. The combination of these two streams generates four main categories:

Data stream \ Instruction stream	Single	Multiple
Single	SISD	MISD
Multiple	SIMD	MIMD

Figure 2.1: The four categories of Flynn's computer architecture taxonomy.

- **Single Instruction, Single Data (SISD):** This category represents the traditional sequential uniprocessor. At any given moment, a single instruction acts on a single data stream. Although seemingly simple, modern SISD processors heavily exploit Instruction-Level Parallelism (ILP) through techniques such as pipelining, speculative execution, and superscalar execution, while maintaining the illusion of sequential execution for the programmer.

  Examples: Intel Pentium.

- **SIMD:** In this model, a single instruction is sent to multiple processing units, which execute the same operation on different data elements simultaneously. This approach exploits DLP. It is particularly energy-efficient as it reduces the overhead of instruction fetch and decode: a single instruction controls multiple arithmetic operations.

  Examples: Vector architectures (RVV, SVE), GPUs, fixed-length SIMD (AVX, Arm Neon (NEON)).

- **Multiple Instruction, Single Data (MISD):** In this architecture, multiple instructions operate independently on the same data stream. It is a theoretical category that has found little commercial practical application.

- **Multiple Instruction, Multiple Data (MIMD):** Each processor in the system executes its own instruction stream on its own data independently. This is the most flexible and general model, capable of exploiting Thread-Level Parallelism (TLP). However, it is generally more expensive and complex to manage than SIMD.

  Examples: Modern multicore processors.

In the following section, we focus on vector processors within the context of the SIMD paradigm, highlighting similarities and differences with their fixed-length and GPU counterparts.

2.1 Vector Processors

The main idea behind vector processors is to exploit DLP by amortizing the overhead of instruction fetch and decode. Unlike the scalar execution model, which operates on single values, vector architectures process vectors containing multiple elements through a single instruction. This paradigm shifts the focus from optimizing the execution of individual instructions to sustaining a high throughput of data operations.

2.1.1 Key Architectural Features

A vector processor is characterized by three fundamental architectural pillars that distinguish it from traditional scalar processors:

- **VLA:** Unlike fixed-width SIMD units that expose the hardware register size to the programmer, vector architectures decouple the instruction set from the physical implementation. The hardware provides a Vector Length Register (vl) that controls the number of elements processed. This allows the same binary code to run on hardware with different vector register sizes, adapting dynamically to the available parallelism without recompilation.

- **Flexible Memory Access:** Vector processors are designed to operate efficiently on non-contiguous data structures, a common occurrence in scientific computing (e.g., sparse matrices in CFD). While traditional SIMD often requires data to be packed contiguously or relies on often inefficient gather/scatter implementations, vector architectures are optimized to natively support strided access and *gather/scatter* operations (indexed loads/stores). This capability allows the processor to fetch distributed data elements directly into vector registers, effectively decoupling the memory layout from the processing logic.

- **Amortized Instruction Overhead:** While scalar processors also utilize pipelined functional units, vector architectures exploit them to

amortize the control overhead. A single vector instruction initiates a sequence of operations on multiple elements (controlled by vl). This implies that the cost of fetching and decoding the instruction is paid once but amortized over the entire length of the vector. Consequently, the processor spends a larger fraction of cycles performing actual computation rather than control logic, sustaining high throughput even with deep pipelines.

Even with a basic single-lane implementation, this model drastically reduces instruction bandwidth, as fewer instructions need to be fetched and decoded to process large workloads.

2.1.2 Multi-lane Implementation

The natural evolution of the single-lane model involves complementing the existing temporal parallelism (pipelining) with spatial parallelism. By replicating functional units, the architecture creates parallel *lanes*.

In a multi-lane configuration, the vector register file and functional units are distributed across lanes, allowing the processor to compute multiple data elements simultaneously within the same clock cycle. Once the initial pipeline startup latency is overcome, the system can sustain a throughput significantly higher than one result per cycle, scaling with the number of lanes.

2.1.3 Comparison with Other DLP Paradigms

To better contextualize vector architectures, we compare them against two distinct approaches to exploiting DLP:

- **Fixed-Length SIMD (e.g., Intel AVX, ARM NEON):** This model exposes the vector length through specific Instruction Set Architecture (ISA) variants or binary encodings (e.g., 128-bit or 256-bit registers). While hardware implementation is straightforward, it introduces code rigidity: porting to a wider architecture typically requires

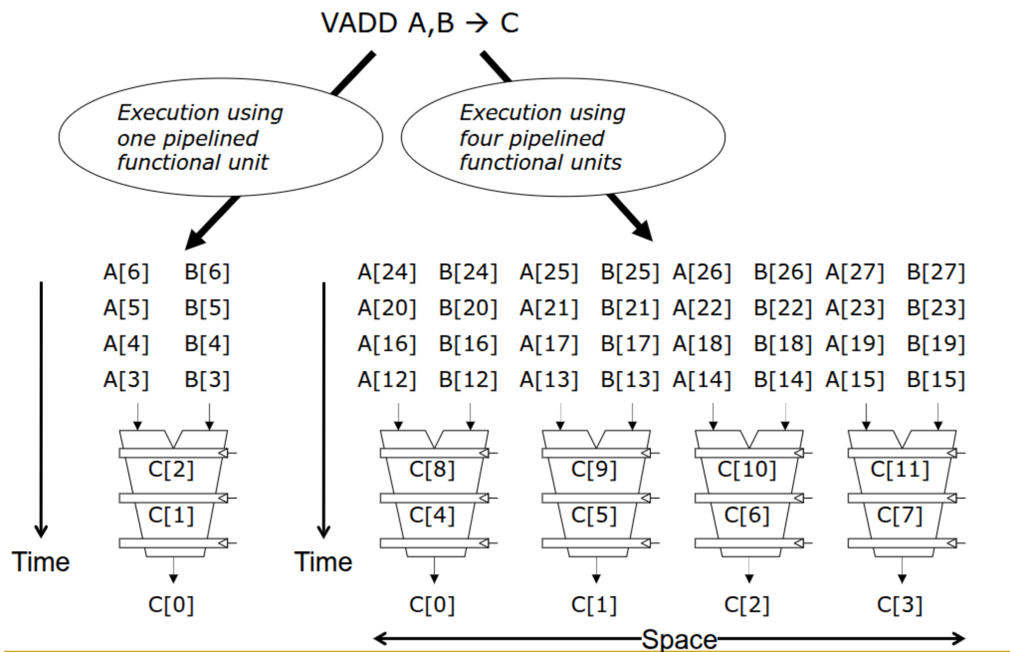


Figure 2.2: Single lane vs multi lane vector processor [14]

recompilation. Furthermore, loops require explicit "tail" or "remainder" code to handle iteration counts that are not multiples of the vector width. In contrast, vector architectures utilize hardware-supported *strip-mining* via the *vl* register (e.g., using `vsetvli` in RISC-V), eliminating the need for manual tail loops and ensuring binary compatibility across different hardware implementations.

- **Graphics Processing Units (GPU / Single Instruction, Multiple Threads (SIMT)):**
  GPUs utilize the SIMT model. SIMT can be viewed as an evolution of Flynn's SIMD taxonomy: it combines the efficiency of hardware-level vector execution with the flexibility of a multi-threaded abstraction. While the hardware executes instructions in lockstep (grouped into "warps"), the programming model follows the Single Program, Multiple Data (SPMD) paradigm. In SPMD, the programmer writes code for a single scalar thread, and the hardware/runtime spawns thousands

of instances to process different data elements in parallel. While effective for massive throughput, this model relies on complex hardware scheduling to hide latency.

Among these paradigms, vector architectures offer superior architectural elegance and generality. By decoupling the programming model from the hardware implementation details (specifically register width), they ensure VLA. This allows the same program to execute on different hardware implementations, a key advantage for the longevity and portability of scientific codes.

2.2 RISC-V and the Vector Extension

RISC-V [33] is an open standard ISA based on established Reduced Instruction Set Computer (RISC) principles. Unlike proprietary ISAs, it is free regarding licensing fees, enabling broad adoption in both academic and industrial settings. Its design is fundamentally modular: it consists of a minimal base integer ISA combined with optional standard extensions. This allows architects to tailor the hardware implementation to specific domain requirements.

Within this ecosystem, the RVV was introduced as a vector-length agnostic extension to the RISC-V ISA. It is designed as a general-purpose approach, allowing for highly diverse hardware implementations.

The architectural state added by RVV includes:

- **Vector Registers:** 32 vector registers, named `v0` through `v31`, each with a width of `VLEN` bits.
- **Vector Functional Units:** Pipelined functional units capable of operating on vectors (e.g., Floating-Point (FP), Integer, Logic) completely decoupled from memory access.
- **Vector Load/Store Unit:** Specialized hardware responsible for transferring data between memory and vector registers. It handles complex

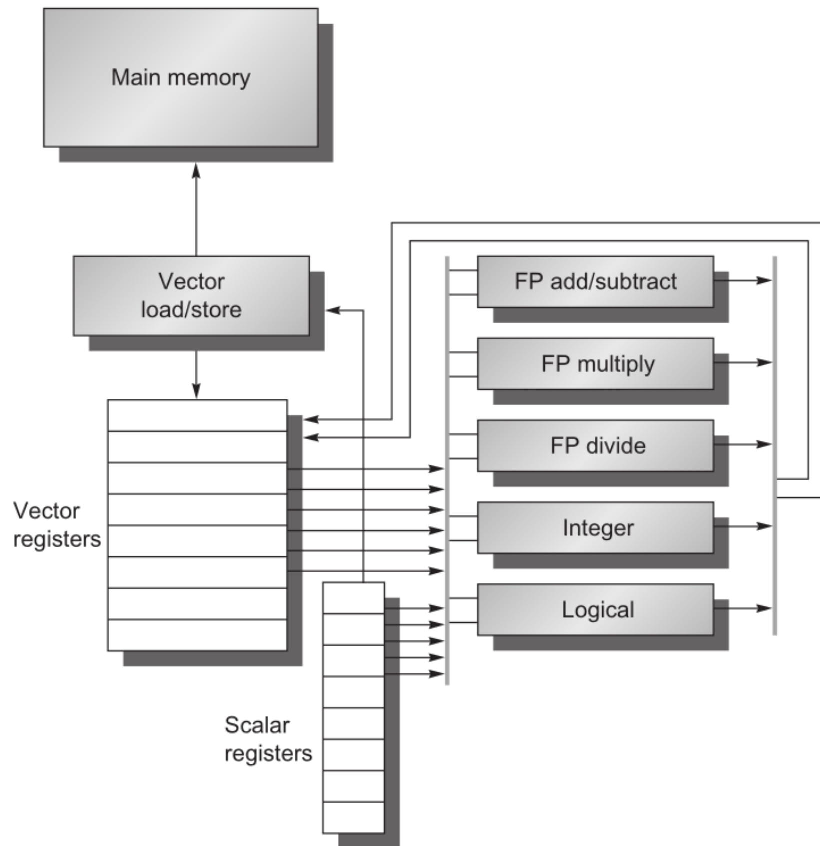


Figure 2.3: RV64V Architecture [14]

access patterns (stride, indexed scatter/gather) and manages the high bandwidth required to feed the functional units.

Vector registers are characterized by several key parameters. Some are constant hardware parameters defined by the implementer (Vector Register Length (VLEN), Element Length (ELEN)), while others are runtime configurations stored in CSRs, specifically `vl` and `vtype`.

Constant Parameters:

- **VLEN**: The width of a vector register in bits. It is a power-of-two constant chosen by the implementer (typically ≥ 128 bits).

- **ELEN**: The maximum size of a single vector element in bits. RVV

elements range from a minimum of 8 bits up to ELEN (usually 32 or 64 bits). It must be a power of two, with $8 \leq \text{ELEN} \leq \text{VLEN}$.

Runtime Parameters:

- `vtype`: A CSR that describes the data format for the current operation. It includes:
  - **Selected Element Width (SEW)**: The size in bits of the elements currently being processed ($8 \leq \text{SEW} \leq \text{ELEN}$).
  - **LMUL**: A configuration that allows grouping multiple vector registers together.
- `vl` (Vector Length): Specifies the number of elements to be processed by the vector instruction.

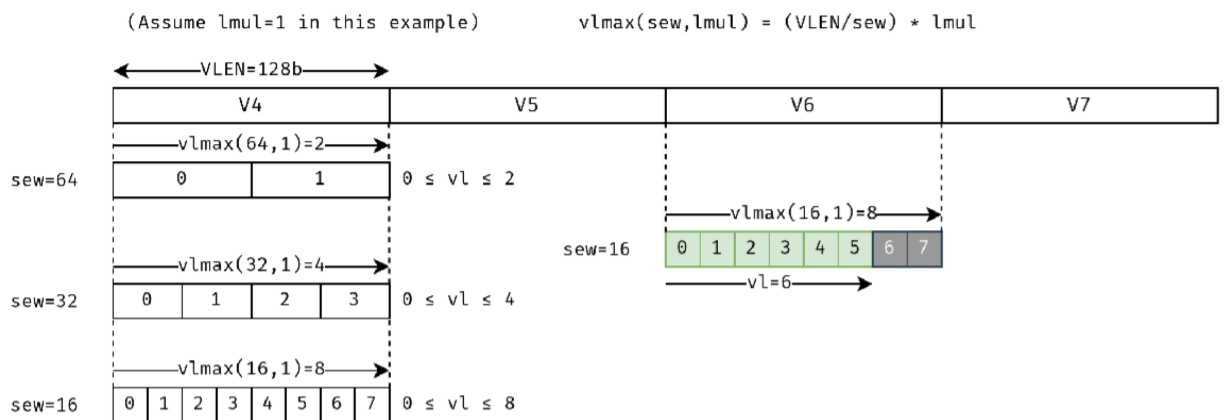


Figure 2.4: Structure of an RVV Register [10]

To harmonize operations across different element sizes or to operate on logical vectors larger than the physical registers, the LMUL parameter can be adjusted:

- When LMUL = 1, operations use exactly one vector register group (one physical register).

- When LMUL < 1 (Fractional LMUL), operations use a fraction of a vector register (LMUL $\in \{1/2, 1/4, 1/8\}$), allowing for more effective register utilization with small data types.

- When LMUL > 1, the operation uses a vector group of LMUL consecutive vector registers. For example, with LMUL = 2, there are 16 vector groups, addressed by even indices: `v0`, `v2`, `v4`, ..., `v30` (where `v0` logically includes `v0` and `v1`).

