SADD MaMA Rocket 🚀
CprE 581 Final Project Report

Steve Frana, Jonathan Tan

December 20, 2024

Abstract

Matrix multiplication is a fundamental operation in numerous computational domains. Many work has been done to implement general matrix multiplication accelerator, such as UCB's Gemmini [7]. However, the accelerator doesn't support sparse matrix multiplication, which is very common in modern workloads like ML. This project proposed a sparse matrix multiplication accelerator inspired by Wang et al.'s SADD architecture [12].

1 Introduction

Matrix multiplication is a fundamental operation in numerous computational domains, for example, machine learning, simulations, and data analytics. Its efficiency and performance are critical for the scalability and speed of these applications. Moreover, many real-world datasets and models are inherently sparse. This highlights the importance of support for sparse matrix multiplication in modern accelerators.

The Berkeley Rocket Core [3] is a 5-stage in-order scalar processor core generator. It was originally developed at UC Berkeley and SiFive and is now maintained by Chips Alliance. In this project, we created a matrix (Ma) multiplication accelerator (MA) for the Rocket core. Due to the short time frame, we decided to implement our sparse matrix multiplication accelerator on the Gemmini DNN accelerator [7], which already has an implemented matrix multiplication unit but no sparse support.

Gemmini uses the Rocket Custom Co-Processor Interface (RoCC) [5] for interfacing with the CPU core. Since our project is based on the Gemmini project, we inherited the Gemmini ISA, allowing easy benchmarking using existing Gemmini c libraries and Scala unit testing support. This approach creates a tightly coupled accelerator that only de-couples from the Rocket core during the execution stage. Moreover, the Chipyard framework gives us access to a high-level flexible RTL coding style, enabling us to test our design under various configurations.

Our sparse matrix multiplication accelerator architecture is heavily inspired by the SADD architecture Want et al. purposed [12]. The architecture focuses on a compression algorithm for compressing sparse matrices and a mesh design that allows for out-of-order processing of compressed sparse matrices.

In our project's final deliverable, we implemented a sparse matrix multiplication accelerator that achieved a 1.2x performance gain for sparse matrix multiplication operations and, along with the existing Gemmini dense matrix multiplication support, achieved a 1.05x performance gain overall.

2 Related Work

Much work has been done investigating the potential wider use, or even commercialization of an open source ISA like RISC-V [2] [8]. Rocket is one such implementation of RISC-V [6]. However, vanilla Rocket is designed to handle general CPU workloads. With the rise of edge image processing and edge AI applications [11], there is a need for an accelerator to perform matrix multiplication fast on the edge. In this section, we will look at the work others have done in this area.

Wang et al. [13] [14] examined a scalable RISC-vector accelerator for DNN applications, SPEED. Compared to the pioneer open-source vector processor Ara, SPEED provides an area efficiency improvement of 2.04× and 1.63× under 16-bit and 8-bit precision conditions, respectively, showing SPEED's significant potential for efficient multi-precision DNN inference.

Perotti et al. [9] build off the RISC-V Vector Extension ISA to optimize the energy efficiency of matrix multiplication operations for low-power embedded systems. This is primarily achieved by minimizing access to the vector register file using a broadcast engine and tile buffers. This also has the effect of producing a more consistent memory access pattern. The MX ISA adds minimal instructions to configure the processor for matrix arithmetic, highlighting a unique approach that minimizes the need for tightly coupled accelerators or dedicated matrix units to support matrix operations.

Jing et al. [10] proposed that RISC-V ISA Extension supports matrix register files, which store N^2 elements per register. This adds another dimension to vector registers and allows execution units or tightly coupled accelerators to simply grab matrices as operands in an instruction, allowing a single instruction to execute N^3 operations. The downside to this approach is that it is costly regarding register file space and adds complexity to load/store units. The cost of holding this data in registers means a high degree of tiling is required in the case of large matrices. This proposed ISA extension would offer an alternative way of handling data for the goals of our project, as registers can be shared between the Rocket Core and RoCC accelerator. However, developers of Rocket have yet to implement the Vector ISA Extension, meaning this would involve extensive modification to the Rocket core itself.

Researchers at UC Berkeley had many experiences creating RoCC accelerators for Rocket/BOOM. Most notably, Genc et.al [7] created the Gemmini accelerator. Gemmini is a DNN accelerator that works to improve convolution operations using a systolic array TPU-like approach. Gemmini implements its own ISA extension for the accelerator, can be configured with a variable number of processing elements, and can operate in a weight-stationary or output-stationary mode. Such configuration options allow it to be set up for matrix multiply operations and could allow us to test our own accelerator's performance against it to validate our results.

