Project Report: GeMM Accelerator Design

CPAEP

Academic year 2025–2026

Deadline: Sunday December 7th 19:00

Student name(s): Simon Forceville

Student ID(s): r0800798

1 Q1 – Architecture drawing and description

1.1 Block diagram and array organization

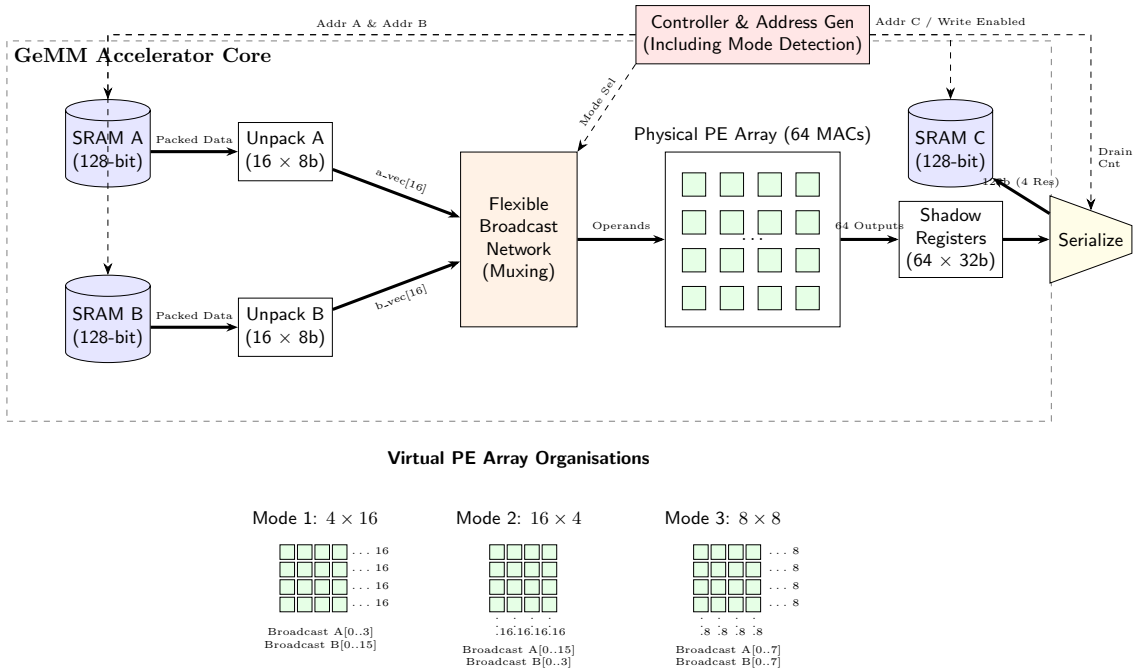

Figure 1: High-level block diagram of the GeMM accelerator.

1.2 Architecture trade-offs and complexity

The primary performance objective is to minimize the average normalized latency across the three cases. The most important metric to get this latency down is the overall MAC utilization. Thus I came up with and compared different MAC array organizations by utilization. A fixed-grid systolic (2D) array (e.g. 8×8) cannot adapt to case 1 (4×16) or case 2 (16×4) and will lead to idle PEs and thus higher latency. In this case 4 rows will be mapped to 8 rows, resulting in 50% of the hardware being idle. To maximize MAC utilization, the design requires a core that dynamically adapts to the shape of the input matrices.

I chose a flexible broadcast network that distributes the input data across the MACs, and with output stationarity so that a single MAC can be mapped to each output. In every case the output contains a multiple of 64 elements, resulting in a 64-MAC array that is fully utilized at all times.

This broadcasting network is essentially a large set of multiplexers that changes connections based on the shape of the matrix calculation (*mode* in block diagram). By reusing data (broadcasting), it is able to keep all 64-MACs busy by a single fetch of 16 A-elements and 16 B-elements. SRAM A and SRAM B were widened to 128-bits to allow this.