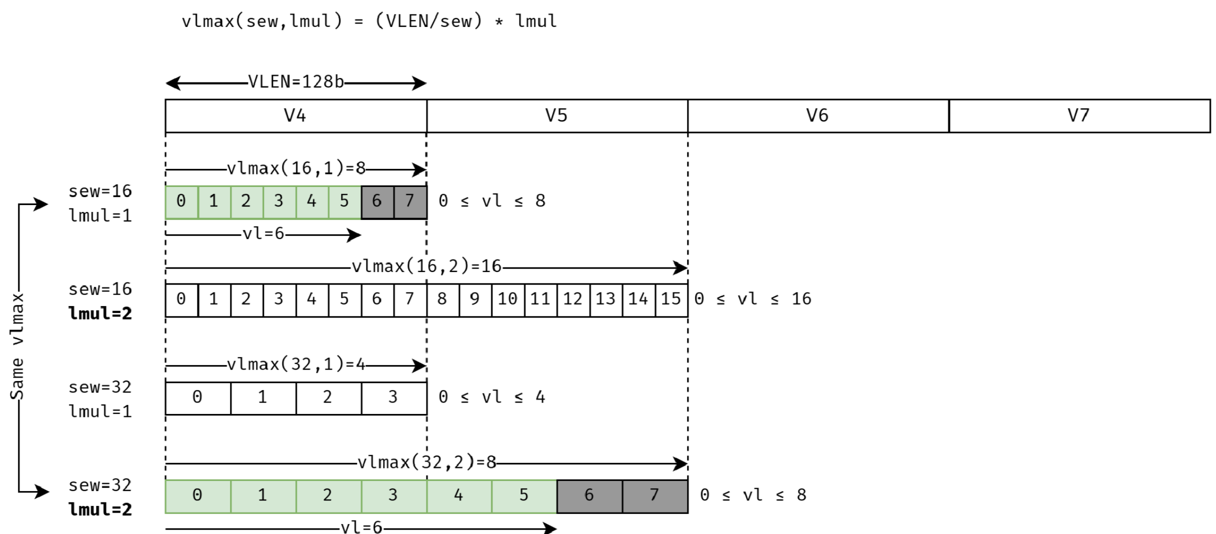


Figure 2.5: Register grouping using the LMUL parameter [10]

Several key instructions form the backbone of RVV operations. These include standard vector-vector arithmetic instructions, such as **`vadd`** (vector add) and **`vmul`** (vector multiply).

Regarding memory access, RVV supports versatile addressing modes to efficiently handle diverse data layouts:

- **Unit-stride loads** (`vle`): For contiguous memory access. Functionally, these are a special case of strided loads where the stride equals the element size (stride=1).

- **Strided loads** (`vlse`): For non-contiguous access where elements are separated by a constant distance (stride) in memory.

- **Indexed loads** or "Gather" (`vluxei`): For loading elements from arbitrary memory locations specified by a vector of indices. (e.g., indirect memory access like $A[B[i]]$).

However, the most distinctive feature of the ISA is arguably the configuration instruction, `vsetvli`. This instruction allows the application to dynamically set the number of elements to be processed by subsequent vector instructions at any point in the execution flow. This mechanism is the key to the vector-length agnostic programming model, as illustrated in the following example.

Scalar C implementation of $y[i] = a \cdot x[i] + y[i]$

```
// Standard DAXPY implementation
void daxpy(size_t n, double a, const double *restrict x, double *restrict y)
    {
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

RISC-V Scalar
(One element per iter)

```
daxpy:
    beqz  a0, .L_end    # N==0?
      exit
.L_loop:
    fld   fa5, 0(a1)    # Load
     X[i]
    fld   fa4, 0(a2)    # Load
     Y[i]

    # Y = Y + a * X
    fmadd.d fa5,fa0,fa5,fa4

    fsd   fa5, 0(a2)    # Store
      Y[i]
    addi  a1, a1, 8     # X ptr
      +=8
    addi  a2, a2, 8     # Y ptr
      +=8
    addi  a0, a0, -1    # N--
    bnez  a0, .L_loop   #
     Repeat
.L_end:
    ret
```

RISC-V Vector (RVV)
(Vector Length Agnostic)

```
daxpy:
    beqz     a0, .L_end # N==0?
      exit
    li       a3, 0      # i = 0
.L_loop:
    # Set vl & config (LMUL=2)
    vsetvli a4,a0,e64,m2,ta,ma
    slli     a5, a3, 3  # i * 8
    add      a6, a1, a5 # &X[i]
    add      a5, a5, a2 # &Y[i]
    vle64.v v8, (a6)    # Load
      vec X
    vle64.v v10, (a5)   # Load
      vec Y
    sub      a0, a0, a4 # N -= vl
    # Y = Y + a * X
    vfmacc.vf v10,fa0,v8
    vse64.v v10, (a5)   # Store
      vec Y
    add      a3, a3, a4 # i += vl
    bnez     a0, .L_loop# Loop if
        N!=0
.L_end:
    ret
```

x86-64 Intel AVX
(Fixed 256-bit width)

```
daxpy:
    mov      rcx, rdi
    and      rcx, -4     # N mult
     4
    xor      rax, rax    # i = 0

.L_vector:
    # -- MAIN LOOP (4 elems) --
    vmovupd ymm1, [rsi+rax*8]
    vfmadd213pd ymm2,ymm0,ymm1
    vmovupd [rdx+rax*8], ymm2
    add      rax, 4      # i += 4
    cmp      rax, rcx
    jl       .L_vector