Finally, we want to mention Wang et al.'s work [12] on creating a matrix multiplication accelerator with sparse support. Their work includes adding a systolic array accelerator that supports sparsity and dynamic dataflow on top of general matrix multiplication (GEMM) accelerators. The architecture first focuses on a Group-Structure-Maintained Compression (GSMC) compression algorithm for compressing sparse matrices, allowing higher data transfer bandwidth. The architecture also combines different dataflow types and allows dynamic configurations. They achieved 2× performance gain compared to Google's TPU on AlexNet, with little space overhead. This paper heavily inspires the accelerator we create in the project.

3 Methodology

3.1 Chipyard

Chipyard is a framework for designing and evaluating full-system hardware using agile workflows commonly not found in HDL development but in higher-level projects like Java projects. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip [1].

In our project, we elected to use Chipyard as it provides us with the tools to get off the ground with much ease. For example, tools like Spike (a RISC-V simulator), Verilator (a Verilog simulator, described in more detail below), RISC-V build tools, and Scala testing framework (sbt, described in more detail below).

Modules within Chipyard, also called "generators", are written in Chisel. Chisel is a domain-specific language (DSL) based on Scala. Working with Chisel presents several benefits that allow for more agile and modular hardware generator code. More examples will be discussed in subsection 3.4. Chisel, along with the Scala language's testing framework, also allows for JUnit-like testing and easy waveform generation, allowing for software development-like debugging, significantly reducing much of the traditional hardware SDK development overhead. More details will be discussed in section 3.3.

3.2 Verilator

Verilator is an open-source cycle-accurate Verilog simulator. During our implementation, it is used to generate waveform files for debugging. During testing, it is used to provide performance results.

3.3 Scala Unit Tests

A significant advantage in the development of this project was the flexibility offered by the Scala framework. Scala allows us to create robust test benches for our Chisel generators with extreme ease. These test benches not only produced results similar to Java JUnit tests but also generated waveform files, which are invaluable for debugging the HDL code. By leveraging Scala's powerful features, we were able to adopt a software-development-like development cycle, with rapid testing and rapid development, significantly improved development turnaround time compared to traditional hardware development like in CprE 381, CprE 487, and CprE 488.

It is worth noting that this testing framework allows for CI/CD in hardware development, something unthinkable when doing traditional VHDL development using languages like VHDL and frameworks like the Xilinx Vitis/Vivado.

3.4 Chisel Quality of Life Features

Chisel [4], being a higher-level HDL generator language, also provides many quality-of-life features. Such features simplified many of the tedious wire management that exists in traditional HDL coding. For example, AXI transaction management in VHDL is a tedious process, Chisel provided special classes that simplify that and allow for cleaner code [Chisel Docs]. Listing 1 is an example of such class `Decouple`. Any signals defined with `Decouple` automatically have the standard read-valid interface.

```scala
class MeshWithDelays[...](...) extends Module {
  ...

  val io = IO(new Bundle {
    val a = Flipped(Decoupled(A_TYPE))
    val b = Flipped(Decoupled(B_TYPE))
    val d = Flipped(Decoupled(D_TYPE))

    val req = Flipped(Decoupled(new MeshWithDelaysReq(accType, tagType.cloneType, block_size)))

    val resp = Valid(new MeshWithDelaysResp(outputType, meshColumns, tileColumns, block_size, tagType.
    cloneType))

    val tags_in_progress = Output(Vec(tagqlen, tagType))
  })
  ...
}
```

Listing 1: Chisel Decouple example

Other quality-of-life features exist; for example, Chisel's Bundle and Vec allow users to easily define groups of signals [Chisel Docs]. Such features simplify signal creation and generate higher readability code. Many other Chisel quality-of-life features exist and can be found on Chisel's documentation site.

