Integration and performance analysis of parallel RISC-V architectures

Casio P. Krebs[1], **Guido Araujo**[1], **Lucas Wanner**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas, SP – Brazil

r264953@dac.unicamp.br, {guido,lucas}@ic.unicamp.br

Abstract. *Vector and matrix architectures accelerate workloads by exploiting data-level parallelism and reducing instruction overhead, but using them typically requires manual code changes. This work explores the Hwacha vector coprocessor and Gemmini matrix accelerator, alongside the BOOM superscalar RISC-V core, aiming to automate their activation. The SMR code rewriting tool was extended with libraries for data preparation, movement, and Hwacha/Gemmini activation for GEMV and GEMM. Integrated with Verilator, a simulation environment evaluated performance across seven Polybench kernels. Results show SMR can activate hardware accelerators without modifying the base code, enabling efficient acceleration more easily.*

1. Introduction

Multiplication and accumulation operations (MAC) involving regular structures, such as matrices and vectors, constitute one of the most fundamental routines in computational problems. Applications involving machine learning, such as neural networks and convolutional networks, may have matrix operations accounting for approximately 70% of the total computational cycles during the training phase [Qin et al. 2020]. As a result of the growing demand for these routines, there has been a significant interest in developing libraries [Oliphant 2006, Xianyi and Kroeker 2020] and algorithms [Strassen 1969, Coppersmith and Winograd 1987, Williams 2012] that aim to accelerate Matrix-Matrix and Matrix-Vector multiplication operations, also known as GEMM and GEMV, respectively. However, as they are software-based solutions, they tend to be limited by the core, which needs to meet area and energy demands.

Historically, the improvement in processing capability has been mainly driven by Moore's Law and Dennard scaling. However, Dennard [Dennard et al. 2007] and other researchers have already identified that transistor-based circuits have reached a critical manufacturing point, where the exponential increase in leakage current prevents the continuation of the original scaling, leading to a rise in power density and overall energy cost of the chip. In this scenario, a research target to further enhance performance is the development of architectures that aim to serve a specific application purpose. A common practice among designers is to utilize these dedicated architectures as accelerators within the chip, operating in conjunction with the general-purpose processor, aiming to improve the overall chip performance by speeding up highly compute-intensive routines and consequently freeing up the processor to continue performing useful computation.

Applications that rely on GEMM and GEMV operations are the target of optimizations by various accelerator projects. Despite with the same objective in

mind, various accelerator topologies have been developed, including vector architectures [Lee et al. 2015, Lomont 2011, Intel 2022] and systolic arrays [Moss et al. 2018, Qin et al. 2020, Liu et al. 2020, NVIDIA 2024]. However, integrating software and hardware is a complex task in the development of these dedicated architectures. With an Instruction Set Architecture (ISA) that extends the native processor instructions, programmers need prior knowledge of the system's functionalities to adapt the base code of the application for their activation. Therefore, the development of high-level libraries to facilitate their use is recommended since different projects can be present in the same System-on-Chip (SoC), and when not activated, a fraction of the hardware remains underutilized.

In this scenario, this proposal aimed to implement and evaluate the Hwacha vector coprocessor [Lee et al. 2015] and the Gemmini matrix processing accelerator [Genc et al. 2021]. One advantage of these projects is their integration into the Chipyard hardware description framework [Amid et al. 2020], which provides tools for implementing and evaluating System-on-Chip (SoC) components. As an open-source framework, it supports a wide variety of processors, including the superscalar RISC-V processor BOOM [Zhao et al. 2020]. To provide high-level support, the programming interface of the architectures was extended in the Source Matching and Rewriting (SMR) compilation flow [Espindola et al. 2023], a tool that performs code matching and rewriting. For this purpose, optimized libraries were developed to perform the routines of these architectures, allowing Hwacha and Gemmini to be used without the need for compiler knowledge, activation functions, and, most importantly, without modifying the base code of the application. Finally, the performance of the developed libraries was compared to the software solution OpenBLAS [Xianyi and Kroeker 2020], executed on the superscalar BOOM processor.