    cmp      rax, rdi    # Tail?
    jge      .L_end

.L_tail:
    # -- TAIL LOOP (1 elem) --
    vmovsd  xmm1, [rsi+rax*8]
    vfmadd213sd xmm2,xmm0,xmm1
    vmovsd  [rdx+rax*8], xmm2
    inc      rax         # i += 1
    cmp      rax, rdi
    jl       .L_tail
.L_end:
    ret
```

Figure 2.6: Code comparison: DAXPY implementation. Targets: RISC-V Scalar (GCC -O2 `-fno-tree-vectorize`), RVV (Clang -O2 `-mllvm -prefer-predicate-over-epilogue=predicate-dont-vectorize`), x86-64 AVX (GCC -O3 `-mavx2`). Note how RVV elegantly handles arbitrary memory lengths within a single compact loop via `vsetvli`, whereas AVX requires fixed-width loop increments and a separate scalar tail loop. The AVX assembly is simplified for readability.

The example presented above represents a classic dense linear algebra kernel, widely used in scientific computing. The assembly code clearly demonstrates the architectural efficiency and flexibility of the RVV approach compared to both scalar execution and traditional fixed-width SIMD extensions like AVX.

We can identify two macro-advantages:

- **Instruction Efficiency:** RVV drastically reduces the dynamic instruction count compared to the scalar version. By processing `vl` elements per instruction, the cost of loop control (pointer arithmetic, counters, and comparisons) is amortized over the vector length. This implies that the loop body is executed roughly N/vl times fewer, consequently reducing the number of loop-control branches that the hardware must predict and execute.

- **VLA and Portability:** Since `vl` is adjusted at runtime based on the hardware implementation (`VLEN`) and the remaining elements, the code dynamically adapts to the available hardware parallelism. Crucially, the same binary code is forward-compatible: it can run without recompilation on future architectures with larger `VLEN`, inherently exploiting the increased vector width. This contrasts with fixed-width ISA extensions (e.g., moving from AVX2 to AVX-512), which often require recompilation or code duplication to leverage wider registers. Furthermore, thanks to the hardware-managed vector length via `vsetvli`, RVV simplifies the control flow by eliminating the need for explicit "cleanup" loops (tails) to handle non-aligned element counts.

Driven by these considerations, the clear efficiency gains, and the open nature of the architecture, the EPI has selected the RISC-V Vector agnostic model for its vector accelerator, which we will analyze in the next section.

Feature	Intel AVX-512	Arm SVE	NEC SX-Aurora TSUBASA	RISC-V Vector
Is the vector register size defined by the architecture?	**Yes**. 512 bit VLE extension allows using 128-bit (SSE) and 256-bit (AVX-2) registers.	**No**. From 128 bit to 208 bits (in multiples of 128 bits)	**Yes**. Current generation is 16,384 bits.	**No**. Powers of two, from 64/128 up to 65,536 bits.
Predication/Masking	**Yes**. 8 mask registers `k0-k7` (`k0` hardcoded to all ones)	**Yes**. 16 vector predicate registers `p0-p15`. (`p0-p7` masking, `p8-p15` loops)	**Yes**. 16 vector mask registers.	**Yes**. Only `v0` as an implicit operand if the instruction is masked.
Set vector length	**No**	**No** (Privileged, compatibility-only)	**Yes**	**Yes**

Figure 2.7: Comparison between RVV, AVX-512, Arm SVE, NEC SX ISA [10]

2.3 Long Vector Architectures and the EPAC Accelerator

2.3.1 Motivation for long vector architectures

In the context of vector processors, the ideal scenario for SIMD parallelism would be to have as many vector units as the number of elements mappable in the vector registers, i.e., $VL_{MAX} =$ Num. Functional Units. In practice, for architectures with very long vector registers (*long vector architectures*) this is not feasible, as implementing hundreds of functional units is prohibitive in terms of chip area, power consumption, production cost.

Given this limitation, it is interesting to investigate the efficiency and behavior of long vector architectures where VL_{MAX} far exceeds the number of functional units ($VL_{MAX} \gg$ Num. Functional Units), and to explore the optimal design trade-off between VL_{MAX} and the number of functional units.

These analyses were conducted in [32], where a customizable

Field-Programmable Gate Array (FPGA)-based setup was used to simulate and study a configurable long vector architecture featuring 8 lanes and very long registers (up to 256 double-precision elements, totaling 16 kbits). The setup is configurable to simulate different levels of memory bandwidth (ranging from 1 to 64 Bytes per cycle) and memory latency (by adding an

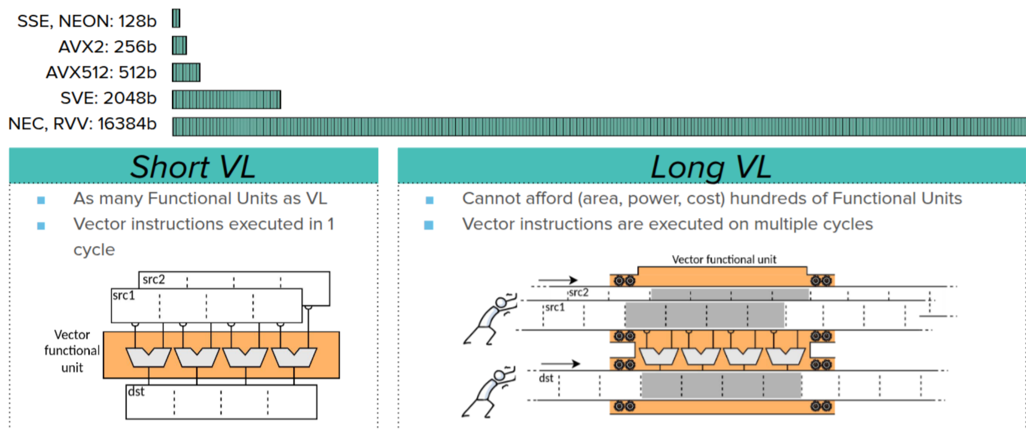
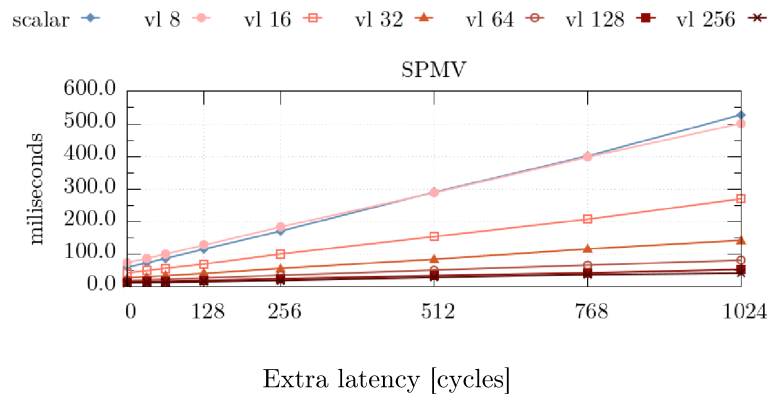


Figure 2.8: Short vectors vs Long vectors [6].

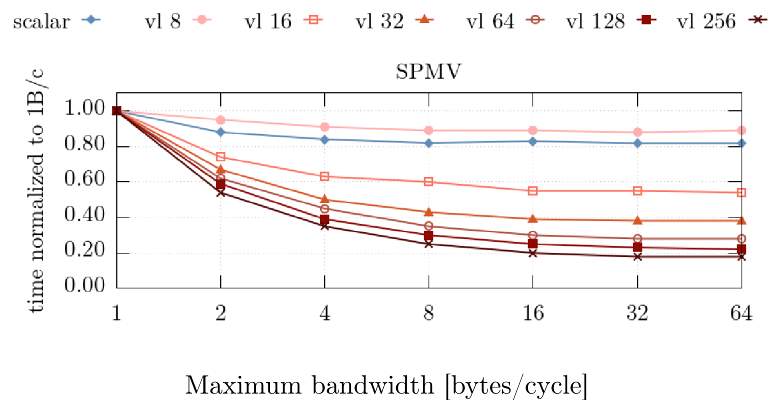
arbitrary number of extra cycles to main memory access). The experiments were conducted on four computational kernels relevant to HPC workloads in the sparse regime, including SpMV. These experiments were systematically repeated using various combinations of vector length, memory bandwidth, and memory access latency, evaluating whether using a high VL brings benefits despite the fixed number of lanes. The conclusions of this work highlighted two key characteristics of long vector architectures:

1. **Long vector architectures tolerate memory latency better than their short vector counterparts:** When latency is added to the memory subsystem, vector architectures are less penalized than scalar ones, a phenomenon that is even more pronounced in long vector architectures compared to short vector ones.

2. **Long vector architectures achieve higher bandwidth utilization:** It was observed that while single-core scalar implementations do not benefit from a bandwidth higher than 1 or 2 Bytes/Cycle, long vector implementations can naturally utilize bandwidths of 32 or 64 Bytes/Cycle.



(a) Scalar vs Vector execution with increasing VL depending on added latency. (low is better)



(b) SpMV kernel execution time normalized to 1 Byte/Cycle limit. (low is better)

Figure 2.9: Performance analysis of the SpMV kernel under various memory latency and bandwidth constraints [32].

The experimental setup, named FPGA-SDV, was specifically designed to emulate parts of the EPAC, combining a RISC-V scalar core with a large VPU.

Based on this rationale, the EPI decided to adopt a long-vector approach for the project's vector accelerator, EPAC, which will be analyzed in detail in the following section.

2.4 EPAC 1.5 Accelerator Architecture

The EPAC, developed within the scope of EPI project, represents the consortium's primary platform designed to address HPC workloads. Its latest iteration, EPAC 1.5, was successfully manufactured in 2023 using GlobalFoundries 22FDX low-power technology.

EPAC 1.5 adopts a modular, heterogeneous tile-based architecture. Unlike monolithic designs, the chip integrates distinct accelerators specialized for different computational domains. These tiles are interconnected via a high-speed Network-on-Chip (NoC) that allows them to share data efficiently.

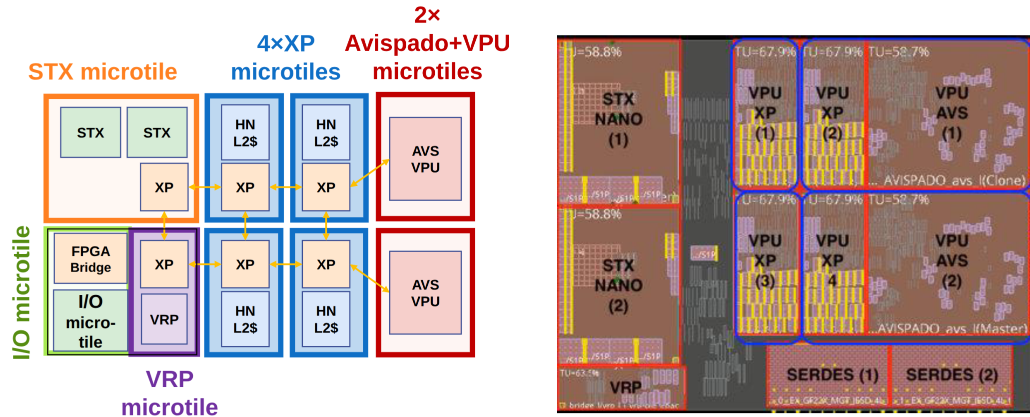


Figure 2.10: EPAC 1.5 accelerator tape-out with multi-tile structure. [20]

The architecture comprises three distinct processing units:

- **(STX) Many-core stencil/machine learning accelerator tile:** Designed by ETH Zurich and Fraunhofer IIS, this unit is highly specialized for machine learning workloads and stencil computations, providing extreme efficiency for specific dense kernels.

- **(VRP) Variable precision tile:** Designed by CEA, this tile features a general-purpose core extended to support variable precision arithmetic. It offers algorithmic flexibility for scientific applications where standard IEEE floating-point precision is either insufficient (requiring

higher accuracy) or overkill (allowing for reduced precision to save energy).

- **(VEC) Vector accelerator tile:** Designed by BSC and Semidynamics, this tile combines a standard RISC-V scalar core with a long vector unit VPU. It acts as the primary engine for general scientific computing, capable of processing large amounts of data in parallel using long vectors. This is the main focus of this thesis and it will be described further below.

2.4.1 The VEC Tile Architecture

In its latest iteration, EPAC 1.5 integrates two independent General Purpose VEC Tiles. Each tile operates as an autonomous unit, possessing its own scalar core and a dedicated VPU. Although, these two cores can be leveraged simultaneously by partitioning the application workload into two threads, utilizing parallel programming models such as MPI or OpenMP. This thesis focuses on the characterization and analysis of a single VEC tile execution (single-threaded).

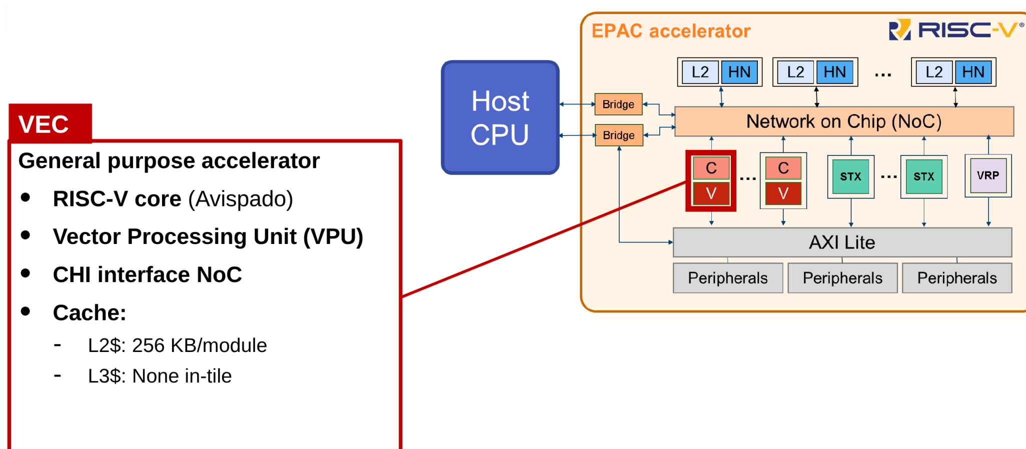


Figure 2.11: EPAC accelerator multi-tile structure with focus on the VEC tile architecture. [20].

Specifically, each VEC tile implements a heterogeneous coupling of a

scalar processor and a vector unit. The scalar component is the Avispado RISC-V core (by Semidynamics), a 2-way in-order core equipped with private L1 caches (16KB Instruction cache, 32KB Data cache).

The scalar core is tightly coupled with a private VPU, based on the Vitruvius+ microarchitecture [21]. This unit supports the RVV (RVV v0.7.1) and features a physical Vector Register File (VRF) containing 40 registers. Each vector register has a length of 256 double-precision elements (16 Kbits). The computational power is provided by 8 parallel lanes, each equipped with Fused Multiply-Add (FMA) units capable of executing 2 DP Floating-Point Operations (FLOPs)/cycle (addition + multiplication).

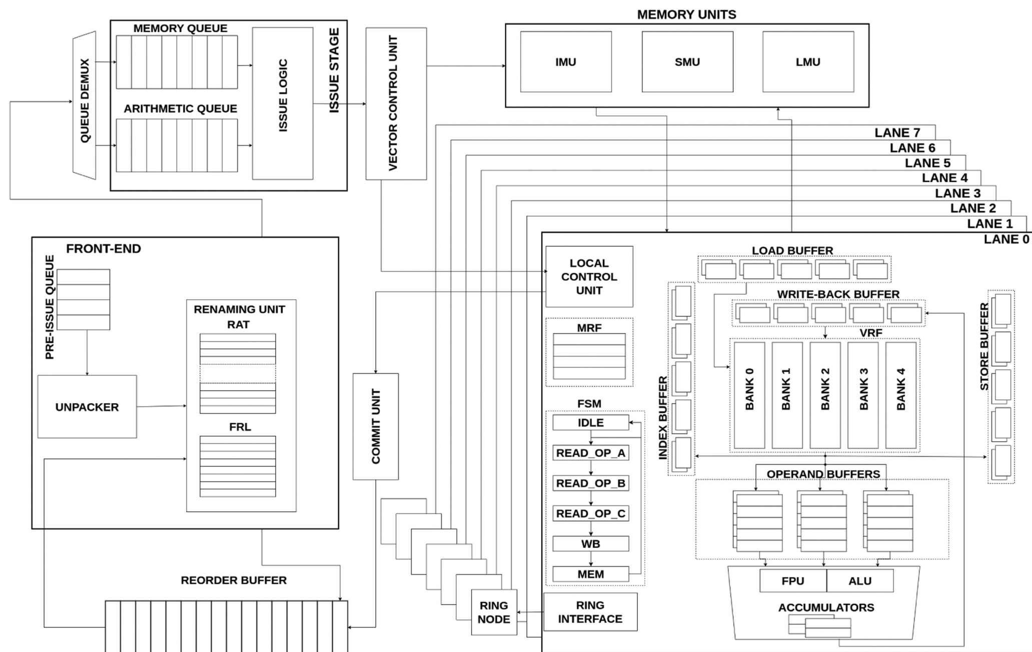


Figure 2.12: Vitruvius+ architecture [21]

A notable microarchitectural feature is the implementation of 40 physical registers versus the 32 architectural registers defined by the RISC-V standard. This over-provisioning enables internal register renaming: by dynamically mapping logical registers to different physical ones, the VPU can eliminate false dependencies (such as Write-After-Write hazards), facilitating Out-of-Order execution and maximizing pipeline utilization. The VEC

tile interfaces with the chip's global memory subsystem via the NoC. Data access is mediated by the distributed L2 cache slices shared across the EPAC architecture. This design allows the VPU to efficiently access large data structures resident in the shared L2 or main memory.

Microarchitecturally, the VPU implements a long vector design philosophy where the maximum vector length is significantly larger than the number of parallel lanes ($VL_{max} \gg N_{lanes}$), as explored in the previous section. This design is intended to operate as a coprocessor for the Avispado core, which acts as the main controller, delegating and offloading all vector instructions to the VPU.

Architecture	**Vector register size** (1 cell = 1 double element)									
Intel AVX512	D1	D2	D3	D4	D5	D6	D7	D8		
Arm Neon	D1	D2								
A64FX	D1	D2	D3	D4	D5	D6	D7	D8		
NEC Aurora SX	D1	D2	D3	D4	D5	D6	D7	D8	...	D256
EPAC VPU	D1	D2	D3	D4	D5	D6	D7	D8	...	D256

Figure 2.13: EPAC VPU VL_{max} versus other different architectures [20]

This architectural implementation follows the findings of preliminary studies conducted on FPGA-based emulations closely resembling the EPAC design [32]. These studies aim to demonstrate the superior efficiency of the long-vector paradigm for scientific computing workloads compared to traditional short-SIMD approaches. Consequently, the platform is currently undergoing extensive validation across a wide range of domain-specific codes. This thesis contributes directly to this effort by focusing on the porting and analysis of the OpenFOAM framework.

Chapter 3

Numerical Foundations of Computational Fluid Dynamics

3.1 Problem discretization

The physical behavior of fluids is governed by a set of Partial Differential Equations (PDEs) known as the Navier-Stokes equations. These equations represent the conservation of mass, momentum, and energy applied to a continuous fluid, modeling how fields such as velocity, pressure, temperature, and density evolve and interact over time.