4 Design

4.1 Gemmini Architecture

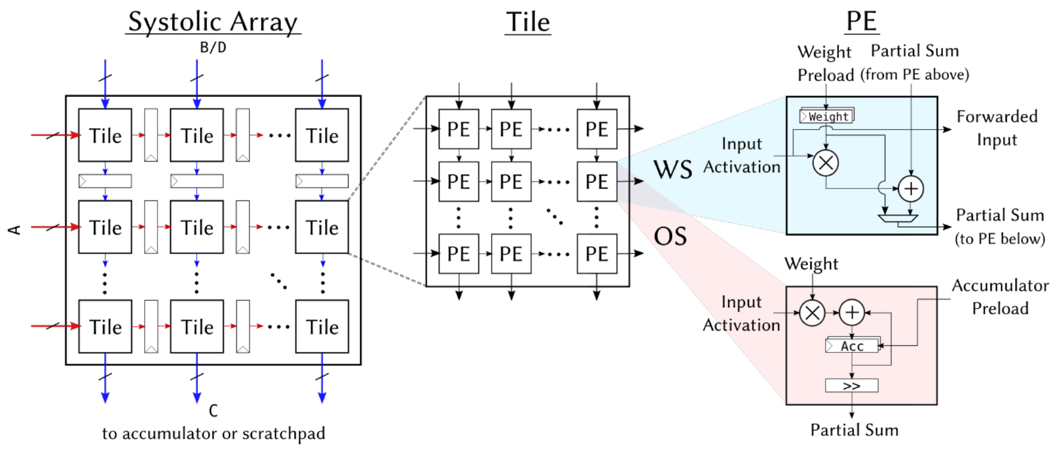


Figure 1: Gemmini's core – mesh and tiles [7]

This sub-section provides a quick summary of the Gemmini architecture as described in [7]. In its core, as shown in figure 1, Gemmini has a systolic array (called mesh), and within a mesh, "tiles" contain PEs that perform the calculation of partial sums. Finally, Gemmini has an accumulator that collects and accumulates results before writing them back to memory. Gemmini is capable of dynamically changing between weight stationary (WS) and input stationary (IS) dataflows. This is reflected in the different configurations in individual PEs as shown in the blue (WS) and pink (IS) boxes in figure 1.

4.2 Matrix compression

As mentioned above, our sparse matrix multiplication accelerator is heavily inspired by the SADD paper [12]. In the paper, the first step to achieving performance improvement is to compress a matrix using the Group-Structure-Maintained Compression (GSMC) algorithm. In the paper, this compression operation is completed in hardware. We accomplished the GSMC compression in software. In our design, the CPU will perform the compression before sending the compressed input matrix to the accelerator. This is mainly to reduce the workload in an attempt to complete the implementation on time.

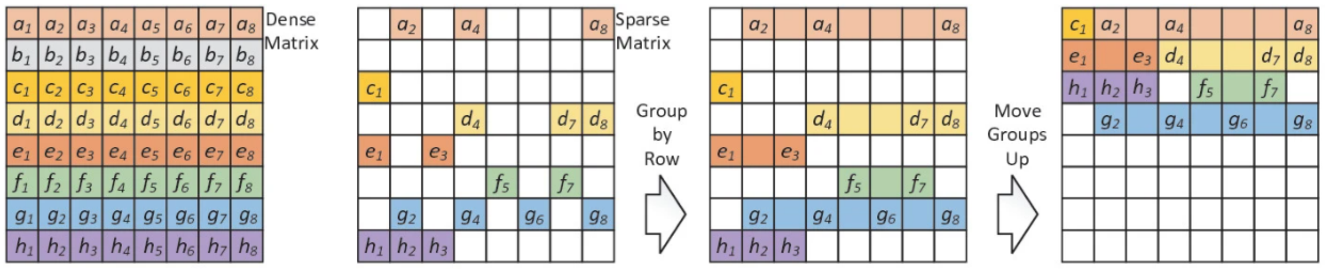


Figure 2: GSMC Compression [12]

Figure 2 shows the GSMC compression algorithm in action. More details are described in [12]. It is worth noting that every row from the dense matrix produces one group. In the compressed matrix, each row can have multiple groups. We used a 1-bit tag to signify end-of-group. The compressed matrix is then passed into the accelerator as shown in figure 3.

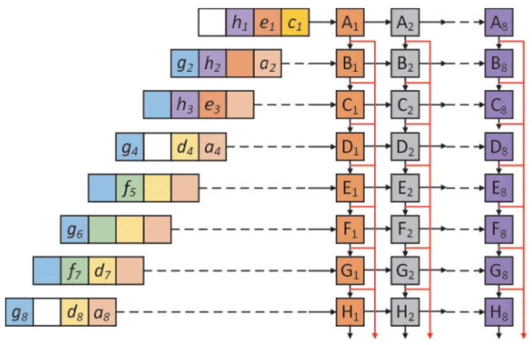
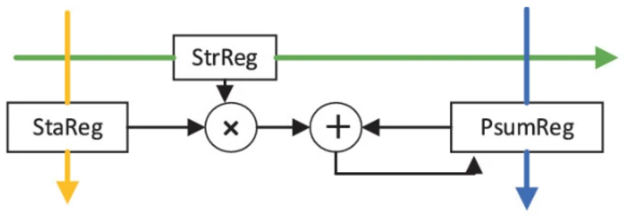


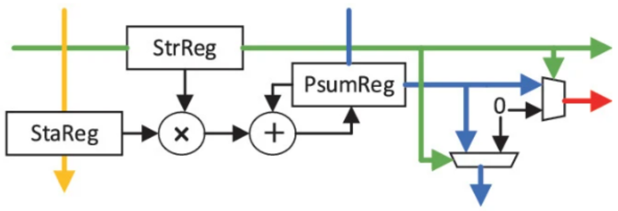
Figure 3: Compressed matrix being passed into the systolic array [12]

4.3 Forwarding Logic

The red line arrows in figure 3 show the forwarding logic. The 1-bit end-of-group tag is used to determine if a group computation is complete and ready to be written back to memory. Details of the forwarding logic are shown in figures 4a and 4b. SADD PEs require additional MUXes (figure 4b) on top of normal, non-SADD, PEs (figure 4a). The 1-bit end-of-group tag directly drives the two MUXes.



