# FPGA-Accelerated Matrix Multiplication with Communication-Centric Performance Optimization

CSE 4602 Capstone Design Project Final Report

Submitted to Professor Michael Hall and the Department of Computer Science & Engineering
Washington University in St. Louis


**Team Members**

Amani Alqaisi - B.A. Computer Science, B.S. Computer Engineering - a.alqaisi@wustl.edu
Mack Liao - B.A. Computer Science, B.S. Computer Engineering - g.liao@wustl.edu
Minh La - Computer Engineering - l.minh@wustl.edu
Chuhan Qiao - Computer Engineering - chuhan.qiao@wustl.edu

**Project Advisor**

Michael Hall - Lecturer, Computer Science & Engineering - mhall24@wustl.edu



Date of Submission: May 1, 2026
Project Duration: January 2026 - April 2026

## Abstract

This project developed an FPGA-accelerated matrix multiplication system on the PYNQ-Z2 platform to address the challenge of performing tiled large-matrix computation under tight memory, communication, and hardware resource constraints. The final system used a three-tier architecture consisting of a client host running a React/FastAPI application, a board-side server on the Zynq processing system, and a 128-MAC broadcast multiply-accumulate compute core synthesized in Vitis HLS. During development, seven-phase timing instrumentation showed that communication and serialization, rather than raw FPGA arithmetic, dominated end-to-end latency. That finding led to a redesign of the backend-to-board interface from HTTP/JSON to a binary TCP protocol with persistent connections. The revised protocol reduced serialization overhead by roughly half, with the JSON-parse path on the host falling by more than 90%. The final system a peak inner-loop compute rate of about 5.1 GMAC/s, produced reproducible benchmark data, and demonstrated that full-stack measurement can guide more effective FPGA optimization than hardware-only tuning.


# 1. Introduction

## 1.1 Background and Motivation

Matrix multiplication is a core kernel in linear algebra, machine learning, scientific simulation, and signal processing. Its cost grows cubically with matrix dimension, so even moderate increases in problem size create a large increase in total work [1], [2]. FPGAs are attractive for this problem because they support spatial parallelism, custom data paths, and direct control over the memory hierarchy. The challenge is that the same memory hierarchy that makes acceleration possible also limits scale and often dominates performance on embedded boards.

The PYNQ-Z2 board combines a Zynq-7020 SoC, DDR3 memory, and a programmable logic fabric that can host custom accelerators [3]. Our project began with the assumption that a larger compute engine and careful tiling would be the central problem. By the end of the semester, the project showed that the harder systems problem was moving data efficiently between the host, processing system, and programmable logic. That shift in understanding shaped the final design and is the main technical story of the project.

## 1.2 Problem Statement

The project goal was to design a scalable matrix multiplication service on the PYNQ-Z2 that could process matrices larger than the board's fast local resources while still exposing a usable interface to a remote client. The final problem can be stated more precisely as follows: build a complete end-to-end system that accepts large matrix jobs, partitions them into tiles, executes those tiles on an FPGA accelerator, accumulates partial results correctly, and reports performance in enough detail to justify design decisions. The key constraints were limited board memory, limited DSP resources, tile-shape requirements imposed by the compute core, and the overhead of communication between software and hardware.

## 1.3 Objectives, Requirements, and Changes from the Proposal

The formal proposal targeted a 16 × 16 systolic array, double-buffered DMA transfers, and a staged path from an 8 × 8 baseline to a high-density hybrid DSP/LUT implementation [4]. The final project kept the core objective of building a tiled FPGA matrix multiplication system on the PYNQ-Z2, but several implementation choices changed once measurement data and toolchain behavior made parts of the original plan impractical.

| Proposal item | Final report status | Reason for change |
|---|---|---|
| True systolic array with inter-PE forwarding | Replaced by 128 broadcast outer-product MAC array | A Vitis 2024.2 cosimulation issue on the RTL black-box path blocked the original direction and the broadcast design was easier to verify and integrate. |
| Communication centered on HTTP/JSON and REST services | Backend-to-board path redesigned as binary TCP | Timing instrumentation showed that JSON serialization and parsing consumed most of the end-to-end latency. |
| Double-buffered AXI-DMA presented as a primary optimization | Final optimization emphasis shifted to persistent connections and protocol efficiency | The largest measured bottleneck was not raw board-side memory transfer but software serialization and network framing. |
| Stretch objective of very large global matrices up to 16,384 x 16,384 | Architecture remains tiled and scalable in concept, but final evaluation focused on representative benchmark sizes and per-tile latency | The semester schedule favored producing a verified, measurable full stack over pursuing a larger but less validated problem size. |

The final project objectives were therefore revised to emphasize a complete, measurable, and validated end-to-end system rather than only maximizing compute density. The final functional requirements were: (1) multiply arbitrary matrix sizes through padding and tiling, (2) provide a usable frontend and orchestration backend, (3) execute tiled matrix products on the FPGA and accumulate partial sums correctly, (4) verify each run against a software reference, and (5) expose timing and benchmark data for evaluation.

The final non-functional requirements were: (1) maintain correctness and reproducibility across benchmark runs, (2) fit within the memory and FPGA resource limits of the PYNQ-Z2 platform, (3) preserve modularity so that communication and compute components could be revised independently, (4) provide enough observability to support data-driven optimization, and (5) present results through a usable interface consistent with the project's full-stack design goals.

# 2. System Design and Implementation

## 2.1 System Overview

The final design is organized as a three-tier system, shown in Figure 1. At the highest level, the client host runs the user-facing React frontend together with the FastAPI backend, which accepts matrix inputs, partitions them into tiles, schedules tile-level jobs, and aggregates partial sums after execution. The host communicates with the PYNQ-Z2 board by sending tile data and control information over the network.