2. Integration between Processor and Accelerators in the Chipyard Framework

The Chipyard ecosystem is an open-source hardware description framework that allows for designing, evaluating, and ultimately implementing System-on-Chip (SoC) components based on the RISC-V Instruction Set Architecture (ISA) [Amid et al. 2020]. Chipyard supports the generation of the superscalar BOOM processor the Hwacha coprocessor, and the Gemmini accelerator, along with peripheral structures such as interconnections and cache. These accelerators are linked to the BOOM core through the RoCC connection interface, which is integrated into the Chipyard ecosystem.

2.1. Processador Berkeley Out-of-Order Machine (BOOM)

The Berkeley Out-of-Order Machine (BOOM)[Zhao et al. 2020], platform offers an out-of-order processor generator, featuring a conditional branch prediction stage, dynamic instruction scheduling, and operating on the RV64GC instruction set. The BOOM core, currently in its third version, has a 7-stage pipeline with an asynchronous Commit stage. During development, the generator allows modifications to structural parameters like the L1 cache, the number of physical registers, and the Fetch and Decode stages to enhance hardware resource utilization by optimizing the number of instructions read from the instruction cache and decoded in a single cycle. The Fetch stage employs a two-level branch prediction mechanism, using the Next-Line Predictor (NLP) and the Backing Predictor

(BPD). For dynamic scheduling, BOOM uses register renaming and dynamic dispatch stages, utilizing a Physical Register File (PRF) for explicit renaming design.

2.2. Hwacha Coprocessor

Hwacha is an open-source, parameterizable vector coprocessor generator [Lee et al. 2015], designed with a decoupled memory-access and execution architecture [Smith 1984]. This design allows it to issue memory requests concurrently with vector computation, supporting configurable floating-point precision and optional predicated instruction execution at compile time.

The coprocessor includes a dedicated Frontend interface that fetches vector instructions. A single instruction from the host processor, referencing a program counter (PC), activates Hwacha's instruction block execution. While Hwacha processes vector operations, the main core continues executing scalar tasks independently. Hwacha comprises four main components: an L1 instruction cache, the Vector Runahead Unit (VRU), Vector Lanes, and a Scalar Unit. The Scalar Unit performs instruction fetch and dispatches execution to the vector lanes. Each Vector Lane is composed of a Vector Execution Unit (VXU) and a Vector Memory Unit (VMU). The VXU contains the Vector Register File (VRF), Predicate Register File (PRF), and functional units for half, single, and double-precision operations. It is organized into four banks, each with an SRAM segment and functional units interconnected via a crossbar that shares operand, predicate, and result lines. The VMU handles data movement between the VXU and the memory hierarchy (e.g., L2 cache), ensuring efficient feeding of operands to the execution units.

2.2.1. Data Flow

Hwacha adopts a programming model that separates instructions into two blocks: a control thread and a work thread. The control thread, executed by the core, issues scalar instructions to configure the Hwacha Vector Unit, while the work thread contains only vector operations. The RoCC interface forwards control thread instructions to the Scalar Unit and the Vector Runahead Unit (VRU) via the Vector Command Queue (VCMDQ) and Vector Runahead Command Queue (VRCMDQ). Upon decoding the activation instruction, the Scalar Unit initiates the Fetch stage of vector instructions from the PC value provided, continuing until the fetch block ends.

In the decode stage, the Scalar Unit dispatches instructions to the master sequencer, which synchronously issues operations to all Vector Lanes. Each lane executes independently. When the vector length exceeds the number of available banks, the Scalar Memory Unit (SMU) preloads the full vector into vector registers, as determined by a configurable vector length register. The master sequencer splits the data into contiguous blocks and dispatches them across lanes. An internal sequencer tracks execution progress within each lane, enabling dynamic instruction dispatch. Unlike the BOOM core, Hwacha lacks register renaming; instructions stall until their operands are ready. Once available, instructions are routed by the expander to the appropriate execution units. Results are stored in output queues, and the master sequencer manages their withdrawal and placement into vector registers asynchronously to ensure correct sequencing.