SRAM C was also widened to reduce the bottleneck it imposed on the system. Although this costs extra area and increases the routing complexity, the inner dimension (K) of A and B is large enough so that overall the impact on performance is well worth the cost. According to Amdahl's law, as the bus is widened further, the write time becomes such a small fraction of the total execution time that further widening yields negligible but very costly gains.

The broadcasting network obviously costs a significant amount of extra area, both in the controller and in the multiplexers themselves. To improve on the bottleneck SRAM C imposed in case 3, shadow registers were added. This pipelining resulted in an additional considerable increase in complexity in the controller and extra registers. In the modified test bench `tb_one_mac_gemm.sv`, it resulted in an additional large speed-up from 50% to 70% over the non-pipelined (but widened) SRAM C.

2 Q2 – Data flow, data layout and address generation

2.1 Dataflow for the three workloads

The full pseudo code is detailed in paragraph 2.3. Instead, I will detail the data flow for each of the three cases. Regardless of the case, every SRAM A and B read supplies the broadcasting network with a vector of 16 elements of A respectively B. However depending on the output dimensions of matrix C, only 4 of these elements will be used at the minimum.

- **Case 1 (4×16):** The array is organized in 4 rows of 16 MACs. Since the elements of vector A map to the rows of the array, in this case, only 4 of the 16 elements of vector A will be used per data fetch. Similarly, the element of vector B map to the rows of the MAC array, so every element of vector B is used.
- **Case 2 (16×4):** In this case, the array is organized in 16 rows of 4 MACs. The resulting dataflow is identical to case 1, but the input vectors are switched. Only 4 elements of every fetch of vector B are used, every vector element of A is used.
- **Case 3 (32×32):** Here, the array becomes a square array of 8×8 MACs. Since we lack the resources to map every output to a single MAC, we instead calculate the resulting matrix in 8×8 tiles. This results in 16 tiles done sequentially. For each of these 16 tiles, the entire inner dimension is run first, so the same principles apply for the data inflow. Because the array is organized in 8×8, half of vector A and half of vector B are used.

The outflow of data from the MAC array to SRAM C is similar in all three cases. As soon as the output array is done calculating a single result, whether it is the full matrix C in case 1 and 2, or a tile of matrix C in case 3, the accumulation registers push the data to the shadow registers. This is a set of 64 32-bit registers, one for each MAC. These temporarily buffer the output from the MAC array, so the limited SRAM C input speed is less of a bottleneck to the system. SRAM C takes in 4 32-bit results at a time, with a drain counter signal coming from the controller directing the shadow registers which values to output at a time. These values are packed in multiples of 4, with the controller computing the corresponding addresses.

2.2 Data layout of A, B and C in memory

To keep the 64-MAC array fully utilized across all workload shapes without stalling, the data width of the three SRAMs were widened to 128-bits. This allows the system to fetch a vector (in case of SRAM A and B) of 16 8-bit operands in a single clock cycle.

The data layout is a result of the requirements of the broadcast network.

- **Matrix A (column-packed)** Each column of A is packed into a single 128-bit address. This fits the architecture because in the three cases, the broadcast network always requires a column of A to distribute across the rows of the MAC array.
- **Matrix B (row-packed)** Each row of B occupies one address, such that in every cycle of the K-loop, we can fetch the necessary row of B and column of A to feed all 64-MACs.
- **Matrix C (result packing)** The 32-bit outputs are packed in pairs of four into a 128-bit word. As detailed in the dataflow section, the shadow registers buffer the 64 results and a serializer writes them over 16 cycles to the C SRAM.

The address generation units (AGUs) support this layout by doing the address generation block-based rather than element-wise. High-level tile coordinates from the controller (e.g., M_{count}, N_{count}) are translated into packed memory indices.
For Matrix A, the address is calculated by determining the 16-row block index from the current tile position and adding the K offset ($Block_i \times K + k$). Similarly for B, the AGU calculates the address by locating the correct 16-column block row. This ensures that as the controller iterates through tiles and the K (inner) dimension, the memories always supply the correctly packed vector to the broadcast network.