As shown in Figure 1, the PYNQ-Z2 processing system (PS) acts as the board-side controller between the host software and the hardware accelerator. It receives requests from the host, stages data in DDR3 memory, performs output scale-back and control logic, and coordinates execution of the compute kernel in programmable logic. The input float → int16 fixed-point conversion is performed on the client host. This layer separates high-level orchestration from board-level execution details and allowed the system to evolve from the earlier HTTP/JSON prototype to the final TCP-based implementation described later in this report.

The board-side hardware integration is shown in Figure 2. The Vivado block design includes the Zynq processing system, AXI interconnect infrastructure, reset and protocol-conversion logic, and the custom matrix multiplication IP block. Although the IP instance in the block design retained the legacy name systolic_tile_processor, the final compute engine is not a true systolic array. As documented in the project's final milestone summary, the implemented accelerator uses a 128 broadcast MAC architecture in which rows and columns are broadcast across the processing array rather than forwarded through a classical systolic dataflow. The older instance name was kept only for naming continuity in the Vivado project and does not reflect the final architecture.

Together, Figures 1 and 2 show the relationship between the software stack and the hardware platform: Figure 1 presents the system-level partitioning across host, board control, and programmable logic, while Figure 2 shows how the board-side components were physically integrated in Vivado to support the final distributed accelerator design.

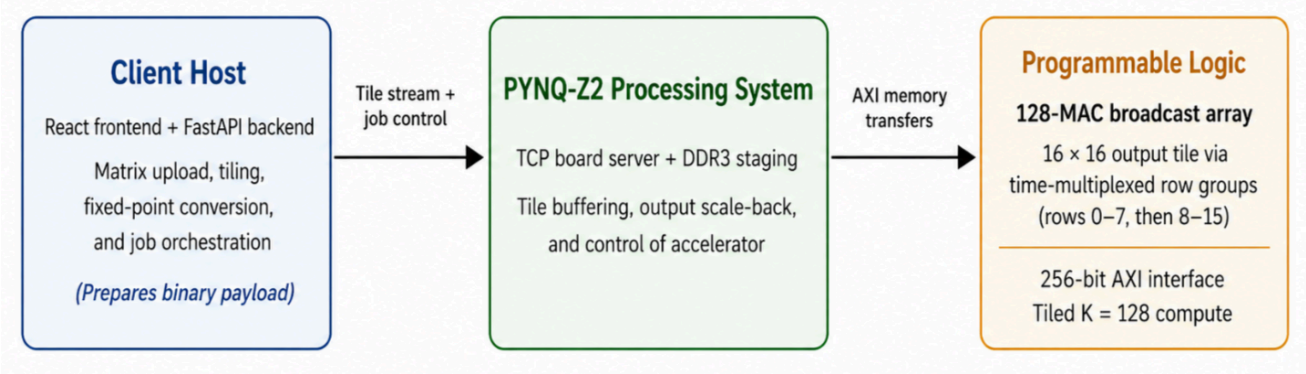


*Figure 1. Simplified three-tier architecture of the final distributed matrix multiplication system, showing the client host, PYNQ-Z2 processing system, and programmable logic compute core.*

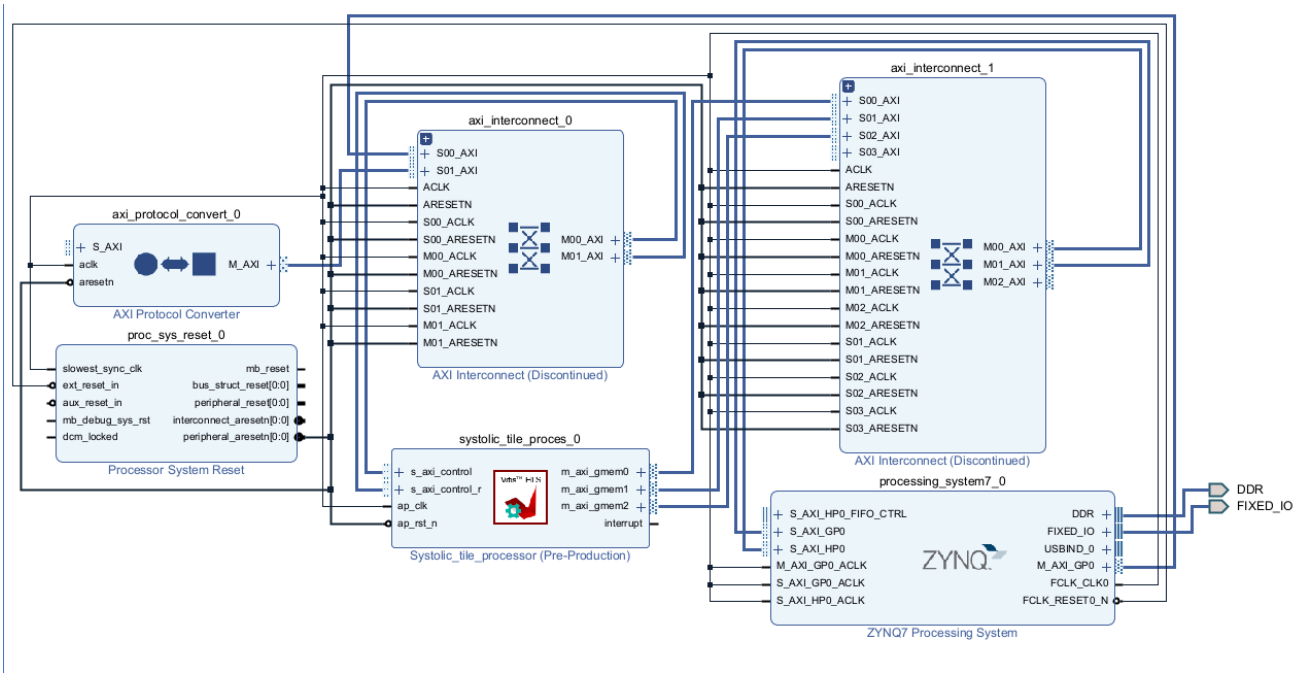


*Figure 2. Vivado block design of the integrated board-side system. The design includes the Zynq processing system, AXI interconnect fabric, reset/control infrastructure, and the custom matrix multiplication IP. The IP instance name systolic_tile_processor was retained from an earlier naming convention, although the final accelerator uses a broadcast MAC architecture rather than a true systolic array.*