2.3. Gemmini Accelerator

Gemmini is an open-source accelerator generator targeting Deep Neural Network (DNN) workloads, fully integrated into the Chipyard ecosystem [Genc et al. 2021]. It is based on a systolic array (SA) architecture, where each Processing Element (PE) performs MAC operations on partial products. Gemmini uses a Scratchpad to explicitly store input and output matrix blocks, maximizing data reuse. An accumulator with adder units reduces writes of partial results. Both are implemented with SRAM to ensure high bandwidth. Pre-processing units support operations such as matrix transpose and scalar multiplication before loading data into the SA. Gemmini accelerates matrix operations of the form $C = \alpha \cdot A \cdot B + \beta \cdot D$.

Gemmini integrates two Direct Memory Access (DMA) units: one for fetching input matrices from L2 to the Scratchpad, and another for writing results back to L2. These DMAs use virtual addresses and share a TLB. To hide memory latency, data movement can proceed in parallel with SA execution, with a Reorder Buffer ensuring in-order completion. A low-level C API supports DNN primitives such as matrix multiplication, additions, and convolutions, with or without pooling. The API is generated based on architectural parameters and performs tiling for matrix dimensions larger than the SA, optimizing data reuse between cache and Scratchpad.

2.3.1. Systolic Array

Systolic arrays [H.T.Kung 1981, H.T.Kung 1982] consist of PEs arranged in a grid, communicating with neighbors to perform the same operation in parallel. Data flows through local registers instead of external memory, enabling high throughput and scalability. Typically, two matrices traverse the array simultaneously: one injected row-wise (shifted downward), and the other column-wise (shifted right). Each PE multiplies inputs and accumulates partial results. Final outputs emerge after complete traversal of a row and column.

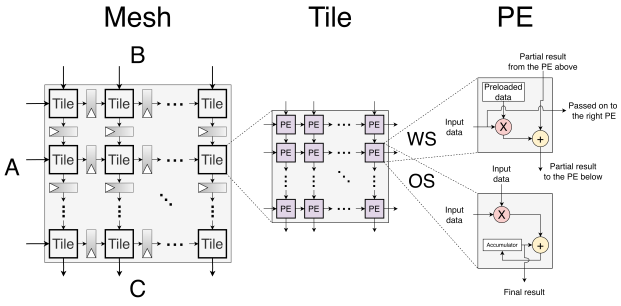

Figure 1. SA and MACs units overview

Gemmini employs a two-level SA structure: an inner layer of Tiles formed by tightly connected PEs, and an outer Mesh layer where Tiles are connected via pipeline registers (Figure 1). Communication is restricted to immediate neighbors. Gemmini supports Weight-Stationary (WS) and Output-Stationary (OS) dataflows. In WS, weights (matrix B) remain fixed in the PEs, while activations (matrix A) are streamed through. Each PE multiplies the stored weight by the streamed value, accumulates it with data from

the PE above, and passes it downward. In OS, both A and B propagate through the array, with intermediate results accumulated in-place within the SA. Final values are stored only after all accumulations, reducing Scratchpad writes.

3. Related Works

With the increasing prevalence of applications expressible as GEMM and GEMV operations, there has been a growing demand for vector and systolic architectures due to their high parallelism. While systolic architectures are promising, vector architectures are more widely adopted, as most commercial CPUs implement fixed-length SIMD instructions. This section reviews state-of-the-art solutions and optimizations for these architectures.

3.1. Vector Architecture

SIMD instructions extending core ISAs represent the most widespread form of vector processing. Intel introduced SIMD support with the MMX extension in 1996, enabling 64-bit integer operations [Peleg et al. 1997]. This was followed by SSE (128-bit, single-precision floating-point) in 1999 and SSE2 (double-precision support) in 2000. The AVX instruction set, introduced in 2008 and deployed commercially in 2011 with the Sandy Bridge processors, added 256-bit registers for floating-point operations [Lomont 2011]. In 2016, AVX-512 extended this to 512-bit registers and 32 registers [Intel 2022]. AMD also supports these extensions. Libraries such as OpenBLAS leverage AVX for optimized vector computation on x86 platforms.