2.3 Loop structure and mapping to addresses

The outer process loops are sequential and handle tiling, meaning they iterate over the output matrix C one tile at a time. M_{tiles} is the number of vertical tiles, N_{tiles} is the number of horizontal tiles. In case 1 and 2, the loop structure will only run once because there is only one tile in the output. In case 3, M_{tiles} and N_{tiles} is 4 to cover the 32×32 output grid with tiles of size 8×8. The innermost parallel loops do the execution of the 64-MAC array. The element $A_{vec}[r]$ is sent to all columns in row r, while the element $B_{vec}[c]$ is sent to all rows in column c.

The virtual array dimensions (H_{tile}, W_{tile}) are decided by the workload mode (4×16, 16×4, or 8×8). In all three cases, the virtual array amounts to 64 MACs.

1: **for** $t_m = 0$ to $M_{tiles} - 1$ **do** ▷ *tiling (rows)*
2: **for** $t_n = 0$ to $N_{tiles} - 1$ **do** ▷ *tiling (cols)*
3: **Reset** Accumulators $C_{tile}[0..(H_{tile} - 1)][0..(W_{tile} - 1)]$
4: **for** $k = 0$ to $K_{size} - 1$ **do** ▷ *inner dimension: input depth*
5: **fetch** $A_{vec}[0..15]$ and $B_{vec}[0..15]$
6: **parfor** $r = 0$ to $H_{tile} - 1$ **do** ▷ *parallel: array rows*
7: **parfor** $c = 0$ to $W_{tile} - 1$ **do** ▷ *parallel: array cols*
8: $C_{tile}[r][c] \leftarrow C_{tile}[r][c] + (A_{vec}[r] \times B_{vec}[c])$
9: **end parfor**
10: **end parfor**
11: **end for**
12: **end for**
13: **end for**

The address generation logic for SRAM A transforms the loop indices into a linear memory

address as follows:

$$\text{Addr}_A(t_m, k) = \underbrace{\left\lfloor \frac{t_m \times H_{tile}}{16} \right\rfloor}_{\text{Vertical Block Index}} \times K_{total} + \underbrace{k}_{\text{Inner Loop Index}} \tag{1}$$

Where:

- t_m is the current vertical tile loop index.
- H_{tile} is the height of the tile in the current mode (e.g., 4, 8, or 16).
- k is the iterator of the inner compute loop (compute depth) (0 to $K-1$).
- The division by 16 accounts for the vertical packing of 16 `int8` elements into one 128-bit memory word.

This layout and loop structure is a good match for the array's dataflow because in every case, the MAC array can execute a full inner cycle (meaning a full compute for index k) with a single data read. Every read, a full vertical vector of 16 elements in the case of the SRAM A. This matches perfectly with the operational need of mode 16×4 to have an operand 16 rows of the array. In the 4×16 or 8×8 modes, three fourth or half of the elements of the A_{vec} are unused. An analogous matching occurs between mode 4×16 and a data read of SRAM B. This ensures a stall is never required to start a computation.

3 Q3 – Utilization, bandwidth and performance

3.1 Latency and MAC utilization

The measured latency and spatial utilization for the three workloads are summarized below.

Workload	Matrix Dim	Latency (C)	Spatial Util. (Compute)	Spatial Util. (Overall)
Case 1	$4 \times 64 \times 16$	82	100%	78%
Case 2	$16 \times 64 \times 4$	82	100%	78%
Case 3	$32 \times 32 \times 32$	545	100%	94%

Table 1: Measured performance metrics for the Flexible Broadcast Accelerator.

Note. Example Overall Spatial Utilization Calculation.
Case 3: $32^2/64$MACs = 16tiles, resulting in 16 tiles $\times$32 compute = 512 total compute cycles so $512/545 = 0.94$.

PEs are idle primarily during the **Drain Phase** (output serialization) and **initial pipeline filling**. This is because the SRAM C interface width (128-bit) allows writing only 4 results per cycle. Writing back the 64 PE results thus requires 16 serialization cycles.

In single-tile workloads (cases 1 & 2), the array must wait 16 cycles for the drain to complete before finishing, leading to a 16 cycle drain phase overhead (64 compute + 16 drain + overhead = 82). An additional two cycles is lost in controller overhead, waiting for counters and memory. In multi-tile workloads (case 3), the drain phase of tile N overlaps with the compute phase of tile N + 1. This reduces the total drain phase overhead. The remaining overhead comes from the final tile's drain and the controller.