## 2.2 Design Approach and Key Decisions

The project progressed through five milestones. Milestone 1 established a reliable host–board path with a trivial stream kernel. Milestone 2 turned that path into a full matrix multiplication service over HTTP/JSON, backed by an 8×8 broadcast HLS core. Milestone 3 was the heaviest optimization milestone: it widened the compute core to a 128-MAC broadcast array producing a 16×16 output tile, moved the memory interface to a 256-bit AXI bus, switched the host–board payload from JSON arrays to base64-packed int16 fixed-point, and added the seven-phase timing instrumentation that guided every later decision. Milestone 4 replaced HTTP with a raw TCP binary protocol over persistent, pooled connections, leaving the HLS core unchanged. Milestone 5 refined the HLS core by merging tile load and compute into a single pipelined compute-and-prefetch loop with double-buffered LUTRAM staging, hiding the DDR transfer behind compute. This milestone structure mattered because a measured bottleneck rather than guesswork drove each redesign.

| Decision | What was chosen | Why it mattered |
|---|---|---|
| Compute dataflow | 128-MAC broadcast array | Preserved high parallelism while avoiding a blocked RTL co-simulation path. |
| Numeric format | 16-bit fixed-point inputs with 48-bit accumulators | Balanced range, hardware cost, and compatibility with the board-side software conversion path. |
| Software architecture | React frontend + FastAPI backend + board server | Separated user interaction, orchestration, and hardware control into easier-to-debug layers. |
| Pipeline depth | Two in-flight requests | Allowed overlap of communication and computation without creating unproductive queueing on a single accelerator. |
| Transport protocol | Persistent TCP with packed binary headers | Removed major serialization and parsing overhead that was masking hardware gains. |

These decisions involved several trade-offs. The broadcast MAC dataflow was chosen over the original true systolic direction because it was easier to verify and integrate under the available toolchain, even though it changed the architectural story relative to the proposal. Similarly, persistent TCP and binary packing added implementation complexity, but they were justified because instrumentation showed that serialization overhead dominated end-to-end latency. Larger tile sizes reduced dispatch overhead but increased memory pressure and integration difficulty.

## 2.3 Hardware Compute Core

The final programmable-logic design was implemented as a tiled matrix-multiplication accelerator on the PYNQ-Z2 using Vitis HLS 2024.2 and integrated into the board-side system through the Vivado block design shown earlier in Figure 2. Although the accelerator produces a 16 × 16 output tile, the final optimized kernel does not activate 256 multiply-accumulate units simultaneously. Instead, it uses a 128-MAC broadcast array and fills the 16 × 16 tile by time-multiplexing two row groups, computing rows 0-7 and then rows 8-15 on alternate phases. This preserved the final tile structure while reducing hardware cost enough to fit the resource budget of the xc7z020. The final kernel also used a 256-bit AXI interface and a tiled K-depth of 128, which reduced memory-transfer overhead compared with earlier narrower-bus versions [5], [6].

This hardware organization reflects several important design decisions and trade-offs. The project originally proposed a more explicitly systolic direction, including RTL-based exploration, but the final implementation moved to a broadcast outer-product HLS design because it was easier to verify, easier to integrate into the board-side system, and less vulnerable to the Vitis 2024.2 co-simulation issue that blocked the earlier RTL black-box path [4], [5]. The two-group compute schedule was another deliberate trade-off: it reduced the number of simultaneously active MAC units from 256 to 128 while still preserving a 16 × 16 output-tile interface. Similarly, widening the AXI interface to 256 bits reduced tile-loading cycles substantially, but it required a more careful memory layout and more aggressive partitioning and prefetch control in the HLS kernel [5], [6]. The runtime fixed-point format was Q7.8 (frac_bits = 9 on the host), traded down from the board's nominal Q1.14 default to add ~7 bits of input dynamic range. That choice avoided clipping on standard-normal-distributed test matrices, which would otherwise have saturated the default range of approximately [-2, 2).

The post-implementation hardware results show that the final design fit the target FPGA while using a substantial share of the available logic resources. As shown in Figure 3, the integrated design used 41,429 LUTs (77%), 61,575 FFs (57%), 142 DSPs (64%), and 34 BRAM_18K blocks (12%). These values indicate that the final hardware refinement successfully increased computational density and memory-path efficiency while remaining within the resource envelope of the PYNQ-Z2. The design therefore did not maximize raw parallelism in the abstract; instead, it balanced compute width, on-chip storage, and data-movement efficiency in a way that was implementable and verifiable on the actual board [5], [6].