(a) Normal PE [12]



(b) SADD PE [12]

Figure 4: Comparison of PE methods

To implement forwarding on top of the gemmini architecture, two muxes were inserted between tiles and pipeline registers to either forward partial sums to a queue and pass zeros down a column or propagate partial sums down the column as normal, depending on the tag of the input. Gemmini's scratchpad and accumulator memory banks are set up to read and write rows of data at a time to reduce complexity in the design. To allow writebacks to take place as normal, forwarded rows are tagged with the current matrix multiply instruction, buffered, and written to the scratchpad after the systolic array completes. This means that output rows of compressed matrix multiplies are written to the scratchpad or accumulator memory out-of-order.

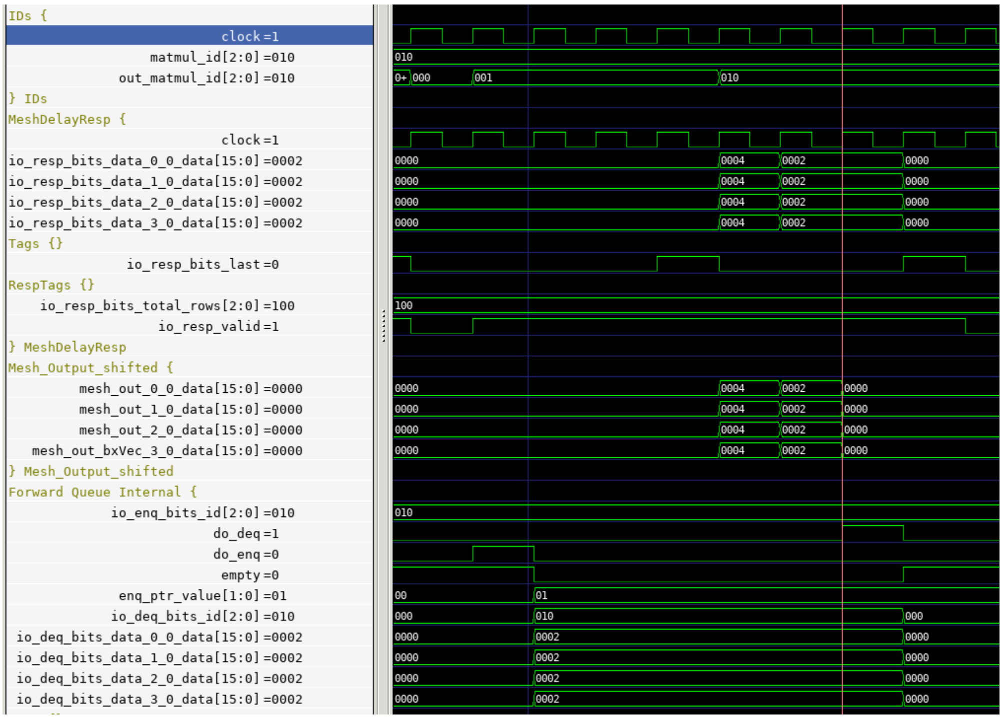

```
IDs {
                              clock =1
                     matmul_id[2:0] =010
                 out_matmul_id[2:0] =010
} IDs
MeshDelayResp {
                              clock =1
io_resp_bits_data_0_0_data[15:0] =0002
io_resp_bits_data_1_0_data[15:0] =0002
io_resp_bits_data_2_0_data[15:0] =0002
io_resp_bits_data_3_0_data[15:0] =0002
Tags {}
                  io_resp_bits_last =0
RespTags {}
    io_resp_bits_total_rows[2:0] =100
                      io_resp_valid =1
} MeshDelayResp
Mesh_Output_shifted {
         mesh_out_0_0_data[15:0] =0000
         mesh_out_1_0_data[15:0] =0000
         mesh_out_2_0_data[15:0] =0000
   mesh_out_bxVec_3_0_data[15:0] =0000
} Mesh_Output_shifted
Forward Queue Internal {
                io_enq_bits_id[2:0] =010
                             do_deq =1
                             do_enq =0
                              empty =0
                 enq_ptr_value[1:0] =01
                io_deq_bits_id[2:0] =010
 io_deq_bits_data_0_0_data[15:0] =0002
 io_deq_bits_data_1_0_data[15:0] =0002
 io_deq_bits_data_2_0_data[15:0] =0002
 io_deq_bits_data_3_0_data[15:0] =0002
```

Figure 5: Mesh Response with Forwarding

Gemmini's execution controller issues requests to a module called MeshWithDelays, this module is responsible for sending control signals to the systolic array, keeping track of matrix multiply writeback addresses, and issuing a response when the computation is complete. (figure 5) shows how forwarded data is inserted into a queue and written into the response once the systolic array has finished writing.