The above numbers were measured using the SystemVerilog test bench `tb_gemm_performance.sv`. A cycle counter inside the `start_and_wait_gemm` task increments on every positive clock edge while the `done` signal remains low.

3.2 Memory bandwidth usage

**Note:* Output bandwidth is only fully used during the drain phase (16 cycles per tile).

Memory Port	In/Out	Word Size	Words/cycle	Bandwidth (b/cycle)
SRAM A Read Port	In	128-bits	1 (16 int8)	128
SRAM B Read Port	In	128-bits	1 (16 int8)	128
SRAM C Write Port	Out	128-bits	1 (4 int32)*	128
Total			**2 In / 1 Out**	**256 In / 128 Out**

Table 2: Memory bandwidth quantification at steady state.

3.3 Critical path and possible improvements

The critical path is most likely the combinational path starting from the SRAM read ports through the broadcast network and ending at the accumulator registers inside the (clocked) MACs. This is because of the serialization of three components with high delay in a single clock cycle.

1. The data from SRAM A and B must pass through the **broadcast logic**. This is complex multiplexing logic (dependent on `current_mode` and tile counters) with a high-fanout, as the input vectors are broadcasted to the 64 MACs. This increases the capacity of the component and thus the delay.
2. After arriving at the **MAC**s, the operands undergo an 8×8-bit multiplication followed by a 32-bit addition with the accumulator register value. Traditionally, a multiplication will cost a lot of time and area, and there is no exception here.
3. The result must stabilize before the setup time of the **accumulator flip-flops**.

There are a few other paths present in this simple system, which are clearly not critical.

- **Pipeline handoff** The accumulation registers of the MAC array hand their results over to the shadow registers. Since this is a direct register to register path without any logic, it is extremely fast.
- **Drain** The shadow registers pass their data 4 results at a time to the SRAM C. Again, no logic is in the way slowing down this path.
- **Controller paths** The controller is sending its control signals to all components in the circuit. The controller signal pathways do involve logic, but since it is mainly counter increments, this doesn't compare to the complexity of the above combinational path, and thus is not critical.
- **Hidden paths** Technically, there is also an internal path in the MAC array itself, looping accumulation register outputs to the adders and back into the accumulation registers. Also the address generation logic from the controller to the SRAM address ports can be considered a separate loop. Though these both contain logic, we can safely assume they are not critical.

To shorten the critical path, I would introduce a pipeline stage between the broadcast network and the PE inputs. This would require the implementation of pipeline registers to capture the resolved `op_a` and `op_b` signals after the broadcast network but before the MAC logic. This splits the critical path into two balanced stages.

Stage 1: Data fetch $\rightarrow$ Broadcast network $\rightarrow$ Pipeline register

Stage 2: Pipeline register $\rightarrow$ MAC (multiplier, adder and accumulation register).

Cost of reducing critical path

- **Area** This would require adding registers for the operands at every MAC. For 64 MACs with two 8-bit inputs, this adds $64 \times 16 = 1024$ flip-flops to the design. This is a moderate amount.

- **Latency** The latency increases by just one cycle (the initial pipeline fill time). Especially for large matrices, this overhead is negligible.
- **Control Complexity** The `gemm_controller` would need to delay the `compute_active` (valid) signals by one cycle to align with the data arriving at the MAC units. The drain logic stays identical.

The additional cost seems reasonable, depending on the performance requirement. If the system is already performing at the required speed, any additional cost is unnecessary.

4 Q4 – Roofline model of your accelerator

4.1 Roofline construction

The compute roof is calculated based on the number of MACs times the operations per MAC per cycle. The design employs 64 MAC units. Each MAC performs 2 operations (multiplication and addition) per cycle, resulting in: 128 Ops/cycle.