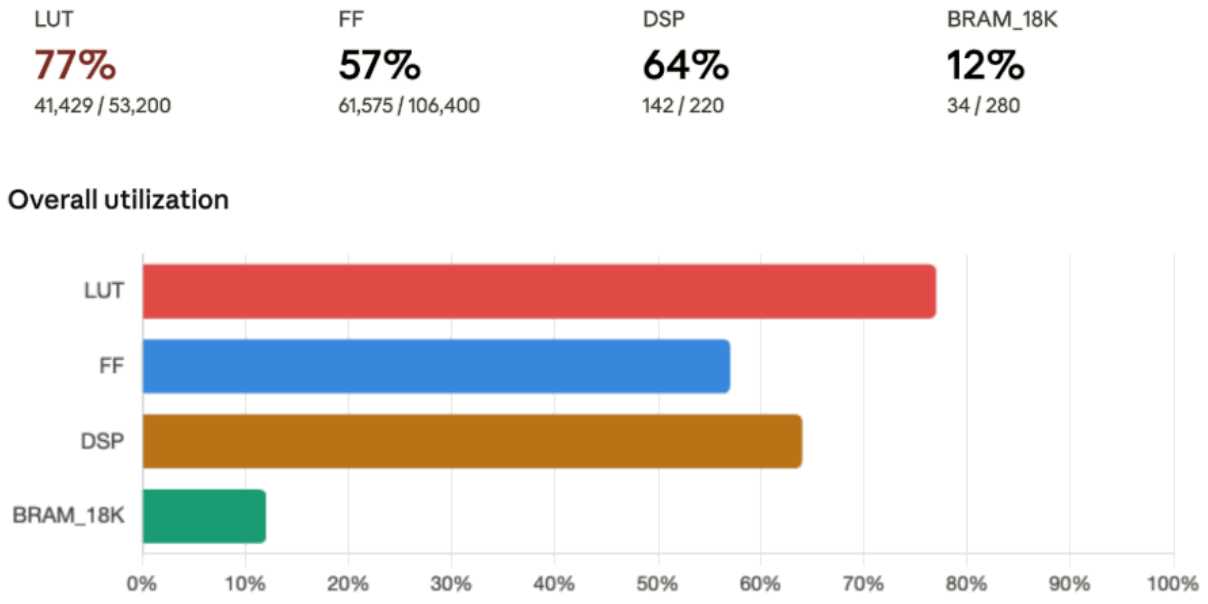


*Figure 3. Overall FPGA resource utilization of the final integrated accelerator design. Post-implementation utilization on the PYNQ-Z2 showing LUT, FF, DSP, and BRAM_18K consumption for the final board-side system*

Figure 4 provides a more detailed view of where those resources were consumed inside the final HLS-generated design. The compute_and_prefetch region dominated DSP usage and also accounted for most LUT consumption, which is consistent with the pipelined tiled-compute structure and overlapping load/compute behavior used in the optimized kernel. Flip-flop usage was distributed

more broadly between the compute region, memory-related logic, and surrounding glue logic, while BRAM usage was concentrated in the memory-interface blocks associated with the global-memory ports. This breakdown is useful because it shows that the limiting factors in the final implementation were not only arithmetic units, but also the control, buffering, and interface structures needed to sustain tiled execution at a practical throughput [6].

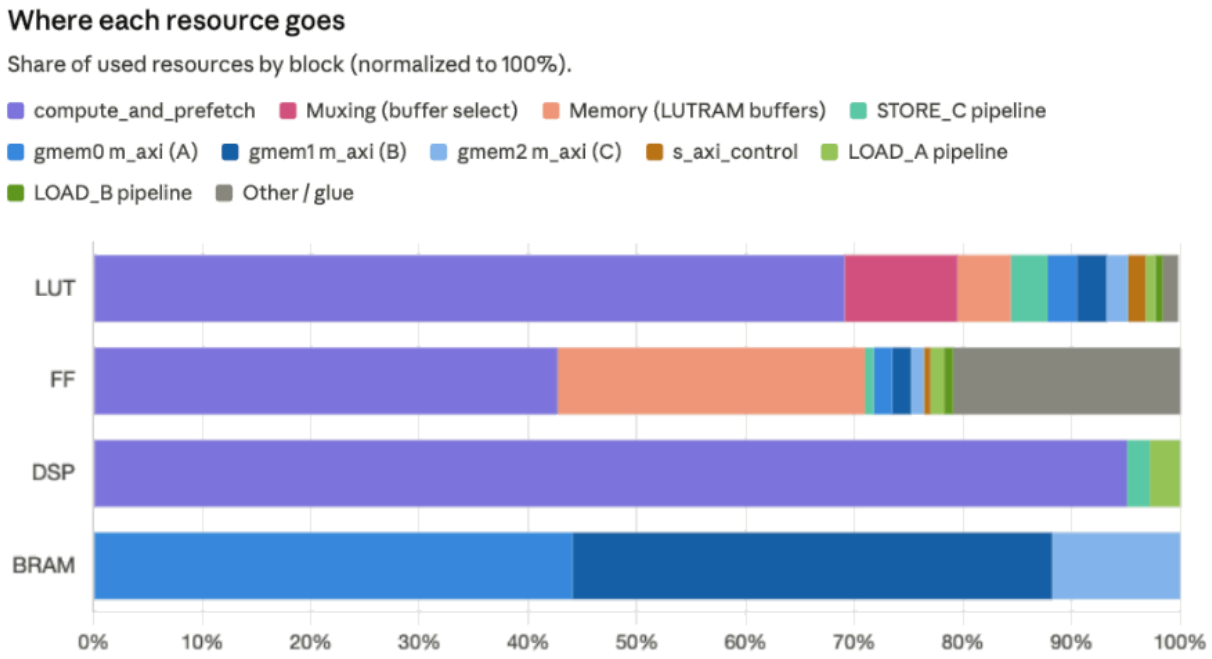


*Figure 4. Normalized breakdown of FPGA resource consumption by major design block. Share of used LUT, FF, DSP, and BRAM resources attributed to the compute-and-prefetch kernel, memory-interface logic, control logic, and supporting glue components*

The hardware core was built and refined using Vitis HLS for kernel development and Vivado for block-design integration, synthesis, and implementation. Board-level integration required repeated inspection of AXI connectivity, addressing, and clock/reset wiring because automation did not always produce a reliable final design. Validation combined HLS-level testing, synthesis and implementation checks, and full end-to-end hardware testing on the PYNQ board against a NumPy software reference. This testing flow was especially important because fixed-point scaling, host-side accumulation, and software/hardware interface behavior all had to remain consistent across the full stack. Together, the implementation results in this section show not only what hardware was built, but also why the final accelerator took its implemented form rather than the more idealized one proposed at the start of the semester [4]–[6].