3.2. Systolic Architecture

Systolic arrays, introduced in the late 1970s [H.T.Kung 1981], have been applied to solving linear systems [H.T.Kung 1982, Gentleman and Kung 1982], convolutions [Kung and Song 1981], and matrix factorizations [Schreiber 1983]. Their potential to accelerate ML kernels has renewed interest in this paradigm. Google's TPU remains a striking example, featuring a 256 × 256 systolic array performing 8-bit MAC operations with high throughput [Jouppi et al. 2023]. Nvidia's GPU architectures evolved from Pascal (2017) with 4×4 matrix units, to Turing (2018) with 8×8 meshes and INT32 support, Ampere (2020) with 16×16 meshes and FP64 support, and more recently Hopper (2022) [NVIDIA 2022] and Blackwell (2024) [NVIDIA 2024], which introduced advances in accuracy, performance, and efficiency.

Central academic efforts have focused on reducing memory accesses and maximizing the utilization of PEs (Processing Elements). Eyeriss [Chen et al. 2016] introduced the Row Stationary (RS) dataflow to improve data reuse and energy efficiency, achieving up to 2.5x better performance than other dataflows. Subsequently, Eyeriss v2 [Chen et al. 2019] improved the handling of sparse matrices by identifying non-zero elements before loading, achieving a 5.3x speedup and a 3.9x energy reduction. The SIGMA accelerator [Qin et al. 2020] further increased PE utilization (up to 82 %) in sparse contexts, although with a 37.7 % increase in area. The Systolic Tensor Array [Liu et al. 2020] introduced the concept of Density Bounded Block (DBB), activating only the necessary PEs, reducing MAC units by up to 75 %, but requiring external tiling if density limits are exceeded.

4. Code Generation

Generating parallel code is challenging due to the difficulty in identifying parallelizable regions, such as loops that can be vectorized. This process requires analyzing data dependencies between iterations. As a result, automatic parallelization often demands modifications to the original code. To address this, we adopted the Source Matching and Rewriting (SMR) tool and developed custom libraries that map GEMV and GEMM calls to OpenBLAS, including activation for Hwacha and Gemmini.

4.1. Matching and Rewriting (SMR)

SMR uses a declarative language called PAT, in which the matching and rewriting functions are explicitly defined [Espindola et al. 2023]. The PAT file contains two parts: the first defines the match conditions, and the second describes the rewrite function. Both parts can be written in different languages. SMR does not verify the correctness of rewrites; the programmer is responsible for that.

The tool takes an input code file and a PAT file, both reduced to their MLIR dialects. SMR generates Control Dependence Graphs (CDGs) for both the input code and the matching function, then compares their structures. If control structures match, it builds Data Dependence Graphs (DDGs) to check data type compatibility, ensuring variable names do not affect comparison. Upon finding a match, the tool replaces the original code with the rewrite version in MLIR and generates an optimized executable. A key benefit of SMR is its support for cross-language rewriting, allowing functions to be matched and rewritten across different programming languages via MLIR. This feature enables the integration of Hwacha and Gemmini-specific activation code into existing matrix operation kernels, independent of the original language used.

4.2. Hwacha Integration in SMR

The Hwacha project provides a C library for tiled matrix multiplication targeting the pattern $C = A \cdot B + C$ [UC Berkeley Architecture Research 2022]. The library loads a row of matrix C into the destination register, a row of matrix B into the source register, and one scalar from matrix A into a shared register. This emulates a weight-stationary behavior, where B remains fixed while A and C propagate. If the width of C and B exceeds the vector register length, the computation is segmented, processing each row in parts.

Since the library does not support matrix transposition or scalar multiplication, an auxiliary layer was developed to enable SGEMM and SGEMV acceleration. This layer includes layout conversion and scalar operations to conform to OpenBLAS calls. A main limitation arises from the orientation mismatch: SMR processes Fortran code (column-major), while Hwacha expects C-style (row-major). Therefore, transpositions are automatically inserted in reverse order.

To generalize SGEMM execution, different flows are used depending on the layout of the operands. When both matrices are in standard format, the operands are swapped to ensure the correct layout in memory and leverage the identity $(A \cdot B)^T = B^T \cdot A^T$. If only one matrix is transposed, it is converted and reordered. When both are transposed, the result is transposed by the core after computation. Scalar multiplication is handled by the core before storing operands in registers. Blocking is applied to partition matrices into sub-blocks that fit into the cache, improving locality and reducing DRAM pressure.