4.4 Metadata Handling

Gemmini is a type-generic accelerator, which allows us to define our own data types and arithmetic operations to further explore and interact with our design. We created our own "SparseInt" datatype to hold some meta information that persists alongside data throughout the computation (figure 6). Currently, we use a 16-bit integer along with 16-bit metadata. The 1-bit end-of-group tag mentioned above is part of the 16-bit metadata. In addition to the 1-bit end-of-group tag, the metadata also consists of tags signaling the row and column or the element from the original dense matrix. These tags allow us to manage forwarding from software and understand the ordering of output rows from a compressed matrix multiply.

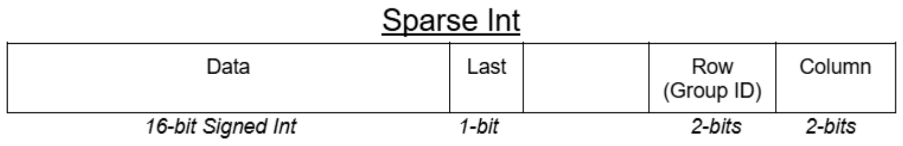


Figure 6: SparseInt with metadata

We believe that a smarter compression algorithm and improved management of forwarded data can completely eliminate the need for row and col tagging, allowing us to reduce the metadata overhead from 16 bits down to 1 bit, which we shall discuss further in section 5: Results.

5 SADD MaMA Evaluation

5.1 Evaluation Methodology

To collect performance results, we created two sets of test code, one with a dense matrix multiplication operation targeting a single-core Rocket CPU and a CPU with our Gemmini+MaMA accelerator and one with a sparse matrix multiplication operation targeting a single-core Rocket CPU and a CPU with our Gemmini+MaMA accelerator.

We define ”sparse” as any matrix with $> 55\%$ sparsity, which is a reasonable assumption as per [15] [12]. However, for our benchmark, we selected 75% sparsity as a similar benchmark with 75% sparsity used in [12]. The dense matrix multiplication benchmark we used is a matrix multiplication test part of the gemmini-rocc-tests [GitHub]. We modified the benchmark a little to arrive at our sparse matrix multiplication benchmark with sparsity of 75%. We used Verilator as the simulation tool.

5.2 Results

Table 2 shows the results we achieved by running our benchmarks. As we can see, the single Rocket core system, the baseline system, achieved a CPI of 1.5. Performance stays constant between dense and sparse as the CPU core is indiscriminate against data values.

Configuration	CPI
Rocket Core Dense	1.5
Rocket Core Sparse	1.5
Rocket+Gemmini Dense	1.6
Rocket+Gemmini Sparse	1.6
Rocket+SADD Dense	1.6
Rocket+SADD Sparse	1.6

Table 1: Performance Results

Profiling a single Rocket core in addition to a single Gemmini accelerator, we observed a CPI of 1.6 for both dense and sparse. The indifference in CPI between dense and sparse isn’t surprising, as the reason is the same as in the single Rocket core case. On the other hand, the decrease in performance with the Gemmini accelerator is surprising. This can likely be attributed to the difference in source code and the number of instructions issued to the CPU. For example, to compute a matrix multiply on the core requires many instructions that can be operated on in parallel, improving CPU utilization. When running Gemmini, the core issues a few instructions to the accelerator, each representing many operations but taking longer to execute.

Configuration	CPI
Rocket+SADD Dense (4x4 Input Matrix)	2.1
Rocket+SADD Sparse (2x4 Input Matrix)	2.0

Table 2: Performance Results

In addition, our workload requires several setup instructions and potentially misses small performance improvements to the matrix multiply operation between sparse and dense matrices. Another test program, shown in Table 2, profiles the CPI in a finer granularity and showed a slight improvement when looking at sparse matrices that could be compressed to half the width of dense matrices. We found software tests to be limited as they only represent the CPI from the perspective of the processor and not the accelerator. Further investigation of performance differences using Verilator would require the insertion of hardware performance counters in Gemmini or a larger-scale test dominated by Gemmini instructions. We chose to analyze waveforms produced by unit tests to see how performance might scale.