The memory roof follows from the widened 128-bit (16-byte) data bus for both SRAM A and SRAM B. Since the SRAMs allow simultaneous reads from both memory instances (A and B), the memory roof allows 32 Bytes/cycle

The arithmetic intensity required to saturate the available compute is thus: $128/32 = \mathbf{4.0}$ Ops/Byte.

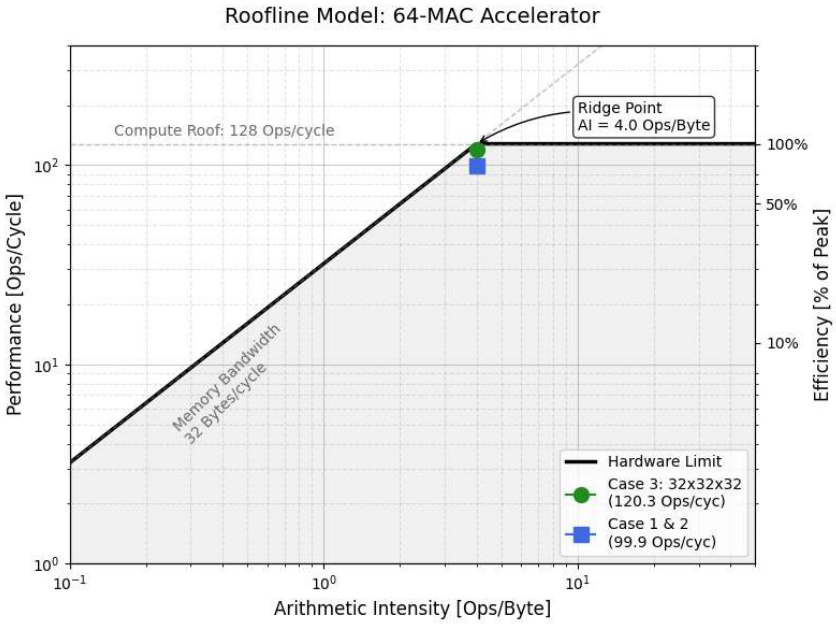

Figure 2: Roofline model

4.2 Arithmetic intensity and operating points

The data packing strategy ensures that for every cycle, exactly one 128-bit word from A and one 128-bit word from B feeds the array, regardless of the aspect ratio.

- **Case 1 ($4 \times 64 \times 16$):** Single pass ($M = 4, N = 16$ fits the 64 MACs).
  - **Ops:** 128 Ops/cycle (Full utilization).
  - **Bandwidth:** 16 Rows of A (12 padding) and 16 cols of B per step.
  - **Intensity:** $\frac{128 \text{ Ops}}{32 \text{ Bytes}} = 4.0$ Ops/Byte.
  - **Compute Bound**.

- **Case 2 ($16 \times 64 \times 4$):** Single pass ($M = 16, N = 4$ fits the 64 MACs).
  - **Intensity:** $\frac{128 \text{ Ops}}{32 \text{ Bytes}} = 4.0$ Ops/Byte.
  - **Compute Bound**.
- **Case 3 ($32 \times 32 \times 32$):** Processed in 8×8 tiles.
  - **Intensity:** $\frac{128 \text{ Ops}}{32 \text{ Bytes}} = 4.0$ Ops/Byte.
  - **Compute Bound**.

All three workloads sit right at the ridge point of the roofline. This means the architecture is heavily optimized and well balanced for all three cases.

4.3 Comparison with measured latency

Below is the comparison of the expectations with the latencies from the `tb_gemm_performance` simulation.

Case	Workload	Ideal (cycles)	Measured (cycles)	Throughput	Efficiency
1	$4 \times 64 \times 16$	64	82	99.90 Ops/Cycle	78.0%
2	$16 \times 64 \times 4$	64	82	99.90 Ops/Cycle	78.0%
3	$32 \times 32 \times 32$	512	545	120.25 Ops/Cycle	93.9%