In SGEMV operations, columns of matrix A are loaded into the vector register while vector acts as weights. Scalar multiplication is applied early to reduce overhead. For transposed matrices, the core loads the relevant addresses and stores them in an auxiliary vector accessed by the coprocessor. This introduces additional memory accesses, which are mitigated by integrating the transposition step into the Hwacha activation.

4.3. Gemmini Integration in SMR

As presented in Section 2.3, the Gemmini library supports matrix multiplication in the form $C = \alpha \cdot A \cdot B + \beta \cdot D$ [Genc et al. 2021]. However, in the SMR integration, the library accepts transposition for only one matrix per call and does not allow D to be transposed. Given that Gemmini operates with row-major layout, the multiplication result must be transposed to match the expected format. This is addressed using the matrix identity that the transpose of a product equals the product of transposed operands in reverse order.

To handle SGEMM operations, three execution flows were implemented. In the case where neither operand is transposed, Gemmini is called with A and B swapped, producing the result in the correct layout. When only one operand is transposed, the operand is swapped and the transposition flag is applied. When both operands are transposed, Gemmini is called with the original order, and the final result is transposed by the core.

For SGEMV operations, scalar multiplications $\alpha \cdot x$ and $\beta \cdot y$ are performed in the BOOM core before accelerator activation. This approach was more efficient than relying on Gemmini's internal pipeline, as the core can dispatch multiple floating-point instructions per cycle. Transposition control is also adjusted in the SMR flow to reverse the layout relative to the original OpenBLAS call. Additionally, if $\beta = 0$, the integration skips memory fetches for D, and Gemmini initializes its accumulator with zero, avoiding unnecessary DRAM accesses.

5. Evaluation

This section details the evaluation methodology for the Hwacha vector coprocessor and the Gemmini matrix accelerator. The process includes defining and configuring the evaluation platform, describing the selected Polybench applications, and comparing performance against the OpenBLAS software library.

5.1. Definition and Configuration of the Evaluation Platform

All experiments were conducted using Verilator [Verilator 2022], a cycle-accurate simulator that translates Verilog HDL into C++ for simulation. The target system uses a 4-wide BOOM processor instantiated via the `WithNMegaBooms` class in the Chipyard framework, with a 32KB 8-way L1 cache and a 512KB 8-way L2 cache (Sifive InclusiveCache [Specification 2018]), both with 64-byte blocks. DRAM is modeled with the `BlackBoxSimMem` module, using a fixed 60-cycle latency.

5.2. Applications

The evaluation includes seven applications from the Linear Algebra suite of the Polybench benchmark [Narayan and Pouchet 2012], all implemented using single-precision floating-point data. For SGEMM-like behavior, the `gemm` kernel is used, with the scalar coefficients `alpha` and `beta` enabled. For SGEMV-like behavior, the `mvt` kernel is

adopted, which performs two independent matrix-vector multiplications. Executables were generated using the SMR rewriting tool (Section 4.1), which maps naïve loops to SGEMM and SGEMV calls, enabling Hwacha and Gemmini acceleration. Cache flushing is performed before each execution to ensure consistent memory behavior across runs. Performance is reported in FLOPS for direct throughput comparison with OpenBLAS [Xianyi and Kroeker 2020].

5.3. Hwacha Evaluation

The performance evaluation of the Hwacha coprocessor was carried out in two stages. First, the scalability of the accelerator was analyzed by varying the number of available Vector Lanes. Next, performance bottlenecks related to data access patterns and library implementation were investigated as the input matrix dimensions increased. The architectural parameters that directly influence Hwacha's performance are based on the original model presented by Lee et al. [Lee et al. 2015]. The configured implementation operated at a frequency of 1 GHz and supported 1, 2, or 4 Vector Lanes, each containing 4 execution banks.