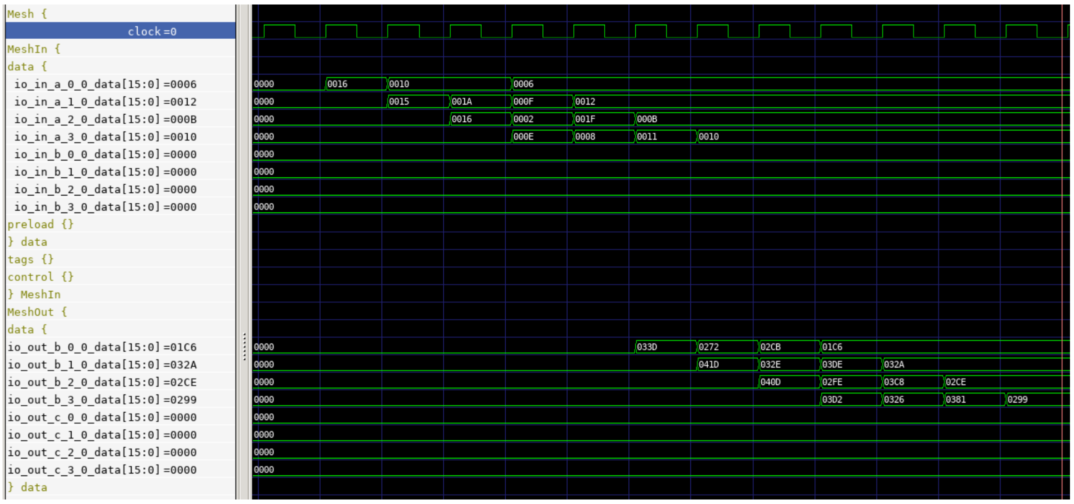

```
Mesh {
clock=0
MeshIn {
data {
 io_in_a_0_0_data[15:0] =0006
 io_in_a_1_0_data[15:0] =0012
 io_in_a_2_0_data[15:0] =000B
 io_in_a_3_0_data[15:0] =0010
 io_in_b_0_0_data[15:0] =0000
 io_in_b_1_0_data[15:0] =0000
 io_in_b_2_0_data[15:0] =0000
 io_in_b_3_0_data[15:0] =0000
preload {}
} data
tags {}
control {}
} MeshIn
MeshOut {
data {
io_out_b_0_0_data[15:0] =01C6
io_out_b_1_0_data[15:0] =032A
io_out_b_2_0_data[15:0] =02CE
io_out_b_3_0_data[15:0] =0299
io_out_c_0_0_data[15:0] =0000
io_out_c_1_0_data[15:0] =0000
io_out_c_2_0_data[15:0] =0000
io_out_c_3_0_data[15:0] =0000
} data
```