To give an example, the mass conservation equation is expressed as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{3.1}$$

where ρ represents the mass density of the fluid and $\mathbf{u}$ is the velocity vector of the flow field [13].

Due to their non-linearity and strong coupling, these equations generally do not admit an analytical solution for real-world industrial applications. Therefore, it is necessary to approximate the continuous problem using a discrete formulation, which can be solved numerically on a computer. While various discretization methods exist (e.g., Finite Differences, Finite Elements), the fvm is the standard approach in CFD and the foundational method used by the OpenFOAM framework.

The FVM relies on decomposing the spatial domain into a finite number of contiguous sub-regions called *control volumes* (or cells), which collectively form the computational grid (*mesh*).

Instead of calculating PDEs at isolated grid points, the FVM divides the spatial domain into small contiguous blocks called control volumes. The PDEs are integrated over these volumes to evaluate the fluxes, representing the exact amount of fluid or energy crossing each face. Since any amount of flux leaving one cell must identically enter the adjacent sharing cell, the FVM is inherently conservative, ensuring no mass or energy is artificially created or lost. This mathematical formulation strictly respects a fundamental physical principle: local conservation. In a continuous fluid, changes within any given region depend exclusively on the fluxes crossing its boundaries.

The FVM translates this physical locality to the discrete mesh. By evaluating fluxes strictly across the shared faces of adjacent control volumes, the state of any cell becomes directly coupled only to its immediate neighbors. This ensures that information propagates physically from cell to cell, avoiding any artificial long-distance coupling in the resulting linear system.

More specifically, let us consider the 2D mesh shown in Figure 3.1, and assume we are discretizing the pressure field p:

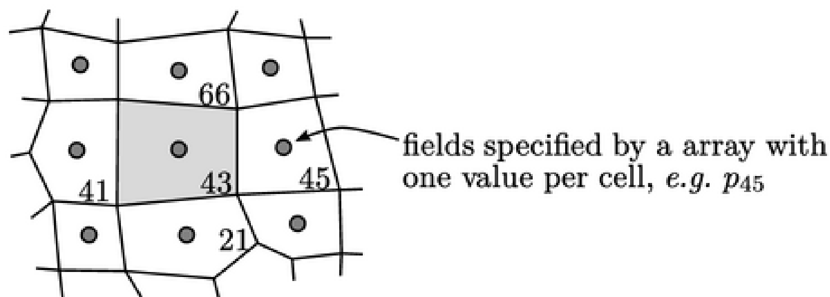


Figure 3.1: Example of an unstructured computational mesh for FVM discretization [13].

The discretization process transforms the integral equations into a set of linear algebraic equations. For a specific control volume (e.g., cell 43),

an equation is built to relate its pressure p_{43} to the pressures of its direct neighbors:

$$a_{43,21}p_{21} + a_{43,41}p_{41} + \mathbf{a_{43,43}p_{43}} + a_{43,45}p_{45} + a_{43,66}p_{66} = b_{43} \tag{3.2}$$

Here, the central coefficient $a_{43,43}$ multiplies the unknown of the cell itself, while the off-diagonal coefficients (like $a_{43,21}$) represent the physical linkage (fluxes) through the faces shared with neighboring cells.

By assembling the N algebraic equations for the N cells of the domain, a global linear system of the form $\mathbf{Ax} = \mathbf{b}$ is obtained:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,N} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & a_{N,3} & \cdots & a_{N,N} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_N \end{bmatrix} \tag{3.3}$$

The coefficient matrix A exhibits the following crucial properties:

1. It is a square matrix with dimensions $N \times N$, where N is the total number of control volumes.

2. Each row i of the matrix corresponds to the linear equation derived for cell i.

3. On row i, the only non-zero elements are the diagonal element $a_{i,i}$ (representing the cell itself) and the off-diagonal elements corresponding to its direct topological neighbors.

Since a typical cell in a 3D mesh is a polyhedron with a limited number of faces, the number of non-zero entries in any row is extremely small compared to N. This characteristic renders the matrix A highly *sparse*, meaning the vast majority of its elements are zero.

As we will see in the next section, this extreme sparsity completely alters the computational approach and the data structures required to solve the system efficiently, shifting the hardware requirements from dense arithmetic to memory bandwidth and indirect addressing.

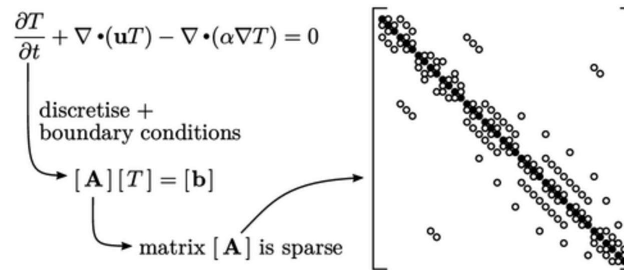


Figure 3.2: From transport equation to sparse linear system [13].

3.2 Iterative solvers

3.2.1 Direct vs iterative methods

As established in the previous section, the FVM discretization process yields a linear system of equations of the form $\mathbf{Ax} = \mathbf{b}$, where the coefficient matrix A is large but highly sparse. In practical CFD applications, the scale of these systems makes the choice of the algebraic solver critical for overall simulation performance.

Historically, linear systems were solved using *direct methods*, such as Gaussian elimination or LU factorization. While direct methods attempt to compute an exact solution in a predictable, finite number of operations, they are fundamentally ill-suited for the sparse matrices generated by 3D FVM meshes.

The primary reason is a phenomenon known as *fill-in*. During the execution of direct algorithms, row-wise operations introduce non-zero elements into positions of the matrix that were originally zero. Consequently, the originally sparse matrix transforms into a substantially dense one.

The memory footprint escalates from $\mathcal{O}(N)$ for the sparse representation to $\mathcal{O}(N^2)$ for the filled-in matrix, and the computational complexity scales as $\mathcal{O}(N^3)$.

Given the memory capacity constraints of modern hardware, storing and factoring massive dense matrices with millions of unknowns is computationally infeasible. To overcome this bottleneck, CFD codes rely almost exclu-

sively on a different class of algorithms known as *iterative solvers*.

Unlike direct methods, iterative solvers start with an initial guess x_0 and generate a sequence of approximate solutions x_k. At each iteration step k, the solver evaluates the error by computing the *residual* vector, defined mathematically as:

$$\mathbf{r_k} = \mathbf{b} - \mathbf{A}\mathbf{x_k} \tag{3.4}$$

The method iteratively updates the solution to minimize this residual. The execution terminates when the norm of the residual, $|r_k|$, falls below a user-defined tolerance level, indicating that the solution has sufficiently converged.

Crucially, the core computational workload in iterative algorithms consists of SpMV operations. By only reading the matrix A without ever modifying it, these operations guarantee the zero fill-in characteristic. This preserves the original $\mathcal{O}(N)$ memory footprint and shifts the computational challenge from dense arithmetic operations to memory bandwidth efficiency, making iterative solvers the only viable approach for large-scale CFD problems.

3.2.2 The Conjugate Gradient Method

The CG method [29] is a notable iterative solver specifically designed for linear systems where the coefficient matrix $\mathbf{A}$ is Symmetric and Positive-Definite (SPD). The fundamental principle of CG is that solving $\mathbf{Ax} = \mathbf{b}$ is mathematically equivalent to finding the global minimum of the quadratic form:

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - \mathbf{x}^\mathsf{T}\mathbf{b} \tag{3.5}$$

Since $\mathbf{A}$ is positive-definite, $\phi(\mathbf{x})$ represents a strictly convex paraboloid. Its unique minimum occurs where the gradient vanishes, leading directly to the original linear system: $\nabla\phi(\mathbf{x}) = \mathbf{Ax} - \mathbf{b} = \mathbf{0}$.

At any iteration k, the residual $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$ corresponds precisely to the negative gradient $(-\nabla\phi(\mathbf{x}_k))$, pointing in the direction of steepest descent. The solution is updated iteratively by taking a step of size α_k along a search

direction $\mathbf{p}_k$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{3.6}$$

Simply using the residual as the search direction ($\mathbf{p}_k = \mathbf{r}_k$) leads to the *Steepest Descent* method. This approach suffers from slow convergence because consecutive steps are strictly orthogonal, leading to inefficient *zig-zag* trajectories across the multidimensional surface 3.3.

To prevent this, the CG method generates a sequence of search directions that are not merely orthogonal, but *conjugate* (or **A**-orthogonal):

$$\mathbf{p}_i^{\mathsf{T}} \mathbf{A} \mathbf{p}_j = 0, \quad \text{for } i \neq j \tag{3.7}$$

Each step's direction is orthogonal to all the previous ones, unlike steepest descent which is just orthogonal to the previous one.

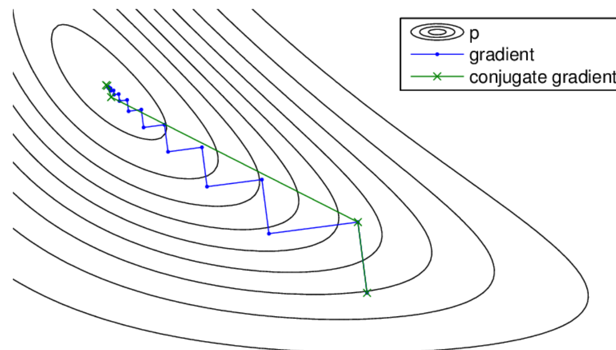


Figure 3.3: Steepest descent versus CG trajectory [15].

The step size α_k minimizing ϕ along $\mathbf{p}_k$, and the coefficient β_k ensuring conjugacy with the previous direction, are efficiently computed using inner products:

$$\alpha_k = \frac{\mathbf{r}_k^{\mathsf{T}} \mathbf{r}_k}{\mathbf{p}_k^{\mathsf{T}} \mathbf{A} \mathbf{p}_k}, \qquad \beta_k = \frac{\mathbf{r}_{k+1}^{\mathsf{T}} \mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathsf{T}} \mathbf{r}_k} \tag{3.8}$$

The new search direction is then updated as $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$. The complete execution flow is presented in Algorithm 1.

Algorithm 1 Conjugate Gradient Method

Require: $\mathbf{A}$ (SPD matrix), $\mathbf{b}$ (rhs), $\mathbf{x}_0$ (initial guess)
Ensure: $\mathbf{x}_{k+1}$ (approximate solution)

1: $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$ ▷ Initial residual
2: $\mathbf{p}_0 := \mathbf{r}_0$ ▷ Initial direction
3: $k := 0$
4: **repeat**
5: $\quad \alpha_k := \frac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$ ▷ Step size
6: $\quad \mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$ ▷ Update solution
7: $\quad \mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$ ▷ Update residual
8: $\quad$ **if** $|\mathbf{r}_{k+1}|$ is sufficiently small **then**
9: $\qquad$ **exit loop**
10: $\quad$ **end if**
11: $\quad \beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$ ▷ Beta coefficient
12: $\quad \mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$ ▷ New search direction
13: $\quad k := k + 1$
14: **until** convergence

In exact arithmetic, the conjugate directions guarantee that the method converges in exactly N steps. However, in CFD applications, CG is strictly employed as an iterative solver to reach an acceptable error tolerance much earlier. The rate of convergence is governed by the spectral condition number of the matrix:

$$\kappa(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})} \tag{3.9}$$

A matrix with a lower condition number is considered better conditioned, resulting in a faster rate of convergence and requiring significantly fewer iterations to reach the desired error tolerance.

3.3 Multigrid method

In numerical analysis, the spatial error across the mesh can be categorized by its frequency. Standard iterative solvers, like the CG, act as "smoothers": they quickly eliminate *high-frequency* errors (localized oscillations between adjacent cells). However, they struggle with *low-frequency* errors (smooth, global deviations). Since each SpMV iteration only exchanges information between immediate neighbors, propagating and resolving these global errors takes a massive number of iterations.

The MG method overcomes this by using a hierarchy of coarser grids. A smooth, low-frequency error on a fine mesh spans fewer cells on a coarser mesh, effectively becoming a high-frequency error. This allows standard solvers to quickly eliminate it at the coarser level.

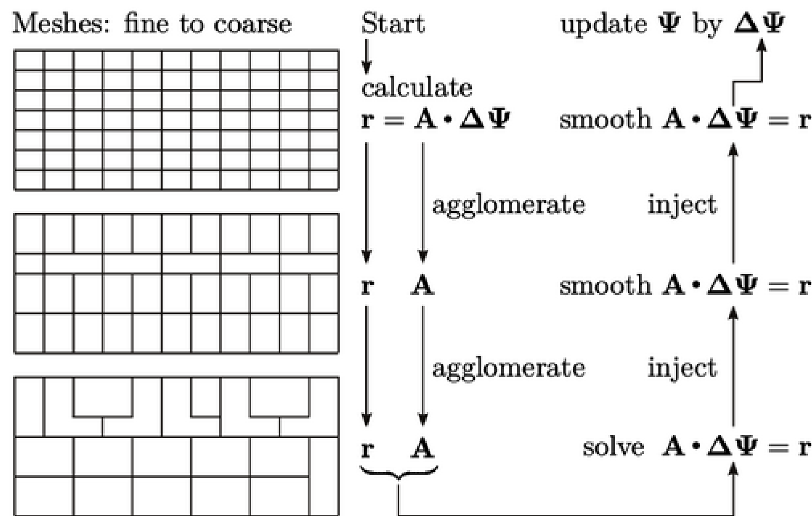


Figure 3.4: MG method [13].

Instead of solving the original problem on coarse grids, MG typically uses the *Correction Scheme*. It solves a residual equation to find a correction for the fine grid. If the fine-grid approximation has an error e_h and a residual

r_h, they are linked by the simple linear relationship:

$$A_h e_h = r_h \tag{3.10}$$

Solving this on a coarse grid provides the necessary correction e_h to update and improve the fine grid solution.

Moving between grid levels relies on two fundamental operations:

- **Restriction:** Transfers the residual from a fine grid down to a coarser one.
- **Prolongation:** Interpolates the calculated correction from the coarse grid back up to the fine grid.

A standard MG cycle (also called V-cycle) follows these straightforward steps:

1. **Smoothing:** Apply a few iterations of a basic solver (e.g., Gauss-Seidel, Richardson) to reduce high-frequency errors on the fine grid.
2. **Restriction:** Compute the residual and project it onto the coarse grid.
3. **Coarse Grid Solve:** Solve the residual equation on the coarse grid to find the correction. On the very coarsest level, an exact direct solver is typically used.
4. **Prolongation:** Interpolate the correction back to the fine grid.
5. **Correction & Post-Smoothing:** Add the correction to the fine grid solution. Then, apply a few more smoothing iterations to wipe out any new high-frequency errors introduced by the interpolation step.

In CFD, solvers often rely on a hybrid approach called gamg. Instead of generating completely new physical meshes for the coarse levels (Geometric MG) or relying strictly on pure matrix mathematics (Algebraic MG), GAMG simplifies the coarsening process.

It agglomerates (groups) adjacent fine-grid cells and directly sums their matrix coefficients to build the coarse systems. This avoids complex geometrical re-meshing and makes GAMG highly efficient and robust for solving pressure equations on complex, unstructured meshes.

3.4 Introduction to OpenFOAM framework

OpenFOAM is a highly modular, open-source C++ framework dedicated to CFD. It is fundamentally based on the FVM and uses an object-oriented design to handle complex tensor mathematics and fluid dynamics models.

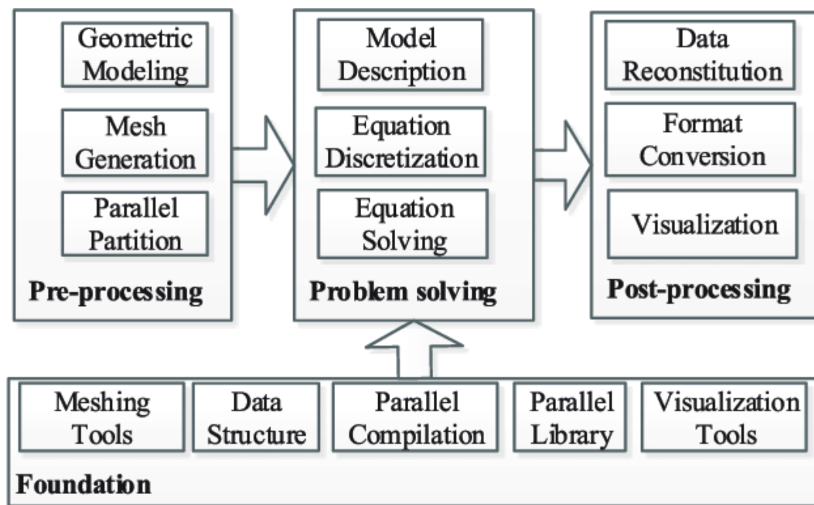


Figure 3.5: OpenFOAM framework structure [19].

A unique feature of OpenFOAM is its intuitive top-level syntax. The C++ code is designed to closely mimic the mathematical notation of the partial differential equations being solved. For instance, the Navier-Stokes momentum equation can be written directly in the solver as:

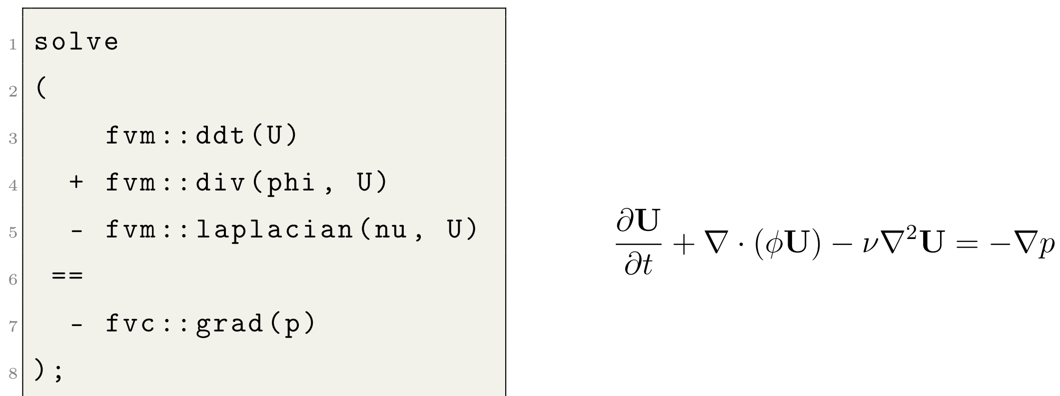


Figure 3.6: Comparison between OpenFOAM syntax (left) and the continuous Navier-Stokes equation (right).

To handle complex geometries, OpenFOAM generates large sparse linear systems. It tackles these computationally heavy problems primarily through domain decomposition, distributing sub-domains across multiple MPI processes.

While this coarse-grained approach scales effectively on distributed memory systems, OpenFOAM struggles to exploit fine-grained, instruction-level parallelism, such as SIMD/vector execution. This bottleneck is fundamentally tied to its internal sparse matrix storage formats, which are not designed for vectorization.

To address this limitation, the following sections present an extension of OpenFOAM's capabilities to support modern vector architectures. By transitioning to a vector-friendly sparse matrix representation, this work demonstrates how the core SpMV kernel can be efficiently accelerated.

Chapter 4

Sparse matrix formats and SpMV vectorization

Although there is no objective definition for sparse matrices, they are generally understood as matrices containing a predominant number of zero elements, typically the number of non-zeros is on the order of the number of rows/columns or fewer. These types of matrices are very common in scientific computing; specifically, as shown in Chapter 3, the discretization of CFD problems inherently involves sparse matrices. When operating on such matrices, it is highly discouraged to store them in a dense format (saving all the zeros in memory), both to avoid wasting storage space and to prevent unnecessary computations on zeros that do not contribute to the final results of the operations.

In iterative solvers like the CG method, the SpMV often dominates the computation time. Therefore, it is fundamental to analyze its computational profile and arithmetic intensity.

Regardless of the underlying data structure, the SpMV operation performs very few floating point operations for each non-zero element.

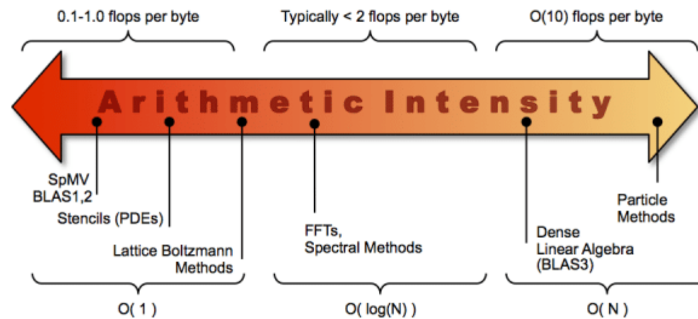


Figure 4.1: Arithmetic Intensity of main HPC kernels [4].

For this reason, SpMV is a memory-bound kernel, meaning that Its performance is primarily dictated by the memory bandwidth and latency rather than the peak computational power of the Central Processing Unit (CPU) or the vector units. Understanding this memory bottleneck is crucial because it directly motivates the need for specialized sparse matrix storage formats. The formats introduced in the following sections are specifically designed to minimize memory traffic, improve data locality, and exploit modern vector architectures.

4.1 Basic sparse matrix formats

The core idea behind sparse matrix formats is to store only the non-zero elements in memory. Generally, in both basic and advanced formats, matrices are represented using multiple arrays: one for the non-zero values, and others for their corresponding indices. We begin our analysis with the simplest and most intuitive formats, which are also among the most widely used.

4.1.1 COO

The COO is the most flexible sparse representation, storing non-zero elements using three arrays: values, row indices, and column indices. COO does not require any specific ordering of elements, making it ideal for matrix construction and data exchange.

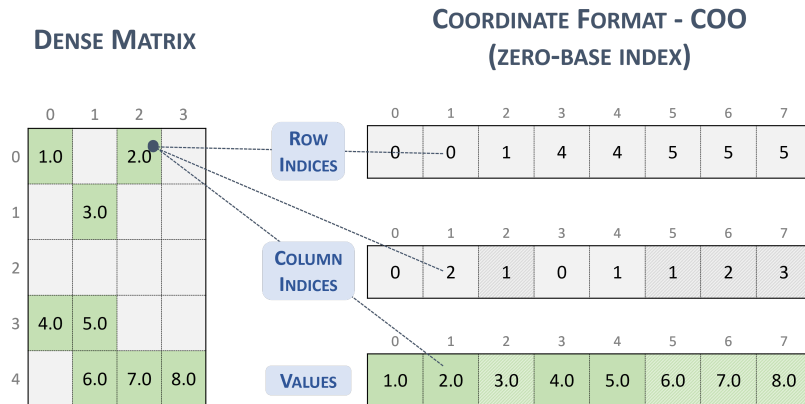


Figure 4.2: COO sparse matrix format [23].

- **Pros:** Straightforward implementation. New entries can be appended without reordering existing data.

- **Cons:** High memory footprint due to redundant index storage. Its main drawback in HPC is its inherent structural limitations for efficient vectorization. Because COO lacks a strict ordering, elements belonging to the same row can be processed simultaneously by different SIMD lanes. In a vectorized SpMV, this triggers *write-after-write* conflicts (race conditions) when multiple lanes attempt to accumulate partial sums into the same memory location of the output vector. Resolving these conflicts requires atomic operations or software serialization, which severely degrades parallel performance. [12]

4.1.2 CSR

The CSR format optimizes storage by compressing row indices into a `row_offsets` array of size $N + 1$. This pointer-based structure eliminates the redundancy of COO, ensuring that all non-zero elements of a row are stored contiguously in memory.

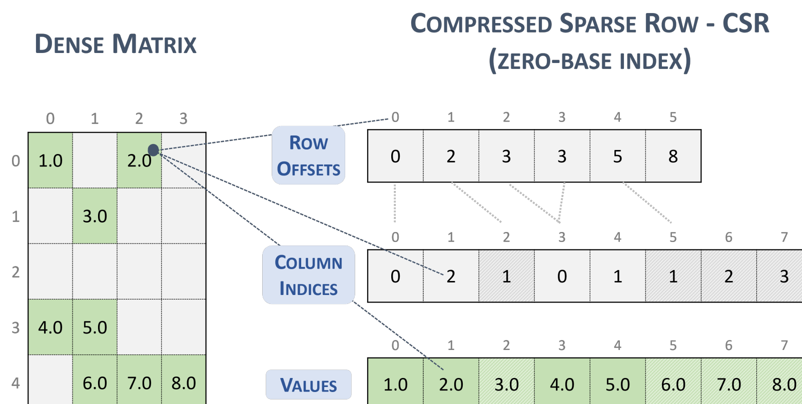


Figure 4.3: CSR sparse matrix format [23].

- **Pros:** Efficient memory footprint compared to COO, as it eliminates the redundant row index storage. Regarding parallelization, CSR enables conflict-free writes during the SpMV operation. By parallelizing across the rows (assigning distinct rows to different threads or processing units), each element of the output vector y is computed independently. This implies that partial sums are accumulated into unique, row-specific memory locations, naturally avoiding race conditions without the need for atomic operations.

- **Cons:** While highly efficient for scalar execution, CSR poses significant challenges for vector architectures, particularly when employing a "vertical vectorization" strategy (where each SIMD lane processes a different row). The core issue is the variable row length, which causes severe load imbalance. When a batch of rows is mapped to a vector unit, the execution time is dictated by the longest row in that batch. Lanes assigned to shorter rows must be masked off once they finish their work, remaining idle and wasting computational cycles while waiting for the longest row to complete [18].

4.2 Vector friendly sparse matrix formats

As we saw earlier, the variable row lengths in CSR make it very inefficient for vector processors. To solve this problem, we need data structures specifically designed to keep vector units busy and avoid idle time.

4.2.1 ELL

The ELL format, often just called ELL, solves the load imbalance problem by forcing all rows to have the same length. Structurally, it represents the sparse matrix using two dense, rectangular arrays of dimensions $N \times K$: one storing the non-zero values and the other storing their corresponding column indices. Here, K corresponds to the maximum number of non-zeros found in any single row. Rows with fewer than K elements are padded with explicit zeros (in the value array) and dummy indices (in the column index array) to match this uniform width.

This transformation creates a regular grid structure. To maximize memory efficiency during vector operations, these arrays are usually stored in column-major order and processed column-wise. This memory layout allows the processor to load a continuous block of elements from a single column directly into a vector register, effectively processing multiple rows simultaneously in a SIMD fashion.

- **Pros:** Excellent suitability for vectorization. The fixed row length effectively eliminates load imbalance, allowing SIMD lanes to execute in lockstep without idle time. Furthermore, the column-major memory layout ensures contiguous memory access during the SpMV iteration, which is crucial for maximizing memory bandwidth utilization.

- **Cons:** Significant memory overhead for irregular matrices. The format requires padding all rows to the length of the single longest row in the matrix. For unstructured CFD meshes, which often exhibit variable row lengths, this results in the storage and fetching of a vast

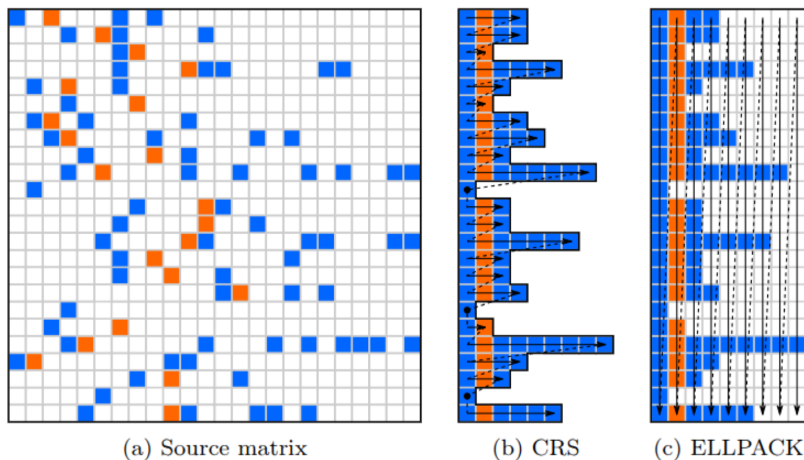


Figure 4.4: CSR and ELL sparse matrix formats. Arrows indicate the order of elements in memory.[18].

amount of explicit zeros. Consequently, valuable memory bandwidth is consumed by transferring non-informative data, potentially degrading performance compared to compact formats like CSR.

4.2.2 SELL-C-σ

The SELL-C-σ format is a variant of ELL that addresses the heavy padding overhead of the standard ELL format while keeping vector units fully utilized. It introduces two main modifications:

1. First, the matrix is divided into equally-sized *chunks* (or *slices*) of C rows. Instead of padding all rows to the global maximum length of the matrix, rows are zero-padded only to match the length of the longest row within their specific chunk. Within each padded chunk, the elements are stored consecutively in column-major order, and the chunks themselves are stored consecutively in memory. The chunk size C is typically chosen in accordance with the relevant SIMD width or register size of the target hardware architecture.

2. Second, it introduces a *sorting scope* parameter, σ. To further reduce

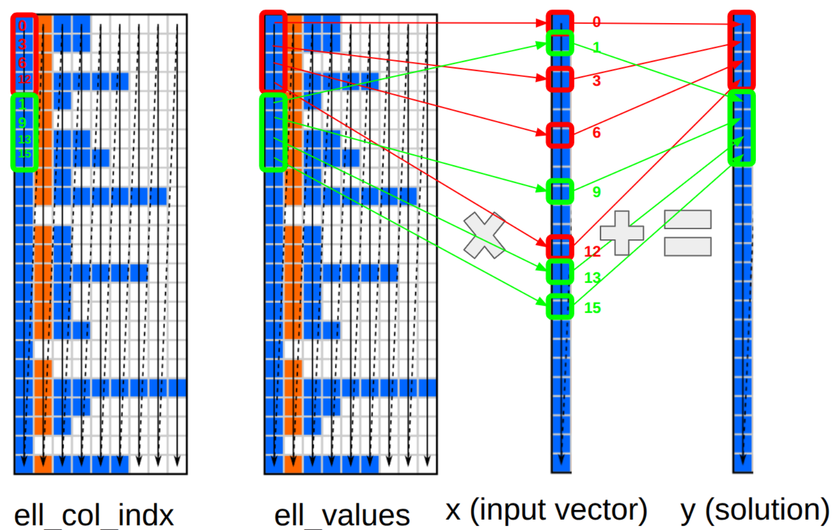
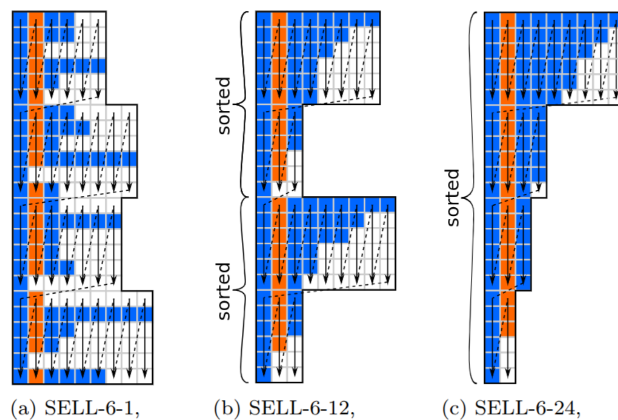


Figure 4.5: SpMV operation of ELL format [18].

padding overhead and improve *chunk occupancy*, rows are sorted by their length (Number of Non-Zeros (NNZ)) in descending order so that rows of similar lengths end up close to each other. However, instead of sorting the matrix globally (which would not only destroy spatial or temporal locality for the Right-Hand Side (RHS) vector access but also significantly increase the format conversion overhead), sorting is restricted to a local window of σ consecutive rows.



(a) SELL-6-1, (b) SELL-6-12, (c) SELL-6-24,

Figure 4.6: SELL-C-σ sparse matrix format [18].

- **Pros:** It combines the vectorization efficiency of ELL with a reduced memory footprint. The inner loops do not require horizontal reduction operations or scalar remainder loops since chunk sizes are padded to multiples of C, leading to highly streamlined SIMD execution. Limiting the sorting scope σ balances low padding overhead with good data locality for the RHS vector.

- **Cons:** Setting up the format requires a preprocessing step to sort the matrix rows. Because rows are reordered, the column indices must also be permuted, which means iterative solvers need to switch between permuted *input* and *output* vectors after each iteration. Additionally, the sorting scope value is critical for cache performance of the RHS vector.

Vectorization of SELL-C-σ SpMV in RVV 1.0 intrinsics