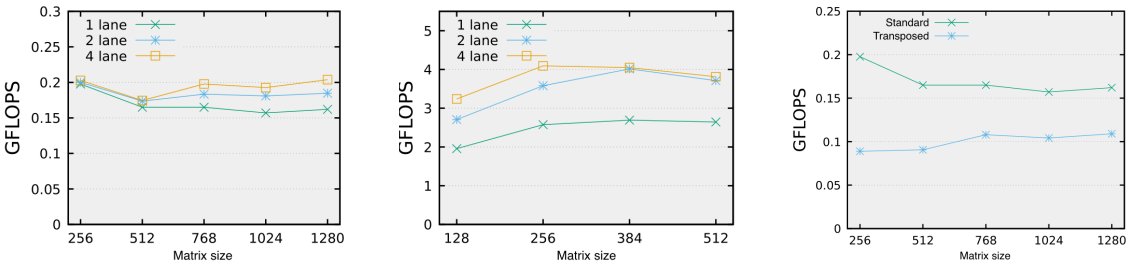

Figure 2. Performance in GFLOPS for the SGEMV (a) and SGEMM (b) kernels, varying the number of Vector Lanes from 1, 2 and 4. (c) Performance impact in GFLOPS when running the SGEMV kernel with and without input matrix transposition

For SGEMV (Figure 2a), performance remained almost constant with more lanes due to its memory-bound nature: DRAM access latency dominates execution, and the master sequencer stalls until all lanes finish. With matrices larger than 512×512, cache reuse reduces latency, improving throughput. In SGEMM (Figure 2b), higher data reuse increases throughput: a single lane reaches 66.1% of peak (4 GFLOPS/lane), but performance drops to 46.4% and 23.8% for 2 and 4 lanes, respectively, showing early memory bandwidth saturation.

Hwacha's Scalar Memory Unit cannot issue non-contiguous memory requests, requiring pre-processing for transposed matrices. Using one lane, SGEMV with transposition adds $2 \times N \times M$ memory accesses. For a 1280×1280 matrix, execution time rose from 10.1 to 15.0 million cycles (32.7% throughput drop), equivalent to 12.3 million extra DRAM accesses. The library mitigates this by storing transposed blocks in intermediate vectors to exploit L2 cache reuse, but limited cache size prevents full matrix allocation, sustaining part of the penalty.

5.4. Gemmini Evaluation

The evaluation of the Gemmini accelerator focused on analyzing the impact of dataflows on the performance of SGEMM and SGEMV operations. While the original implemen-

tation described by Genc et al. [Genc et al. 2021] features a 16×16 systolic array with 8-bit integer units, the present experiments modified the processing units to support 32-bit floating-point operations using the `GemminiFPConfigs` class. The evaluated configuration operates at 1 GHz and includes one tile with an 8×8 grid of Processing Elements (PEs), resulting in a total of 64 units. The architecture also incorporates a 256 KB Scratchpad memory and a 64 KB Accumulator buffer. These parameters were selected to balance computational capacity with efficient data movement and locality.

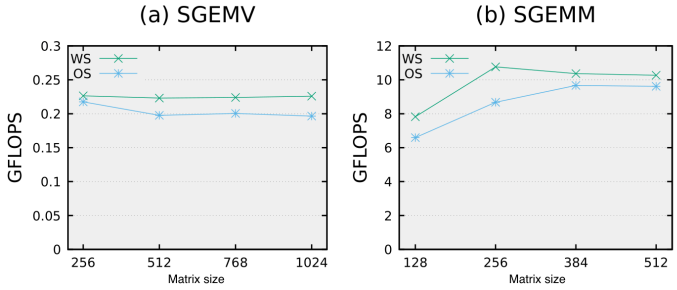

Figure 3. Performance in GFLOPS in the (a) SGEMV and (b) SGEMM kernels, evaluating the Weight-Stationary (WS) and Output-Stationary (OS) data flow

As described in Section 2.3.1, two dataflow modes were evaluated: Weight-Stationary (WS) and Output-Stationary (OS). The WS mode maintains matrix B fixed in the PEs, minimizing memory accesses and improving temporal locality. The OS mode, in contrast, maximizes accumulation reuse by preserving output positions. Since the applications in this study perform only a single propagation step, WS demonstrated superior performance for both SGEMM and SGEMV, as shown in Figure 3. For smaller matrix sizes, such as 128×128, the Scratchpad is underutilized, as it can accommodate all input and output data simultaneously. When matrix sizes exceed 256×256, the Scratchpad is fully allocated, enabling reuse of intermediate results and significantly increasing throughput due to low-latency access provided by the SRAM-based memory.