Table 3: Comparison of theoretical vs. measured performance

- **Case 1 ($4 \times 64 \times 16$):** With latency of **82 cycles**, only 18 cycles above the theoretical minimum of 64. This result confirms the broadcast network improves on a systolic array. A fixed 8×8 systolic array would suffer from 50% utilization here (taking ≈128 cycles), but this architecture keeps all 64 MACs active. The 18-cycle overhead is due to the initial pipeline fill and the drain phase, which are significant in case 1 and 2 because the total workload is small.
- **Case 2 ($16 \times 64 \times 4$):** Identical to case 1, the workload finished in **82 cycles**. Because of the symmetry in the system in terms of address generation and interconnect logic, the same latency as in case 1.
- **Case 3 ($32 \times 32 \times 32$):** The accelerator achieved **120.25 Ops/Cycle**, which is **94%** of the theoretical maximum (128 Ops/Cycle). As the workload increases, the fixed overhead of pipeline filling and draining is split over a longer compute time. This essentially proves the design scales well to larger loads, and approaches the roofline limit.

5 Q5 – Up-/down-scaling scenarios

5.1 Scaling up to 256 PEs

To scale to 256 PEs, I would expand the virtual PE array to a 16×16 physical array.

- In terms of bandwidth, the broadcast network needs to be expanded to support 16 8-bit length vectors. To reach 100% utilization, the datawidth would need to double to 256 bits (32 bytes) to fetch entire rows/columns (16×8-bit) in a single cycle. Increasing the SRAM input and/or output width to 256 bits or even 512 bits is an immensely costly and complex operation due to the need for the far more complex interconnect and mux system. Possible additional 128 bit SRAMs would be preferred over widening the existing memory busses.
- Case 3 throughput would increase by 4× as the system is now able to process tile size up to 16×16. However, for cases 1 & 2, utilization would drop to 25%.

- As for raw compute performance, the compute roof moves up by 4×. This results in the ridge point shifting to the right, resulting in an even higher bandwidth to saturate the array.

5.2 Scaling under reduced bandwidth

The current 64 MAC design would become very memory-bound, resulting in many data stalls and far reduced performance.

The topology could be optimized by implementing a buffer in front of the broadcasting network, similar to the shadow registers behind the PE array. This decouples memory speed from compute, by being able to prefetch data. Alternatively, the array could be reduced to 32 or even 16 PEs. Without the available bandwidth, spending that much area on additional MACs doesn't increase performance and thus is useless. Because throughput drops linearly with bandwidth, utilization of the 64-MAC array would fall to $< 50\%$ without additional buffering. The slanted memory-bound ceiling drops by 2× and 4× respectively. The ridge point shifts to the right. All workloads (which are currently compute bound) would shift into the memory-bound region.

6 Q6 – Test coverage and verification quality

6.1 Executed tests

The following suite of unit and top-level tests is used.

- **Unit tests:**
  - `tb_mac_pe`: To test the correctness of the MAC PE unit and its accumulator logic.
  - `tb_single_port_memory`: Test SRAM read write and data retention capabilities.
  - `tb_ceiling_counter`: Test the ceiling counter.
- **Integration tests:**
  - `tb_one_mac_gemm`: A testbench validating the full accelerator. It generates random matrices A and B, packs them into memory, runs the hardware accelerator, and compares the unpacked output C against a given golden model. 10 Randomized iterations were run each test.
  - `tb_gemm_performance`: A testbench testing the final three cases. It generates random matrices of test case 1, 2 and 3 sizes and runs analogously to the previous integration test. It concludes with a detailed report on the latency, operational efficiency and number of arithmetic operations.

6.2 Coverage results

All the modes in the datapath, the main FSM state transitions, and the address generation logic are heavily included in the tests. Also tile counters (M, N, K) and SRAM accesses are verified across with multiple random inputs.

6.3 Remaining gaps and future tests

The current tests rely on dimensions that align perfectly with tile boundaries (multiples of 4 or 16). I propose adding tests with odd or prime dimensions (e.g., $M = 17, N = 5$) to verify that the controller's padding logic and tile counters handle these unaligned conditions. Other edge cases such as a test with $K = 1$ to ensure the logic functions correctly when the compute phase is shorter than the drain phase. Additionally, a full 64×64 matrix multiplication test would stress test the address generation logic (because the DataDepth is set to 4096) to make sure there is no overflow and the logic remains intact at the capacity limit.