```
1  void mv_sellcsigma_symmetric_permuted_rvv1_0_m4(
2      int n, int C, int nslices, int *slice_ptr,
3      int *slice_width, double *diag_perm,
4      double *sell_values, uint64_t *sell_cols,
5      double *x_perm, double *y_perm) {
6      //... Diagonal contributes
7      //...
8      for (int s = 0; s < nslices; ++s) { // Off-Diagonal contributes
9           int r0 = s * C;
10          int rows_in_slice = (r0 + C > n) ? (n - r0) : C;
11          int w = slice_width[s];
12          int base_ptr = slice_ptr[s];
13         for (int slot = 0; slot < w; ++slot) {
14             int slot_base_ptr = base_ptr + slot * C;
15             for (size_t rl = 0; rl < rows_in_slice; rl += vl) {
16                 vl = __riscv_vsetvl_e64m4(rows_in_slice - rl);
17                 size_t current_pos = (size_t)slot_base_ptr + rl;
18                 size_t current_row = (size_t)r0 + rl;
19                 vfloat64m4_t v_vals = __riscv_vle64_v_f64m4(
20                     &sell_values[current_pos], vl);
21                 vuint64m4_t v_col_idx = __riscv_vle64_v_u64m4(
22                     &sell_cols[current_pos], vl);
23                 vbool16_t mask = __riscv_vmsne_vx_u64m4_b16(
24                     v_col_idx, (uint64_t)-1, vl);
25                 vuint64m4_t v_byte_offsets = __riscv_vmul_vx_u64m4(
```