5.5. OpenBLAS

To establish a software baseline, version 0.3.20 of the OpenBLAS library was employed. For fair comparison, Hwacha was configured with a single Vector Lane, while Gemmini was operated in Weight-Stationary mode. Both SGEMM and SGEMV routines were tested using OpenBLAS and compared against the accelerated implementations.

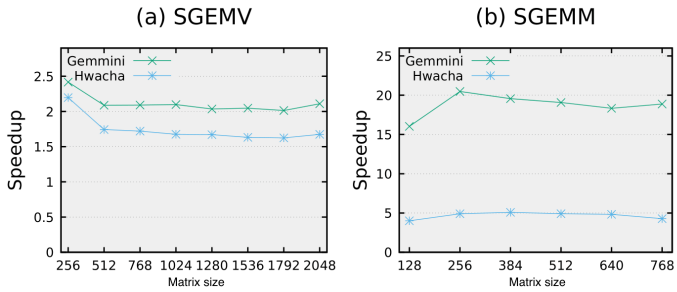

Figure 4. Performance of SGEMM and SGEMV kernel execution when run on Hwacha and Gemmini, compared to core execution

As depicted in Figure 4, for SGEMV, Hwacha achieved a maximum speedup of 1.7×, while Gemmini reached 2.1×. Despite having 64 PEs, Gemmini's rigid structure

limited its utilization during vector-based workloads, as only one row of the array was effectively used. This constraint arises from the requirement to maintain correct accumulation across columns during vector propagation. In SGEMM workloads, however, Gemmini fully exploited all 64 PEs, achieving speedups close to 19×. Hwacha, under the same configuration, delivered a speedup of approximately 4.9× using 768×768 matrices. These results underscore the advantage of Gemmini's Scratchpad and local data movement across PEs, which allow computation and memory access to proceed concurrently with minimal latency. As discussed previously, Hwacha's performance shows minimal improvement when scaling the number of Vector Lanes, highlighting its susceptibility to memory bottlenecks. Gemmini, by contrast, benefits from local data reuse in its systolic structure, especially when operand sizes exceed the Scratchpad capacity.

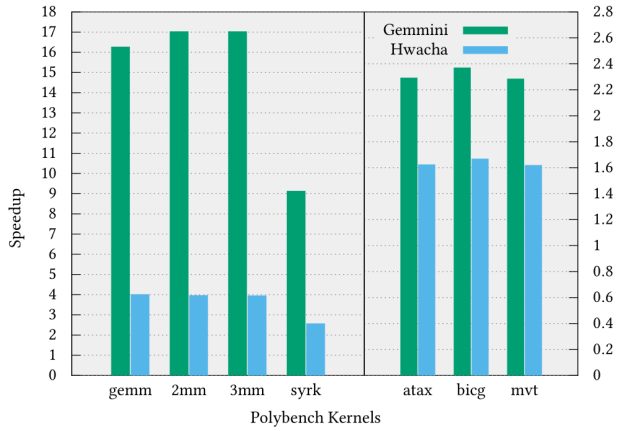

Figure 5. Performance of Polybench kernels when run on Hwacha and Gemmini, compared to core execution

A broader evaluation is presented in Figure 5. In kernels such as `gemm`, `2mm`, and `3mm`, which were rewritten by SMR to perform one or more SGEMM calls, Gemmini achieved speedups between 16× and 17×, while Hwacha delivered up to 4× gains. In the `syrk` kernel, although OpenBLAS provides an optimized implementation, SMR replaced the computation with two SGEMM calls, leading to lower acceleration: 9.1× for Gemmini and 2.6× for Hwacha. For more memory-bound kernels like `atax`, `bicg`, and `mvt`, Gemmini showed gains between 2.28× and 2.37×, while Hwacha remained between 1.61× and 1.67×. The limited arithmetic intensity and additional overhead from matrix transposition reduce acceleration effectiveness, especially for Hwacha.