Figure 7: Dense Matrix Multiply on a Mesh with 4x4 PEs

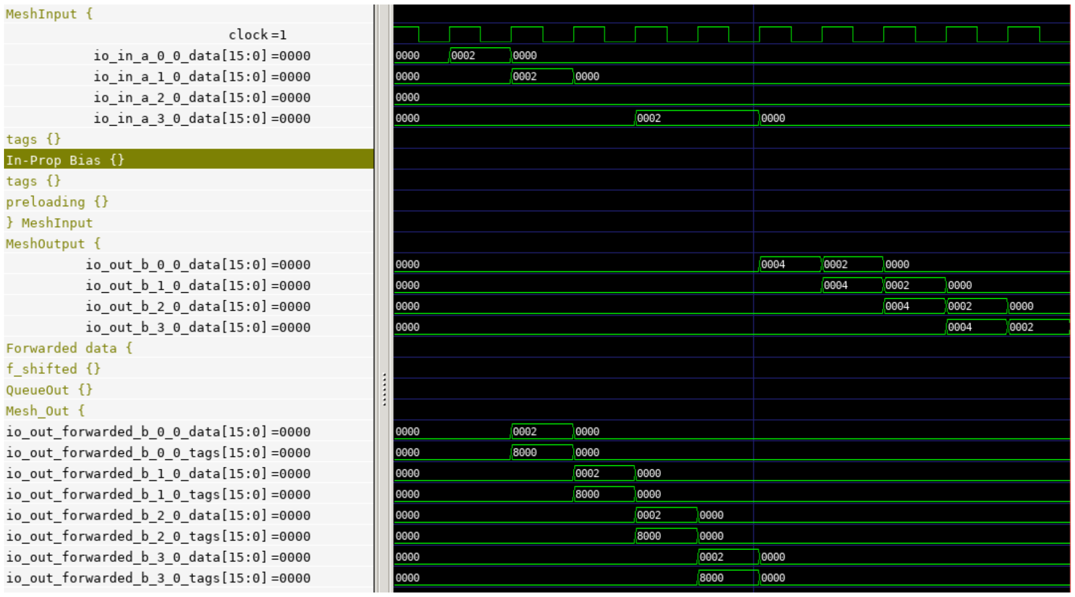
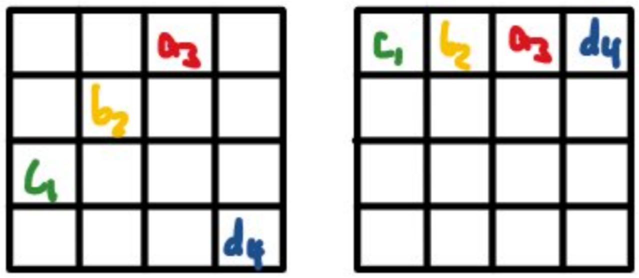
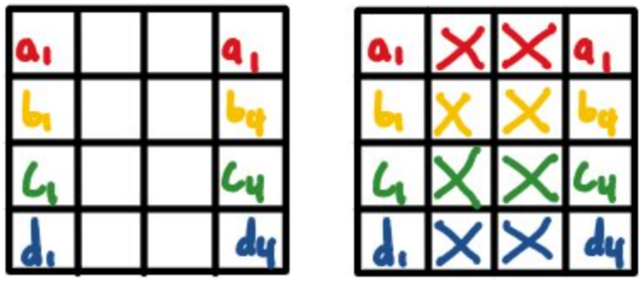

```
MeshInput {
clock=1
io_in_a_0_0_data[15:0] =0000
io_in_a_1_0_data[15:0] =0000
io_in_a_2_0_data[15:0] =0000
io_in_a_3_0_data[15:0] =0000
tags {}
In-Prop Bias {}
tags {}
preloading {}
} MeshInput
MeshOutput {
io_out_b_0_0_data[15:0] =0000
io_out_b_1_0_data[15:0] =0000
io_out_b_2_0_data[15:0] =0000
io_out_b_3_0_data[15:0] =0000
Forwarded data {
f_shifted {}
QueueOut {}
Mesh_Out {
io_out_forwarded_b_0_0_data[15:0] =0000
io_out_forwarded_b_0_0_tags[15:0] =0000
io_out_forwarded_b_1_0_data[15:0] =0000
io_out_forwarded_b_1_0_tags[15:0] =0000
io_out_forwarded_b_2_0_data[15:0] =0000
io_out_forwarded_b_2_0_tags[15:0] =0000
io_out_forwarded_b_3_0_data[15:0] =0000
io_out_forwarded_b_3_0_tags[15:0] =0000
```

Figure 8: Sparse Matrix Multiply on a Mesh with 4x4 PEs

(figure 7) shows a dense matrix multiply operation for our testing setup with Gemmini configured for a 4x4 systolic array of tiles with one PE each. For this operation, 12 cycles of computation are required. (figure 8) shows a sparse matrix multiply operation on the same setup. In this scenario, the input matrix was compressed to 2x4, and the first element into the systolic array was tagged to be forwarded. For this operation, 10 cycles of computation are required. This shows a speedup of 1.2 for sparse input matrices representative of our workload when considering only the matrix multiply operation.



(a) Best-Case Compressed Matrix



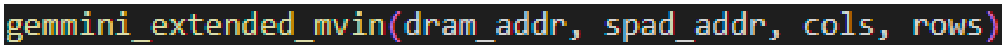
(b) Worst-Case Compressed Matrix

Figure 9: Best/Worst Case Matrix Example

An ideal matrix can be compressed to 1x4, where each element of the first row is representative of its own row of outputs; this is shown in (figure 9a). However, sparse matrices may not compress ideally as shown in (figure 9b), in this case, no speedup would be observed. A better speedup can be observed for a systolic array consisting of more PEs. For an NxN systolic array, we can quantify the number of cycles required to compute a sparse matrix as:

$$\text{Min Cycles} : 2N + 1$$
$$\text{Max Cycles} : 3N$$
$$\text{Average Cycles} : 2N + C$$

Here, N is the dimensionality of the systolic array, and C is the number of rows an input matrix is compressed to on average. This shows a greater potential computation speedup when considering large systolic arrays; however, as the number of PEs grows, so will the wire length on each forwarding path. A deeper investigation into how this would affect the maximum frequency is required to find an optimal design.



```
gemmini_extended_mvin(dram_addr, spad_addr, cols, rows)
```

Figure 10: Gemmini ISA instruction specifying rows and columns of a matrix loaded into scratchpad

In the SADD paper [12], Wang et al. discussed the bottleneck of the system lies with systolic array stationary value replacement and data movement. This is where our SADD modification for Gemmini comes in. We could potentially improve the bandwidth-bound by using pre-compressed matrices and Gemmini ISA instructions that load compressed matrices with exact dimensions as shown in figure 10. However, because learning about the framework took a good chunk of our time, as mentioned above, our initial solution generated significant metadata overhead, effectively evening out the bandwidth for compressed matrices. Although our final system isn't fully developed, it is safe to say little performance gain could be achieved using this approach. However, given more time, it is possible to reduce metadata overhead from 100% to ≈ 10% by using the 1-bit tag to signify the last element in a group instead of 16-bits.