```
26                         v_col_idx, 8, vl);
27                     vfloat64m4_t v_xp_gathered = __riscv_vluxei64_v_f64m4_m(
28                         mask, x_perm, v_byte_offsets, vl);
29                     vfloat64m4_t v_y = __riscv_vle64_v_f64m4(
30                     &y_perm[current_row], vl);
31                     v_y = __riscv_vfmacc_vv_f64m4_m(
32                         mask, v_y, v_vals, v_xp_gathered, vl);
33                     __riscv_vse64_v_f64m4(&y_perm[current_row], v_y, vl);
34                 }
35             }
36         }
37 }
```

Listing 4.1: RISC-V Vectorized SpMV using symmetric SELL-C-σ format

The algorithm processes the matrix in slices of C rows. Within each slice, data is accessed in column-major order. This layout ensures that elements in the same local column reside in contiguous memory locations, allowing the RISC-V vector units to process multiple rows simultaneously without scattered memory accesses. The vector length (`vl`) is dynamically adjusted to handle any remaining elements of the slice.

The key RVV 1.0 operations in the inner loop are:

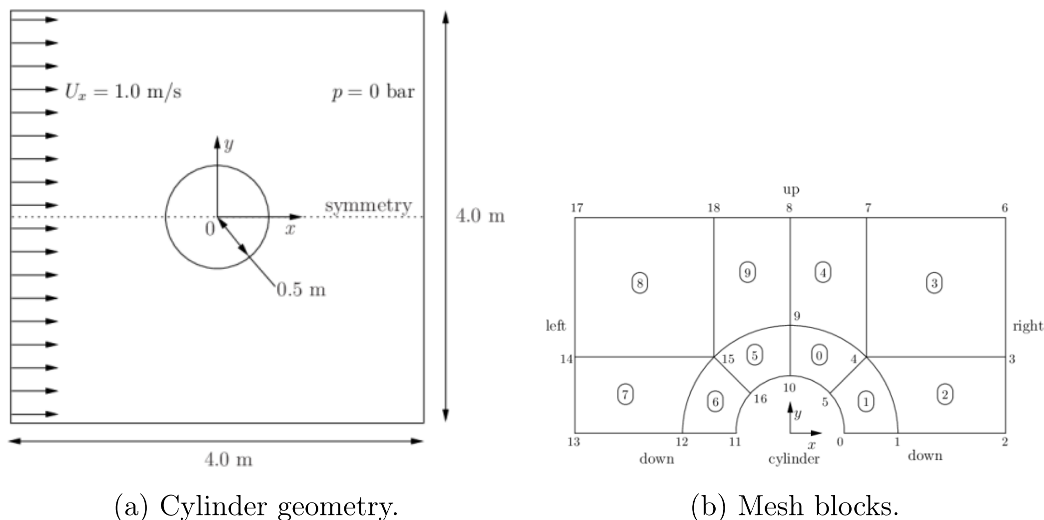
- **Contiguous loads:** The `__riscv_vle64_v_f64m4` and `__riscv_vle64_v_u64m4` intrinsics perform sequential memory reads to fetch the non-zero values (`v_vals`) and their corresponding column indices (`v_col_idx`).
- **Masked gather:** Elements of the input vector x must be fetched indirectly. After scaling the indices by 8 to compute byte offsets, a masked gather (`__riscv_vluxei64_v_f64m4_m`) loads the data. The mask prevents illegal memory accesses by disabling the operation for artificially padded elements (typically marked with an index of -1).
- **Masked FMA:** The core arithmetic uses `__riscv_vfmacc_vv_f64m4_m` (masked FMA) to multiply `v_vals` by the gathered x elements, accumulating the result directly into the partial y sums.

This combination of sequential loads, masked gathers, and vectorized

FMAs keeps the compute pipelines saturated, effectively resolving the load imbalance typical of unstructured matrices.

4.3 SpMV Evaluation: Long-Vector vs. Short-Vector Architectures

We evaluated the vectorized SpMV kernels by running microbenchmarks to analyze the speedup over the scalar baseline as a function of the problem size. The chosen scalar baseline relies on the symmetric COO format, which was selected to closely simulate the native memory layout and execution behavior of OpenFOAM during SpMV operations. The matrices were extracted directly from an OpenFOAM case study modeling a fluid flow impacting a solid cylinder4.7a. To properly assess scaling, the mesh resolution of the problem was systematically increased, generating matrices that range from 128k up to 4096k cells.



(a) Cylinder geometry. (b) Mesh blocks.

Figure 4.7: Overview of the CFD test case setup, showing the solid cylinder geometry (a) and the corresponding mesh resolution blocks (b).[24]

As depicted in Figure 4.8, the matrices extracted from this specific CFD test case exhibit a highly regular distribution of NNZ per row. Based on

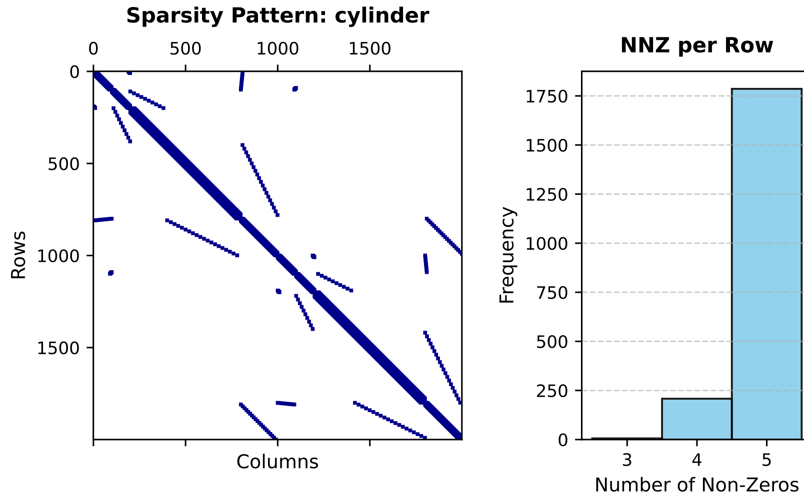


Figure 4.8: Non-zero distribution on the *cylinder_2k* matrix.

this observation, we configured the SELL-C-σ format with a sorting parameter $\sigma = 1$, meaning that no row sorting is performed. Since the variance in the NNZs per row is extremely low, sorting the rows by length would yield negligible benefits in terms of zero-padding reduction, and it is unlikely to compensate for the additional computational time required for the matrix conversion phase. For the chunk size parameter, we selected $C = 256$, which directly matches the VLEN of the EPAC architecture. Interestingly, although the SG2044 processor features a short-vector design with 128-bit vector registers, empirical tuning revealed that $C = 256$ remains a highly effective chunk size for this architecture as well, amortizing loop overheads and improving cache locality.

Our study primarily focused on understanding the performance speedup provided by the vectorized ELL and SELL-C-σ kernels relative to the scalar COO baseline across these growing problem sizes. In addition to performance scaling, we conducted microarchitectural profiling to gain deeper hardware insights. We utilized the `perf` tool on the SG2044 system and `sdv_trace` on the EPAC architecture. On the SG2044, where hardware cache counters are readily available, we tracked miss rates for both Level 1 (L1) and Last

Level Cache (LLC). Conversely, on EPAC, where such cache metrics are not directly accessible, we profiled the percentage of clock cycles during which the VPU stalled, an event that predominantly occurs when the processor is waiting for data to be fetched from memory.

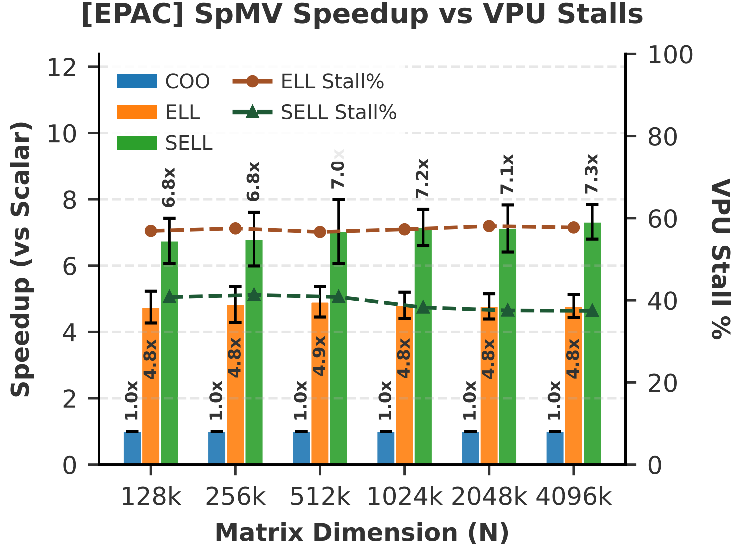


Figure 4.9: SpMV speedup and percentage of VPU stalls on EPAC.

The overall results demonstrate an average SpMV speedup of approximately 7× on the EPAC architecture using the SELL-C-σ format compared to the symmetric scalar COO baseline 4.9. In contrast, the speedup reaches only around 2× on the SG2044. Furthermore, the SELL-C-σ format (configured with parameters $C = 256$ and $\sigma = 1$) proved to be consistently more memory-efficient than its standard ELL counterpart. This memory efficiency is quantitatively confirmed by the reduced percentage of VPU stalls on EPAC and the lower percentage of L1 and LLC cache misses on the SG2044.

Consistent with our previous observations for dense Basic Linear Algebra Subprograms (BLAS) kernels [31], exploiting vector register grouping (LMUL=4) proved highly effective in boosting both the absolute performance and the relative speedup of SpMV on the SG2044. This specific optimization, however, is precluded on EPAC due to its inherent microarchitectural design choices: because EPAC natively features extremely long physical vec-

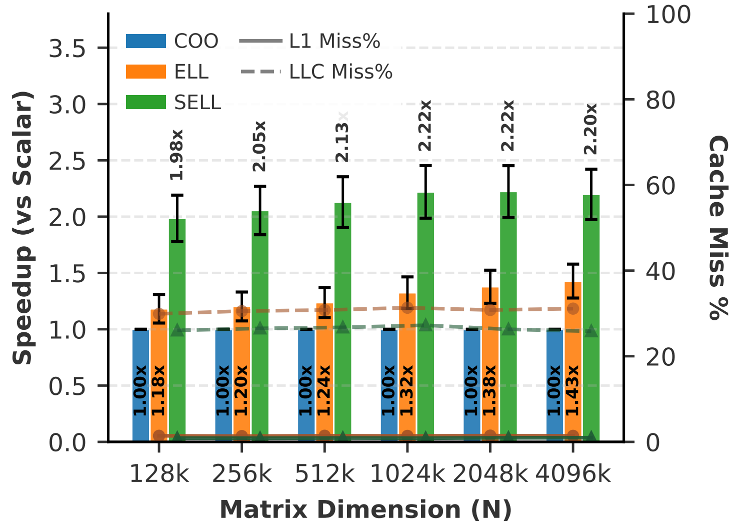


Figure 4.10: SpMV speedup and cache miss rates on SG2044.

tor registers, hardware-level register grouping is not implemented, as a single register already provides more than enough elements to effectively amortize instruction overhead and hide memory latency.

Ultimately, this evaluation highlights how the EPAC long-vector architecture performs significantly better than the SG2044 on memory-bound workloads. As extensively discussed in the literature [32], long-vector instructions are inherently superior at amortizing memory latency. By issuing a single instruction that requests hundreds of elements, the long-vector processor can effectively hide the memory access latency and sustain a much higher utilization of the available memory bandwidth compared to short-vector designs, which inevitably struggle with the frequent and fragmented memory requests typical of sparse matrix operations.

Chapter 5

End-to-End Acceleration of CFD Simulations: Smoother Integration and Evaluation

In the previous chapters, we demonstrated the efficacy of vectorizing the SpMV kernel using vector-friendly formats on RISC-V architectures. In this chapter, we will discuss its integration into the OpenFOAM framework, specifically within the GAMG solver.

Currently, OpenFOAM's parallelism relies almost exclusively on domain decomposition, where the computational mesh is partitioned and distributed across MPI processes. At the core level, explicit SIMD/vector parallelism is virtually unsupported. The standard distribution relies entirely on compiler auto-vectorization, which correctly handles dense arrays but fails to vectorize the complex, indirect memory accesses inherent in sparse matrix computations, the actual bottleneck of CFD solvers. Furthermore, while frameworks like SPUMA[7] have pioneered OpenFOAM's transition to GPU architectures, a comparable effort to explicitly exploit modern vector architectures is currently missing.

The root cause of this limitation lies in OpenFOAM's native matrix storage format, known as Lower-Diagonal-Upper (LDU). The LDU format is

conceptually similar to the COO format: it stores the main diagonal coefficients in a separate dense array, while the off-diagonal elements are stored in contiguous arrays dictated by explicit face addressing lists (`lowerAddr` and `upperAddr`). While memory-efficient and well-suited for sequential, scalar execution, LDU is fundamentally hostile to vector processors. It necessitates indirect, non-contiguous memory accesses (scatter/gather operations) that prevent efficient contiguous vector loads/stores, ultimately failing to saturate the memory bandwidth and stalling the VPU.

The core idea of this project is to bridge this architectural gap without modifying large portions of OpenFOAM's codebase. Our strategy involves four key steps:

1. Identifying a computationally intensive portion of the solver that is highly amenable to vectorization.
2. Intercepting the standard execution flow to dynamically convert the internal LDU matrices into vector-friendly layouts (such as ELL or SELL-C-σ) at runtime.
3. Executing the hardware-specific vectorized kernels directly via RVV intrinsics.
4. Integrating this entire workflow transparently as a dynamically loaded external plugin.

The following section analyzes a GAMG-based CFD simulation to isolate the most promising kernels for integrating our vectorized matrix-vector multiplication.

5.1 Simulation set-up and profiling of scalar execution

To evaluate the performance of our proposed vectorization strategy within a realistic computational framework, we selected a classic CFD benchmark:

the laminar flow past a solid cylinder[25]. The temporal evolution of the fluid dynamics was simulated using `icoFoam`, a standard OpenFOAM application. Specifically, `icoFoam` is a transient solver designed for incompressible, laminar flow of Newtonian fluids, utilizing the PISO (Pressure Implicit with Splitting of Operator) algorithm to resolve the pressure-velocity coupling. Within this algorithmic structure, the computationally intensive pressure equation is resolved using the GAMG linear solver.

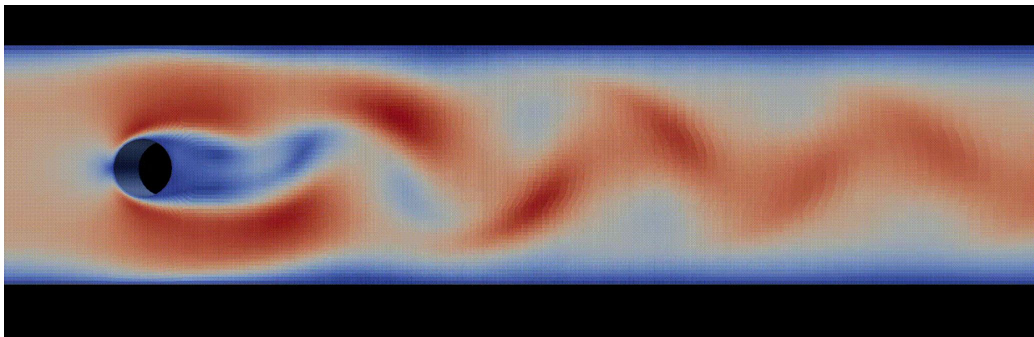


Figure 5.1: ParaView[3] visualization of the velocity magnitude ($|U|$) for the flow past a cylinder test case in OpenFOAM.

To identify the main computational bottlenecks and pinpoint the best candidate for hardware acceleration, we conducted an extensive performance profiling of the scalar execution. By leveraging the Paraver performance analysis tool, we traced and visualized the execution timeline of the complete simulation. This fine-grained temporal breakdown allowed us to isolate the execution times of the solver's individual algorithmic components and understand the dynamic behavior of the application.

The execution trace analysis revealed that the GAMG smoother is the most time-consuming phase of the entire simulation, accounting for nearly 50% of the total execution time. Specifically, the Richardson smoother was chosen as the primary target for our acceleration efforts, as it represents the most promising and easily vectorizable component within the GAMG pipeline. In the following section, we will analyze the breakdown of this smoother to study its fundamental computational building blocks in detail.

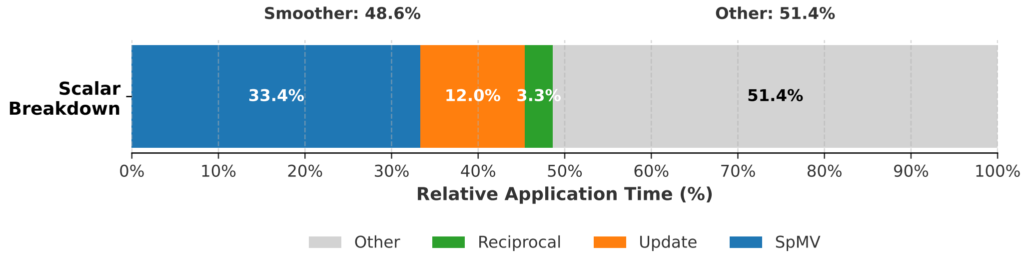


Figure 5.2: EPAC scalar application profiling

5.1.1 Anatomy of the Richardson smoother

The smoother acts as an iterative relaxation method applied to the linear system $A\psi = b$, typically to damp high-frequency errors in the multi-grid hierarchy.

Analyzing the native `Foam::richardsonSmoother::smooth` the algorithm can be logically partitioned into three distinct computational phases:

1. **Pre-computation (Diagonal Inversion):** At the beginning of each smoother call, the reciprocal of the matrix's main diagonal is computed once (D^{-1}).

2. **SpMV:** The core of the smoothing operation is an iterative loop executed for a defined number of `nSweeps`. The first step of each sweep is the evaluation of the full matrix-vector product $A \cdot \psi^{(k)}$ via the `matrix_.Amul()` method. In OpenFOAM's standard scalar format, this operation computes the main diagonal contributions via an element-wise array multiplication, and subsequently traverses the LDU addressing lists (upper and lower triangular arrays) to accumulate the off-diagonal contributions.

3. **Dense Vector Update:** Following the SpMV, the solution vector is updated using a damped Jacobi-like formulation. For each cell, the new value is computed as:

$$\psi_i^{(k+1)} = \psi_i^{(k)} + \omega \cdot D_i^{-1} \left(b_i - (A \cdot \psi^{(k)})_i \right) \tag{5.1}$$

where ω is a pre-calculated relaxation factor. In code, this translates to a loop over dense arrays executing fused multiply-add-like operations.

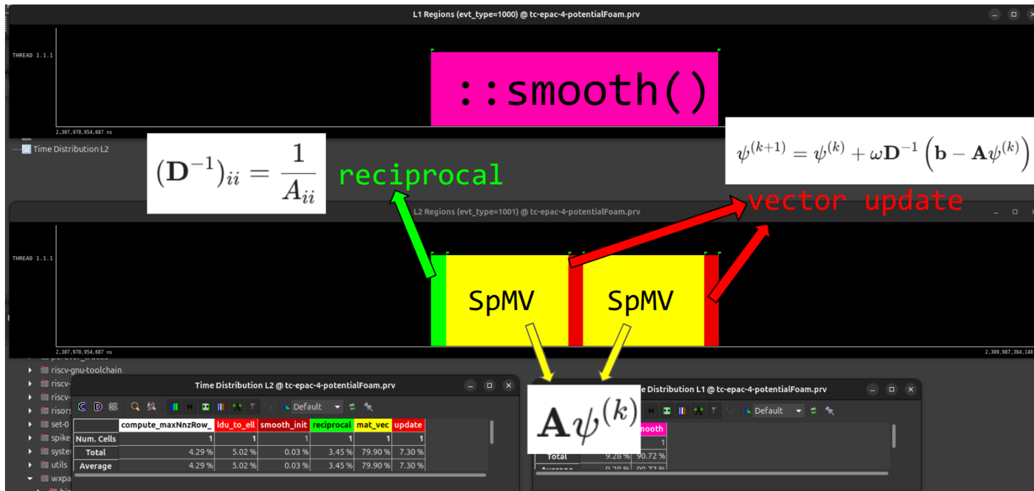


Figure 5.3: Execution trace from Paraver[5], a performance analysis and visualization tool, highlighting the execution time breakdown of the smoother's internal sections: SpMV, vector updates, and initialization.

From a computational profiling standpoint, the SpMV operation dominates the smoother, consuming approximately two-thirds of the total execution time. The remaining time is primarily distributed between the dense vector update and the initial diagonal inversion.

In the next section, we will analyze how to integrate a runtime matrix format conversion, transitioning from OpenFOAM's native LDU to a vector-friendly format, to enable a high-performance, vectorized SpMV kernel. Conversely, the diagonal inversion and the solution update operate exclusively on contiguous dense arrays, so they do not require conversions or modifications.

5.2 Runtime format conversion and plugin integration

To accelerate the GAMG smoother without altering OpenFOAM's source code, we packaged our vectorized kernel as an external, dynamically linked plugin. By leveraging OpenFOAM's modularity, we can load this new feature at startup via a configuration file, transparently redirecting the standard solver calls to our custom implementation.

Figure 5.4: Paraver execution trace showing the time distribution of the GAMG solver phases. The setup phase overhead is safely amortized over the repeated smoother executions.

The efficiency of this strategy relies entirely on separating the expensive matrix translation from the repetitive mathematical operations. As shown in the execution trace (Figure 5.4), the GAMG solver operates in two distinct phases:

1. **Setup Phase (Grid Hierarchy Construction):** Before solving the equations, GAMG builds a hierarchy of progressively coarser grids, initializing a new smoother C++ object for each level. The actual conversion from LDU to ELL/SELL-C-σ is performed in this phase,

specifically inside the *constructor* of our custom smoother class. This ensures the data translation happens strictly once per grid level.

2. **Solve Phase (Iterative Cycles):** Once the grids and converted matrices are prepared in memory, the solver iterates up and down the hierarchy. During this phase, the `::smooth()` method of the already instantiated object is called repeatedly, alongside other standard GAMG operations (e.g., restriction and prolongation). Crucially, the smoother's constructor is *never* re-invoked during these iterations, completely isolating the matrix conversion overhead from the actual solving cycles.

By isolating the format conversion within the class constructor, the initial time penalty is heavily amortized. The more iterations the solver requires to converge, the more times the converted matrices are reused, maximizing the speedup.

5.3 Final profiling and application speedup

In this final section, we analyze the relative speedup of the vectorized smoother (utilizing both the ELL and SELL-C-σ sparse matrix formats) compared to the scalar LDU baseline. This performance evaluation is conducted on both the EPAC and SG2044 architectures. Furthermore, by establishing the theoretical speedup upper bound dictated by Amdahl's Law, we will compare the empirically measured speedup across the end-to-end application against its theoretical maximum.

5.3.1 Smoother standalone performance

An analysis of the relative speedup of the smoother reveals significant considerations regarding architectural efficiency. On both platforms, the performance benefits obtained in the SpMV kernel are reflected in the performance of the isolated smoother. Specifically, a maximum speedup of 6x is observed