6. Final Considerations

This work evaluated the Hwacha vector coprocessor and the Gemmini matrix accelerator for accelerating GEMM and GEMV operations on RISC-V architectures. Custom libraries were developed to abstract activation routines, and the SMR tool enabled integration without modifying the base application code. Hwacha achieved up to 4× speedup in SGEMM and 1.6× in SGEMV, while Gemmini reached up to 17× and 2.3×, respectively, compared to OpenBLAS. These results demonstrate the effectiveness of both architectures for dense matrix operations. However, performance is limited by memory system latency, especially in memory-bound applications like SGEMV. Despite this, the integration of SMR and high-level libraries proved effective in simplifying accelerator usage, making them accessible without requiring low-level code intervention.

References

Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Bachrach, J., Shao, S., Nikolić, B., and Asanović, K. (2020). Invited: Chipyard - an integrated soc research and implementation environment. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

Chen, Y.-H., Emer, J., and Sze, V. (2016). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379.

Chen, Y.-H., Yang, T.-J., Emer, J., and Sze, V. (2019). Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308.

Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–6.

Dennard, R. H., Cai, J., and Kumar, A. (2007). A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics*, 51(4):518–525.

Espindola, V., Zago, L., Yviquel, H., and Araujo, G. (2023). Source matching and rewriting for mlir using string-based automata. *ACM Trans. Archit. Code Optim.*, 20(2).

Genc, H., Kim, S., Amid, A., Haj-Ali, A., Iyer, V., Prakash, P., Zhao, J., Grubb, D., Liew, H., Mao, H., Ou, A., Schmidt, C., Steffl, S., Wright, J., Stoica, I., Ragan-Kelley, J., Asanovic, K., Nikolic, B., and Shao, Y. S. (2021). Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774.

Gentleman, W. M. and Kung, H. T. (1982). Matrix triangularization by systolic arrays. In *Real-Time Signal Processing IV*, volume 298, pages 19–26. SPIE.

H.T.Kung, C. E. L. (1981). Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA.

H.T.Kung, H.-T. (1982). Why systolic architectures? *Computer*, 15(01):37–46.

Intel (2022). *Intel Architecture Instruction Set Extensions and Future Features*. Intel.

Jouppi, N. P., Kurian, G., Li, S., Ma, P., Nagarajan, R., Subramanian, S., et al. (2023). Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, pages 1–14.

Kung, H. T. and Song, S. W. (1981). A systolic 2-d convolution chip. Technical report, Carnegie Mellon University, Department of Computer Science.

Lee, Y., Ou, A., Schmidt, C., Karandikar, S., Mao, H., and Asanović, K. (2015). The hwacha microarchitecture manual, version 3.8.1. Technical report, EECS Department, University of California, Berkeley.

Liu, Z.-G., Whatmough, P. N., and Mattina, M. (2020). Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*, 23.

Moss, D. J. M., Krishnan, S., Nurvitadhi, E., Ratuszniak, P., Johnson, C., Sim, J., Mishra, A. K., Marr, D., Subhaschandra, S., and Leong, P. H. W. (2018). A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 107–116.

Narayan, M. and Pouchet, L.-N. (2012). PolyBench/Fortran 1.0.

NVIDIA (2022). Nvidia hopper architecture in-depth. Technical report, NVIDIA.

NVIDIA (2024). Nvidia blackwell platform: Powering a new era of computing. Technical report, NVIDIA.

Oliphant, T. E. (2006). *A Guide to NumPy*, volume 1. Trelgol Publishing USA.

Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38.

Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. (2020). Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70.

Schreiber, R. (1983). A systolic architecture for singular value decomposition. Technical report, Stanford University, Department of Computer Science.

Smith, J. E. (1984). Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308.

Specification, S. T. (2018). Sifive tilelink specification.

Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356.

UC Berkeley Architecture Research (2022). riscv-benchmarks (hwacha branch).

Verilator (2022). Verilator, Open-source tool for Verilog HDL simulation.

Williams, V. V. (2012). Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 887–898.

Xianyi, Z. and Kroeker, M. (2020). Openblas: An optimized blas library. `https://www.openblas.net/`.

Zhao, J., Korpan, B., Gonzalez, A., and Asanovic, K. (2020). Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*.