6 Future Work/Conclusion

6.1 Future Work

While our partial implementation and analysis of the SADD MaMA demonstrated the feasibility of integrating the SADD-inspired architecture with Gemmini, there are still some improvements that can be done. The most obvious task is to fully implement the SADD architecture into the Gemmini, addressing the current metadata overhead by optimizing the tagging system. Transitioning from a 16-bit to a 1-bit tag, as mentioned above, could significantly reduce overhead and improve performance.

Another consequence of the current implementation is the software burden of compressing matrices, managing data forwarding, and reordering output data. Additional control can be utilized to ensure output rows are reorganized in hardware. One interesting solution to compressing matrices would be in-memory processing. This may allow us to increase the bandwidth-bound while reducing the software burden of performing the compression manually. When considering DNN workloads, it could be advantageous to use an input-stationary dataflow and compress the weights of a model during training. A training algorithm with knowledge of the GSMC algorithm could result in increased probably of producing ideally compressed weight matrices. This type of workload would be ideal for our design.

Additionally, extending our accelerator to support a broader range of sparsity levels and matrix sizes would also be a good improvement. Beyond simulation, deploying our design on an actual FPGA platform will also allow us to better analyze space and energy cost/performance, offering insights into the practicality of our SADD MaMA for a RISC-V core in the real world.

6.2 Conclusion

In this project, we partially integrated a sparse matrix multiplication accelerator inspired by the SADD architecture into the Gemmini accelerator. Our development leveraged the Chipyard and Gemmini frameworks and tools, enabling rapid development and initial performance evaluation. Although preliminary results indicated a bandwidth-bound bottleneck when utilizing the Gemmini accelerator, our work in this project laid a solid foundation for future enhancements. By addressing metadata overhead and completing the full implementation, we expect significant performance gains, especially for applications with high sparsity. Ultimately, our work contributes to the ongoing efforts to optimize matrix operations in modern accelerators, paving the way for more efficient and scalable solutions in machine learning and data analytics.

References

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[2] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, Apr 2016.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Risc-V Boom. BOOM RoCC Interface. https://github.com/ucb-bar/riscv-benchmarks/tree/master/mt-matmul.

[6] Christopher Celio, David A Patterson, and Krste Asanovic. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.

[7] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[8] Samuel Greengard. Will risc-v revolutionize computing? *Communications of the ACM*, 63(5):30–32, 2020.

[9] Matteo Perotti, Yichao Zhang, Matheus Cavalcante, Enis Mustafa, and Luca Benini. Mx: Enhancing risc-v's vector isa for ultra-low overhead, energy-efficient matrix multiplication. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[10] Risc-V Qiu Jing. XuanTie Matrix Extention Spec.

[11] Raghubir Singh and Sukhpal Singh Gill. Edge ai: a survey. *Internet of Things and Cyber-Physical Systems*, 3:71–92, 2023.

[12] Bo Wang, Sheng Ma, Zhong Liu, Libo Huang, Yuan Yuan, and Yi Dai. Sadd: A novel systolic array accelerator with dynamic dataflow for sparse gemm in deep learning. In *IFIP International Conference on Network and Parallel Computing*, pages 42–53. Springer, 2022.

[13] Chuanning Wang, Chao Fang, Xiao Wu, Zhongfeng Wang, and Jun Lin. A scalable risc-v vector processor enabling efficient multi-precision dnn inference. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 2024.

[14] Chuanning Wang, Chao Fang, Xiao Wu, Zhongfeng Wang, and Jun Lin. Speed: A scalable risc-v vector processor enabling efficient multiprecision dnn inference. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14, 2024.

[15] Jianlei Yang, Wenzhi Fu, Xingzhou Cheng, Xucheng Ye, Pengcheng Dai, and Weisheng Zhao. S2 engine: A novel systolic architecture for sparse convolutional neural networks. *IEEE Transactions on Computers*, 71(6):1440–1452, 2022.

Footnote

GitHub repo:

- Chipyard: link
- Gemmini: link
- Gemmini benchmarks: link