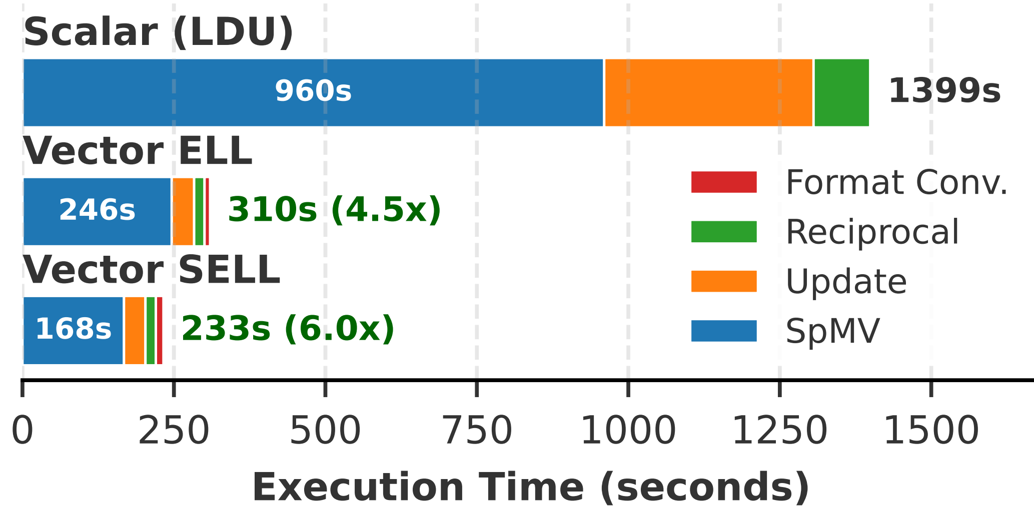


Figure 5.5: Performance profiling of the vectorized Richardson smoother on the EPAC architecture (128k cells problem).

for EPAC and 1.5x for SG2044 (both using the SELL-C-σ format), with their respective ELL variants consistently recording lower results. This strong discrepancy between the two platforms is justified by two main factors. First, as analyzed in the previous chapter, the long vector architecture of EPAC benefits considerably more from the vectorization of the SpMV kernel compared to the short vector implementation present on SG2044.

Secondly, a marked performance difference emerges in the execution of dense kernels, such as the reciprocal calculation and vector updates. On EPAC, these phases benefit from an extremely high relative speedup (up to 9.75x for the update phase and 5.45x for the reciprocal in SELL-C-σ format), exceeding the performance margins gained on the SpMV itself. Conversely, on SG2044, the manual vectorization of these specific sections proved ineffective, bringing no measured improvement over the scalar baseline.

This behavior might be attributed to how each architecture manages strictly memory-bound operations. On platforms equipped with short vectors and advanced superscalar cores (like the out-of-order C920 on SG2044), it is hypothesized that the out-of-order execution mechanism is already ca-

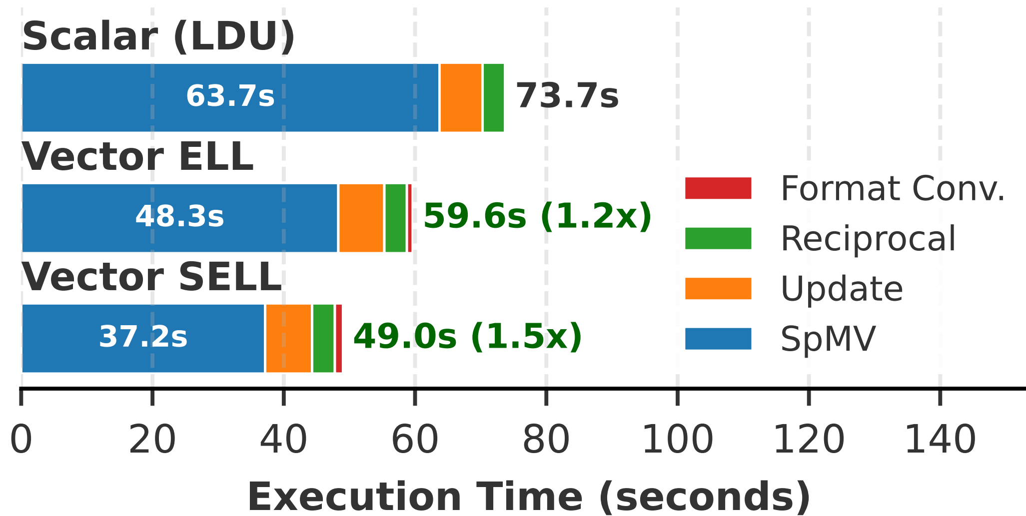


Figure 5.6: Performance profiling of the vectorized Richardson smoother on the SG2044 architecture (128k cells problem).

pable of hiding memory latency and saturating the memory bandwidth on its own, rendering vectorization potentially futile against the memory wall. In contrast, EPAC relies on the simpler, in-order Avispado scalar core, which easily stalls on cache misses. Here, long vectors become crucial to generate extensive contiguous memory transactions. This approach seems to optimally amortize main memory access latency, directly compensating for the weak in-order scalar pipeline and eliminating most loop control overhead.

5.3.2 Amdahl's Law and theoretical speedup bounds

Before analyzing the final speedup data of the end-to-end application, a preliminary study was conducted to evaluate the expected performance gain compared to the scalar baseline. This estimation is based on the relative weight of the smoother with respect to the total execution time of the application in the scalar run. According to Amdahl's Law[1], it is possible to determine the maximum theoretical speedup of an application by discriminating the portion of computation time that will be optimized from the unoptimized one.

Amdahl's Law defines the overall speedup ($Speedup_{overall}$) as:

$$Speedup_{overall} = \frac{1}{(1-P) + \frac{P}{S}} \tag{5.2}$$

where:

- P is the fraction of the execution time in the scalar version occupied by the part of the code that is enhanced or vectorized (in our case, the smoother).
- $1 - P$ is the fraction of the execution time occupied by the code that remains unmodified (the serial fraction).
- S is the local speedup factor of the vectorized portion of the code.

To find the maximum theoretical speedup (i.e., assuming an ideal scenario where the execution time of the vectorized part approaches zero, $S \to \infty$), the term $\frac{P}{S}$ becomes zero, and the formula simplifies to:

$$Speedup_{max} = \frac{1}{1-P} \tag{5.3}$$

From the data collected through the standard setup, it emerges that during the scalar execution, the smoother accounts for 48.6% of the total time for the EPAC 5.2 architecture and 43.5% for the SG2044 system. The remaining execution time encompasses all other components of the GAMG solver and the *icoFoam* application, which were not modified in this project. Therefore, these components assume the role of the non-parallelized fraction $(1 - P)$ in Amdahl's Law and will consequently determine the strict theoretical limit of the speedup.

Applying the formula for the maximum limit ($S \to \infty$), we obtain the following theoretical bounds:

- For EPAC ($P = 0.486$): $Speedup_{max,EPAC} \approx 1.945\times$
- For SG2044 ($P = 0.435$): $Speedup_{max,SG2044} \approx 1.770\times$

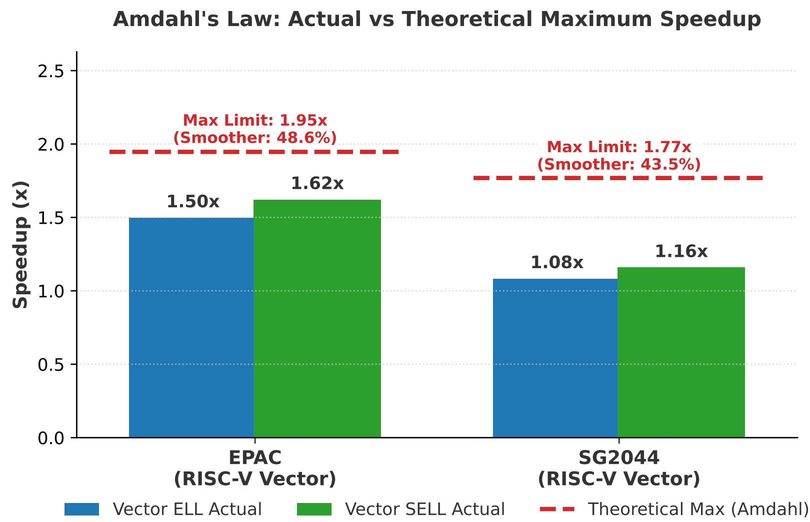


Figure 5.7: Theoretical maximum speedup according to Amdahl's Law compared to the realized application speedup.

This means that, regardless of how efficient the vectorization of the smoother may be, the application as a whole will never be able to exceed a speedup factor of $\approx 1.945\times$ on EPAC and $\approx 1.770\times$ on SG2044 compared to their respective scalar executions.

5.3.3 End-to-end simulation profiling

Having established the theoretical speedup bounds, we now evaluate the performance data for the final end-to-end simulation.

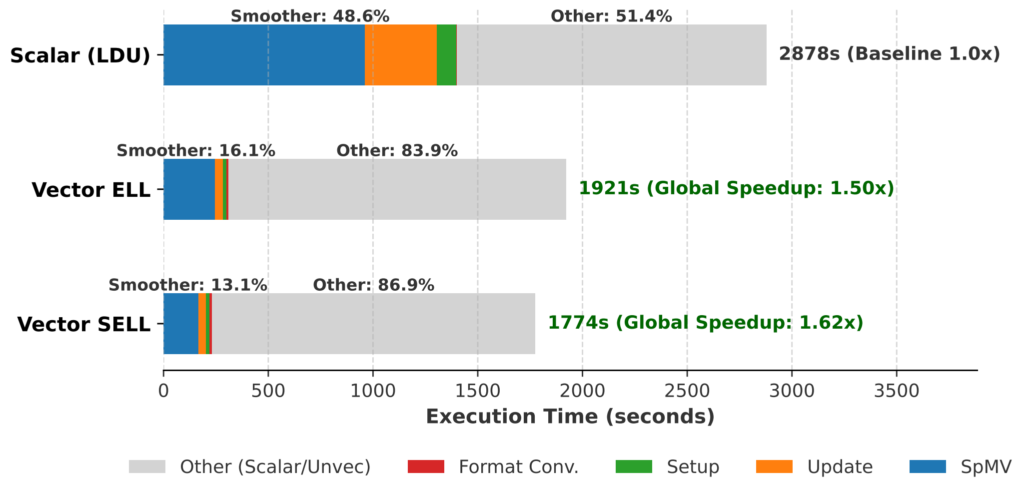


Figure 5.8: Application profiling on EPAC

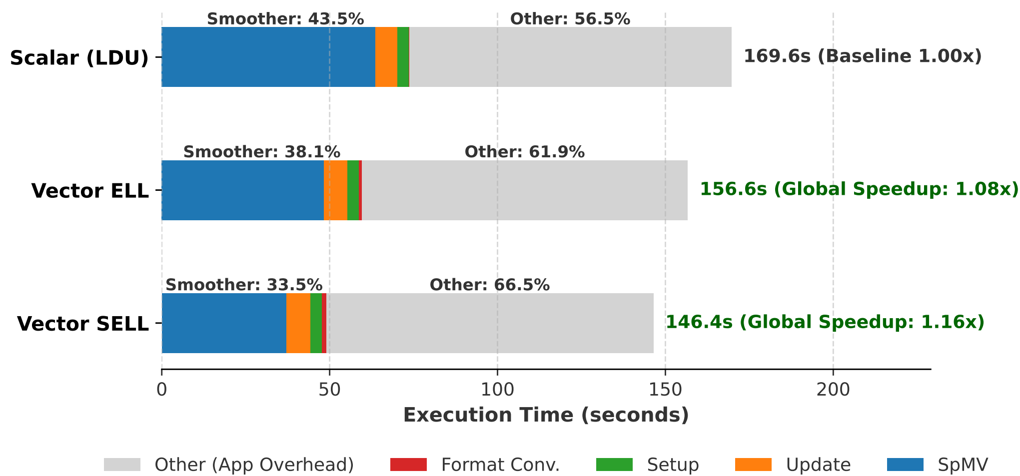


Figure 5.9: Application profiling on SG2044

For the EPAC architecture, the application achieves an overall speedup of 1.62× when using the SELL-C-σ format and 1.50× with the ELL format, compared to the scalar LDU baseline. On the SG2044 system, the improvements are more modest, peaking at 1.16× for SELL-C-σ and 1.08× for ELL.

Notably, the results obtained on EPAC approach the theoretical maximum limit dictated by Amdahl's Law (1.95×). This demonstrates a highly

efficient vectorization of the smoother, considering the strict constraints imposed by the non-parallelized fraction of the application.

Furthermore, the consistent superiority of the SELL-C-σ format over ELL on both RISC-V architectures suggests that SELL-C-σ provides a better balance between memory bandwidth utilization and vector lane efficiency for the irregular sparsity patterns typical of unstructured CFD meshes. The more pronounced performance gap on the SG2044 system can likely be attributed to differences in the vector microarchitecture and memory subsystem bottlenecks, which dictate how effectively the hardware can hide the memory latency inherent to sparse matrix-vector operations.

Chapter 6

Conclusions and Future Work

In this thesis project, we explored the feasibility of vectorizing crucial sections within the GAMG solver in the OpenFOAM framework. This was achieved by leveraging the runtime conversion of the internal matrix format into a more vector-friendly representation.

The analysis was conducted on two very different architectures: the EPAC prototype as an example of an experimental long-vector architecture, and the SG2044 processor as a commercial short-vector architecture. The results highlighted both the viability and the performance advantages of this approach on both processors. Specifically, our intrinsic-based vectorization yielded a 6× speedup for the isolated smoother on the EPAC test chip and a 1.5× speedup on the SG2044. When evaluating the complete end-to-end CFD simulation, the overall performance improvements reached 1.62× and 1.16× on the EPAC and SG2044 architectures, respectively. Most notably, the EPAC results confirmed that long-vector architectures are better suited to handle sparse, memory-bound kernels, as they can more effectively amortize memory latency and maximize memory bandwidth utilization.

Furthermore, the adoption of a non-invasive external plugin methodology demonstrates that it is possible to operate on complex and legacy frameworks without requiring extensive modifications to the core codebase. This approach allows developers to selectively optimize only the code regions of

greatest interest.

Building upon the findings of this work, the most logical future developments articulate into two main directions. First, it is essential to continue the vectorization effort across other sections of the GAMG solver. By reducing the remaining scalar bottlenecks, this further optimization would inherently increase the maximum theoretical speedup of the entire application, as dictated by Amdahl's Law. Second, future research should focus on extending the scale of the applied parallelism to multicore and multinode execution environments. Identifying the most effective distributed-memory programming model will be a key objective of this investigation. This extension would allow for a comprehensive evaluation of how distributed-memory parallelism, based on domain decomposition, interacts and scales in conjunction with the SIMD hardware parallelism provided by modern vector architectures.

List of Acronyms

AI Artificial Intelligence.

AVX Advanced Vector Extensions.

BLAS Basic Linear Algebra Subprograms.

CFD Computational Fluid Dynamics.

CG Conjugate Gradient.

COO Coordinate List.

CPU Central Processing Unit.

CSR Compressed Sparse Row.

CUDA Compute Unified Device Architecture.

DLP Data-Level Parallelism.

ELEN Element Length.

ELL ELLPACK.

EPAC European Processor Accelerator.

EPI European Processor Initiative.

FLOP Floating-Point Operation.

FMA Fused Multiply-Add.

FP Floating-Point.

FP64 64-bit Floating-Point.

FPGA Field-Programmable Gate Array.

FVM Finite Volume Method.

GAMG Geometric-Algebraic MultiGrid.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HPC High-Performance Computing.

HPCG High Performance Conjugate Gradients.

HPL High-Performance Linpack.

ILP Instruction-Level Parallelism.

ISA Instruction Set Architecture.

L1 Level 1.

LDU Lower-Diagonal-Upper.

LLC Last Level Cache.

LMUL Length Multiplier.

LU Lower-Upper.

MG Multigrid.

MIMD Multiple Instruction, Multiple Data.

MISD Multiple Instruction, Single Data.

MPI Message Passing Interface.

NEON Arm Neon.

NNZ Number of Non-Zeros.

NoC Network-on-Chip.

OpenFOAM Open Field Operation and Manipulation.

PDE Partial Differential Equation.

RHS Right-Hand Side.

RVV RISC-V Vector Extension.

SELL-C-σ SELL-C-σ.

SEW Selected Element Width.

SIMD Single Instruction, Multiple Data.

SIMT Single Instruction, Multiple Threads.

SISD Single Instruction, Single Data.

SPD Symmetric and Positive-Definite.

SPMD Single Program, Multiple Data.

SpMV Sparse Matrix-Vector multiplication.

SVE Scalable Vector Extension.

SYCL Single-source C++ Heterogeneous Programming.

TLP Thread-Level Parallelism.

VLA Vector Length Agnostic.

VLEN Vector Register Length.

VPU Vector Processing Unit.

VRF Vector Register File.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. AFIPS, 1967.

[2] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*, 2026.

[3] Utkarsh Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., 2015.

[4] Ashish Baghudana and Vartan Kesiz Abnousi. A parallel algorithm for lda using stochastic collapsed variational bayesian inference. 05 2018.

[5] Barcelona Supercomputing Center. Paraver - performance data analyzer. https://tools.bsc.es/paraver, 2026. Accessed: 2026-03-06.

[6] Marc Blancafort, Roger Ferrer, Guillaume Houzeaux, Marta Garcia-Gasulla, and Filippo Mantovani. Exploiting long vectors with a CFD code: a co-design show case, May 2024.

[7] Simone Bnà, Giuseppe Giaquinto, Ettore Fadiga, Tommaso Zanelli, and Francesco Bottau. SPUMA: a minimally invasive approach to the GPU porting of OPENFOAM. *arXiv preprint arXiv:2512.22215*, 2025.

[8] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. The hpcg benchmark: a new metric for ranking high performance computing sys-

tems. *The International Journal of High Performance Computing Applications*, 30(6):610–624, 2016.

[9] European Processor Initiative. European processor initiative (EPI), 2026.

[10] Roger Ferrer Ibanez. Introduction to the RISC-V Vector Extension. Presented at the 2022 ACM Summer School on HPC and AI, 2022.

[11] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[12] Constantino Gómez, Filippo Mantovani, Erich Focht, and Marc Casas. Efficiently running spmv on long vector architectures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 292–303, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Christopher Greenshields and Henry Weller. *Notes on Computational Fluid Dynamics: General Principles*. CFD Direct Ltd, Reading, UK, 2022.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2017.

[15] Antti Honkela, Tapani Raiko, Mikael Kuusela, Matti Tornio, and Juha Karhunen. Approximate riemannian conjugate gradient learning for fixed-form variational bayes. *Journal of Machine Learning Research*, 11:3235–3268, 11 2010.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2026.

[17] Ronan Keryell. A single-source c++ standard for heterogeneous computing, 2019.

[18] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, abs/1307.6209, 2013.

[19] Hao Li, Xinhai Xu, Miao Wang, Li Chao, Xiaoguang Ren, and Xuejun Yang. Insertion of petsc in the openfoam framework. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2:1–19, 08 2017.

[20] Filippo Mantovani, Fabio Banchelli, and Pablo Vizcaino. The EPI vector RISC-V accelerator: Architecture, platform, tools and hands-on. Institute for Advanced Simulation Seminar, Barcelona Supercomputing Center (BSC), April 2023. Presented on April 25, 2023.

[21] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, Jonnatan Mendoza, Ivan Vargas, César Hernandez, Joan Cabre, Lina Khoirunisya, Mustapha Bouhali, Julian Pavon, Francesc Moll, Mauro Olivieri, Mario Kovac, Mate Kovac, Leon Dragic, Mateo Valero, and Adrian Cristal. Vitruvius+: An area-efficient risc-v decoupled vector coprocessor for high performance computing applications. *ACM Trans. Archit. Code Optim.*, 20(2), March 2023.

[22] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda, 03 2008.

[23] NVIDIA Corporation. Sparse matrix storage formats — NVIDIA performance libraries (NVPL) documentation. https://docs.nvidia.com/nvpl/latest/sparse/storage_format/sparse_matrix.html, 2026. Accessed: 2026-02-21.

[24] OpenCFD Ltd. Openfoam tutorial guide: 2.2 flow around a cylinder. https://www.openfoam.com/documentation/tutorial-guide/2-incompressible-flow/2.2-flow-around-a-cylinder, 2024. Accessed: 2026-03-04.

[25] Pavlord98. Laminar flow around a cylinder - openfoam simulation. https://github.com/Pavlord98/Laminar_flow_around_a_cylinder. GitHub repository.

[26] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. https://www.netlib.org/benchmark/hpl/. Accessed: 2024.

[27] Yousef Saad. *Iterative methods for sparse linear systems.* SIAM, 2nd edition, 2003.

[28] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoki Shida, Ikuo Miyoshi, et al. Co-design for A64FX manycore processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[29] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical Report Edition 1 1/4, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, August 1994. Available at: http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

[30] Keichi Takahashi, Soya Fujimoto, Satoru Nagase, Yoko Isobe, Yoichi Shimomura, Ryusuke Egawa, and Hiroyuki Takizawa. *Performance Evaluation of a Next-Generation SX-Aurora TSUBASA Vector Supercomputer*, pages 359–378. Springer Nature Switzerland, 2023.

[31] Emanuele Venieri, Simone Manoni, Gabriele Ceccolini, Giacomo Madella, Federico Ficarelli, Daniele Gregori, Andrea Acquaviva, Luca Benini, and Andrea Bartolini. Monte cimone v2: Hpc risc-v cluster evaluation and optimization, 2025.

[32] Pablo Vizcaino, Georgios Ieronymakis, Nikolaos Dimou, Vassilis Papaefstathiou, Jesus Labarta, and Filippo Mantovani. Short reasons for long vectors in hpc cpus: A study based on risc-v. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, pages 1543–1549, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Andrew Waterman and Krste Asanović. The risc-v instruction set manual, volume i: Unprivileged isa. Technical Report 20191213, RISC-V International, 2019. Document Version 20191213.

Ringraziamenti

Vorrei ringraziare il Prof. Bartolini per l'opportunità che mi ha concesso e per la preziosa guida durante questo progetto di tesi. Ringrazio i miei correlatori, Federico Ficarelli, Filippo Barbari e Simone Bnà, per avermi costantemente affiancato e supportato. Li ringrazio profondamente per l'ottimo rapporto che si è instaurato dal punto di vista umano, nella speranza di poter continuare a lavorare insieme anche in futuro.

Ringrazio mio padre Domenico, mia madre Sara, mio fratello Marco e tutto il resto della famiglia, da sempre e per sempre uniti, ai quali devo tutto.

Un ultimo ringraziamento va a tutti i miei cari amici, da quelli storici con cui sono cresciuto e che ho ancora al mio fianco ogni giorno, fino a quelli conosciuti durante quest'ultimo percorso di università.