## 2.4 Software and Communication Stack

The backend owned the high-level blocked multiplication algorithm. It padded matrices to valid tile boundaries, generated the triple-nested tile schedule over output rows, output columns, and inner-dimension tiles, sent each tile pair to the board, and accumulated the returned partial tile into the final result. The frontend exposed matrix upload, random matrix generation, a tile-status grid, a timing panel, and a benchmark page. Live updates to the tile grid and progress bar were driven by a backend Server-Sent Events endpoint (/api/jobs/{id}/progress), where {id} denotes the active job identifier. The endpoint emitted one event per tile state change and a final event carrying the

aggregated seven-phase timing payload used to generate Figure 7. The board server evolved from Flask-based HTTP endpoints to a threaded TCP server (socketserver.ThreadingMixIn) in which a single connection carries many sequential tile requests, while an in-process lock around the accelerator serializes hardware access. The host opens PIPELINE_DEPTH = 2 such connections at job start and recycles them through an asyncio queue, so TCP setup cost is amortized across the entire job rather than paid once per tile.

```
REQ_HDR_FMT = "!16sIIIHH"      # request_id, M, K, N, frac_bits, flags
RESP_HDR_FMT = "!16sBIIIII"    # request_id, status, M, N, compute, deser, ser

def pack_request(request_id: bytes, M: int, K: int, N: int,
                 frac_bits: int, flags: int = FLAG_RETURN_FULL) -> bytes:
    return struct.pack(REQ_HDR_FMT, request_id, M, K, N, frac_bits, flags)

def pack_response(request_id: bytes, M: int, N: int,
                  compute_us: int, deser_us: int, ser_us: int) -> bytes:
    return struct.pack(RESP_HDR_FMT, request_id, STATUS_OK,
                       M, N, compute_us, deser_us, ser_us)
```

*Listing 1. Binary protocol helpers from Milestone_4/backend/services/protocol.py. The packed headers replaced the earlier JSON and base64 path.*

```
# (one of PIPELINE_DEPTH persistent connections held by the pool)
reader, writer = await asyncio.open_connection(settings.FPGA_HOST, settings.FPGA_PORT)
result_tile, tile_timings = await dispatch_tile(
    reader, writer, A_tiles[i][k], B_tiles[k][j],
    frac_bits=settings.FRAC_BITS, timeout=settings.REQUEST_TIMEOUT,
)
t_accum_start = time.time()
accumulate(C, result_tile, i, j, T)
tile_timings["accumulation_sec"] = round(time.time() - t_accum_start, 6)
all_tile_timings.append(tile_timings)
```

*Listing 2. Orchestration logic from Milestone_4/backend/services/orchestrator.py. The backend measured each phase and accumulated K-tiles into the final matrix.*

## 2.5 Implementation Workflow, Testing, and Debugging

The implementation combined Vivado, Vitis HLS, Python on the PYNQ board, FastAPI on the host, and a React/TypeScript frontend. Testing took place at several levels. The earliest milestone validated bitstream loading and DMA data motion with a simple y = 2x + 1 function. Later milestones added backend unit tests for slicing, accumulation, and orchestration; board-side health checks; full NumPy result verification; and end-to-end benchmarks that exercised both the user interface and the accelerator. A software mock of the FPGA (backend/mock_fpga.py) implementing the same binary wire protocol let the orchestrator, dispatcher, and frontend be exercised end-to-end without the physical board, which made it possible to develop and unit-test the M4 protocol redesign in parallel with hardware bring-up. Debugging work concentrated on manual Vivado wiring, memory-map access, fixed-point scaling, cross-platform board deployment, and the eventual move away from the stalled RTL co-simulation path.

# 3. Evaluation and Results

## 3.1 Testing and Data Collection

The final evaluation used four categories of evidence: correctness, latency, scalability behavior, and hardware implementation cost. Correctness was tested by comparing FPGA-generated matrix products against a NumPy software reference after applying the same fixed-point scaling used by the board-side conversion path. These checks verified both individual tile products and the final accumulation of partial tiles into a complete matrix result.

The evaluation used randomly generated matrices, uploaded matrix files through the frontend, and benchmark runs across the supported tile sizes — 128, 256, 512, 1024, and 2048. The 2048 option, added at the start of Milestone 5 along with a MAX_DIM bump on the board, lets a 2048 × 2048 multiplication finish in a single host–board dispatch and exercises the board's pre-allocated DMA buffers at full size.

Latency data was collected using phase-level instrumentation distributed across the backend and board server. The measured phases included client serialization, network round-trip, board deserialization, board computation, board serialization, client deserialization, and accumulation. Hardware cost data, including clock estimates and FPGA resource utilization, was taken from Vivado and Vitis implementation summaries. Together, these measurements were used to determine whether runtime was limited primarily by compute throughput, communication overhead, or hardware resource constraints.

| Evaluation category | How it was collected | Purpose |
|---|---|---|
| Correctness | NumPy golden-model comparison and tile accumulation checks | Verify functional correctness across tiled jobs. |
| Latency | Seven-phase timing instrumentation in backend and board server | Identify whether communication or computation dominated runtime. |
| Scalability behavior | Benchmark sweeps over tile sizes and matrix sizes | Measure how dispatch count and tile granularity affected throughput. |
| Hardware cost | Vivado/Vitis implementation summaries | Relate performance gains to resource usage and timing closure. |

## 3.2 Results

A representative end-to-end comparison is a 2048 × 2048 multiplication, which decreased from a projected M1 baseline of about 368 s to 7.30 s on the final M5 system, an overall improvement of about 50×. As shown in Figure 6, this reduction did not come from one isolated optimization. Instead, each milestone removed a different dominant source of overhead, and only after one bottleneck was reduced did the next one become visible. The linear and logarithmic views in Figure 6 are both useful here: the linear scale shows the absolute drop in wall-clock time, while the log scale makes the later improvements easier to compare after the largest early reductions had already occurred.

Milestone 1 served as the high-latency baseline rather than a full matrix-multiplication implementation. It demonstrated a single DMA round-trip using a simple y = 2x + 1 kernel, and the projected 368 s total for a 2048 × 2048 multiplication was extrapolated from that per-request behavior. Most of that projected cost came from repeated kernel setup and text-based transport rather than useful matrix computation. Milestone 2 removed one major source of avoidable overhead by moving kernel creation out of the per-request path and reusing a single kernel handle across tiles. As listed in Table 3.1, that change reduced total job time to 63.5 s, an 83% reduction from the baseline. At that point, the dominant remaining cost was no longer setup overhead but the matrix payload itself.

Milestone 3 then improved the transport of matrix data while preserving the same general HTTP-based structure. The JSON object representation was replaced with a base64 matrix transfer over HTTP, reducing total job time to 27.3 s, a further 57% improvement over Milestone 2, as shown in Figure 6 and summarized in Table 3.1. Milestone 3 also widened the on-board compute organization from the earlier 8 × 8 style path to a 128-MAC array producing a 16 × 16 output tile, while introducing the 256-bit AXI bus for wider transfers. Even with those changes, the per-phase data showed that transfer and protocol costs still dominated user-visible runtime. That conclusion appears clearly in Figure 5, where the M3 per-tile wall-clock decomposition shows that network round-trip and related transfer costs still occupied a larger share of time than the FPGA compute phase itself.

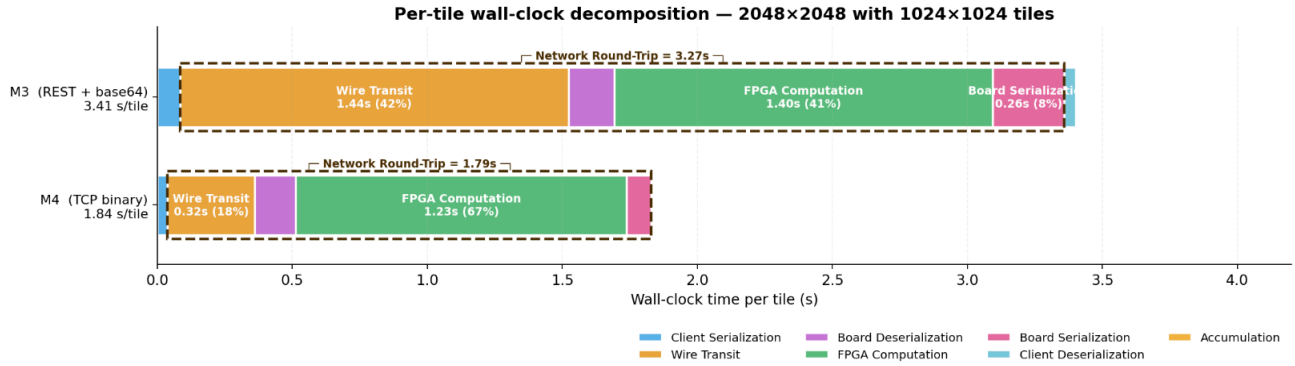


*Figure 5. Per-tile wall-clock decomposition for M3 and M4 on a 2048 × 2048 multiplication with 1024 × 1024 tiles.*

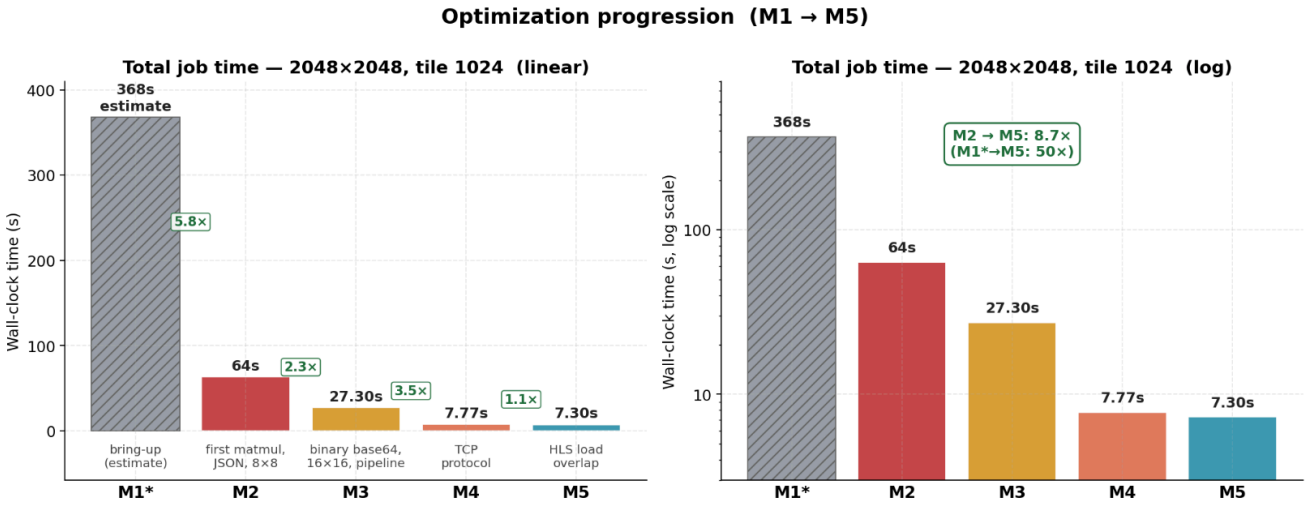


*Figure 6. Optimization progression across milestones M1 through M5, showing total job time for a 2048 × 2048 matrix multiplication with tile size 1024 on linear and logarithmic scales.*

The most important system-level change came in Milestone 4, which replaced the REST/HTTP path with a raw TCP binary protocol using packed headers and persistent connections. This reduced total job time to 7.77 s, a 72% improvement over Milestone 3, as shown in Figure 6 and Table 3.1. The effect of that redesign is shown in two complementary ways. First, Figure 5 compares the per-tile wall-clock decomposition for M3 and M4 on a 2048 × 2048 multiplication with 1024 × 1024 tiles. In M3, each tile took about 3.41 s, with network transfer and surrounding communication costs dominating the tile budget. In M4, the same tile dropped to about 1.84 s, and the FPGA's share of the tile time became much larger. Second, Figure 7 breaks the timing down by phase and shows where the savings came from. Client serialization, wire transit, board serialization, and client deserialization all fell substantially from M3 to M4, while FPGA compute changed only modestly. This is strong evidence that the major speedup in Milestone 4 came from communication redesign rather than from a change in arithmetic throughput.

Only after Milestone 4 did the system begin to behave like a genuinely compute-centered accelerator. In Milestone 3, the FPGA accounted for roughly 41% of per-tile time, while communication still dominated the remainder. In Milestone 4, the FPGA share rose to roughly 67%, as seen in Figure 5, which meant that further hardware refinement could finally produce visible end-to-end benefit. Milestone 5 then addressed the remaining board-side inefficiencies through HLS-level refinement of the compute core, including merging previously sequential tile load and compute behavior into a single pipelined compute_and_prefetch loop with double-buffered LUTRAM staging and an II = 1 inner pipeline. That final step reduced total job time from 7.77 s to 7.30 s, as shown in Figure 6 and Table 3.1. The absolute improvement was smaller than the earlier protocol redesign, but by this stage the system had already removed the most severe software-side bottlenecks.

Figure 5 is therefore the clearest evidence that the system changed from being communication-bound to being much closer to compute-bound, while Figure 7 shows exactly which leaf phases were reduced to make that transition possible. Figure 6 then places those changes in the broader milestone progression by showing how the total job time fell across M1–M5. Together, these figures show that the project's performance gains were cumulative and measurement-driven rather than the result of a single optimization.

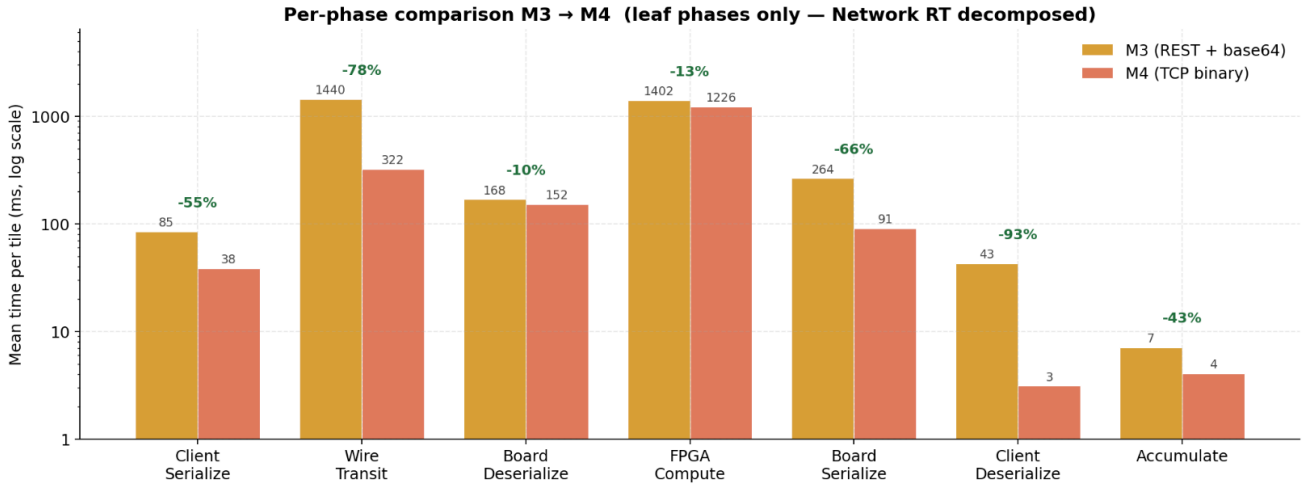


*Figure 7. Mean per-tile phase timing comparison for Milestone 3 and Milestone 4, with network round-trip decomposed into its leaf phases.*

The milestone summary in the following table consolidates that progression. It shows that the largest early reduction came from eliminating repeated kernel creation, the next major reduction came from improving matrix transfer format, the most dramatic system-level reduction came from replacing REST/HTTP with raw TCP binary transport, and the final refinement came from HLS-level overlap of load and compute in the optimized hardware core.

| Milestone | Total Job Time (2048×2048) | Reduction vs Previous | Key Change |
|---|---|---|---|
| M1 (Baseline) | ~368 s | Baseline | Kernel created per tile; JSON over HTTP host–board path. |
| M2 | 63.5 s | 83% | Moved kernel creation out of the per-request path and reused a single kernel handle across tiles. |
| M3 | 27.3 s | 57% | JSON object representation switched to a base64 matrix transfer over HTTP. On-board array widened to 128 MACs and 256-bit AXI bus introduced. |
| M4 | 7.77 s | 72% | Protocol switched from REST/HTTP to raw TCP binary path using packed headers and persistent connections. |
| M5 | 7.30 s | 6% | HLS core optimized by merging tile load/compute into a single pipelined compute_and_prefetch loop with double-buffered LUTRAM staging, achieving II=1. |

These results show that the final system met its core end-to-end objectives. It accepted matrix inputs through the interface, executed tiled FPGA-backed multiplication correctly, exposed progress and timing information during execution, and reduced overall runtime through measured communication and hardware optimizations. Just as importantly, the figures in this section show why those gains occurred: for much of the semester the system was communication-bound, so hardware improvements alone would not have delivered comparable end-to-end speedup.

### 3.3 Evidence Shown in the Final Presentation

The final presentation briefly demonstrated the system's complete operating path rather than only isolated hardware results. It showed the end-to-end pipeline from browser-based matrix input, backend tiling and orchestration, and persistent TCP dispatch to the PYNQ board, through FPGA execution, NumPy-based verification, and frontend progress and timing output. It also presented milestone-based performance results, per-phase timing breakdowns, implementation visuals such as the binary protocol and Vivado block design, and a live demo with a backup demo prepared as supporting evidence of system operation. This kept the report and presentation aligned around the same central result: a complete, measured FPGA-accelerated matrix multiplication system with evidence of both functionality and performance.

# 4. Discussion and Reflection

## 4.1 Technical Discussion

The results show that the most important engineering lesson of the project was not simply that FPGA acceleration can speed up matrix multiplication, but that end-to-end performance depends on the entire system path. The final milestone data showed a reduction from a projected 368 s baseline in M1 to 7.30 s in M5 for a 2048 × 2048 multiplication, an overall improvement of about 50×. However, that improvement did not come primarily from raw arithmetic scaling on the programmable logic. Instead, the largest visible gains occurred when the team removed overheads outside the compute kernel, especially repeated kernel setup and text-based communication. This is why the protocol redesign in M4 produced a much larger end-to-end improvement than the later HLS refinement in M5.

The milestone progression also explains how the dominant bottleneck changed over time. M2 showed that moving kernel creation out of the per-request path eliminated a major source of avoidable overhead and cut total job time dramatically. M3 reduced the cost of matrix transfer and widened the on-board compute organization, but the timing breakdown still showed that communication remained the dominant cost. The figures in the previous section make this especially clear: in M3, the network-related portion of each tile still consumed more time than FPGA execution, while in M4 the binary TCP redesign reduced those communication costs enough that FPGA compute became the largest remaining share of per-tile time. That shift matters because it means later hardware optimization finally had a realistic chance of affecting user-visible runtime.

This pattern is significant because it directly answers the project's central design question. The original intuition from the proposal emphasized a more compute-centered view of performance, including systolic structure, resource mapping, and communication hiding through hardware-oriented overlap. Those goals were reasonable, but the final measurements showed that optimizing arithmetic throughput alone would not have produced comparable end-to-end gains until the software and protocol path had first been improved. In that sense, the project validated a measurement-driven full-stack methodology: instrument the system, identify the dominant bottleneck, redesign that bottleneck, and only then move to the next layer of optimization.

The results also show that the final system satisfied the project's main practical goal better than some of the original proposal-specific goals. It delivered a complete three-tier accelerator system with frontend input, backend orchestration, board-side execution, verification against a software baseline, progress reporting, and timing visibility. It did not fully preserve the exact architectural path proposed at the start of the semester, since the final hardware moved away from the original true systolic/hybrid DSP-LUT direction toward a broadcast-style HLS implementation. But that deviation was not arbitrary; it was caused by toolchain limitations and reinforced by evidence that software and communication overhead were the more urgent performance problems to solve first.

### 4.2 Requirements Satisfaction and Remaining Gaps

The project satisfied the core functional and non-functional requirements more successfully than it satisfied every proposal-era architectural detail. That distinction is important. The final system did what the capstone most needed it to do: it accepted inputs, executed tiled matrix multiplication on the FPGA platform, verified correctness, exposed progress and timing, and produced defensible measurements that explained its behavior. The remaining gaps are therefore best understood as differences between the original technical path and the final implemented one, rather than as failures of the capstone itself.

| Requirement | Outcome | Comments |
|---|---|---|
| Working matrix multiplication accelerator on PYNQ-Z2 | Satisfied | The final system executed tiled matrix multiplication end to end and returned verified results. |
| User-facing software stack | Satisfied | Frontend, backend, board server, progress visualization, timing views, and benchmark support were completed. |
| Quantitative performance evidence | Satisfied | Milestone comparisons, per-phase timing, and benchmark data were collected and used to justify design decisions. |
| Reduction of end-to-end latency, not just kernel time | Satisfied | Total job time fell from the projected M1 baseline of 368 s to 7.30 s in M5, with the largest gains coming from reduced setup and communication overhead. |
| Correctness and verification against software reference | Satisfied | FPGA-backed tiled results were checked against a NumPy golden model and accumulation correctness was validated across full jobs. |
| Original systolic plus hybrid DSP/LUT implementation path | Partially satisfied | The final design kept the tiled accelerator concept but moved to a broadcast HLS implementation rather than completing the exact original systolic/hybrid path. |
| Communication hiding and overlap | Satisfied in revised form | Overlap was achieved through persistent connections, pipelined orchestration, and compute-and-prefetch behavior rather than the originally emphasized DMA-centered approach alone. |

### 4.3 Strengths, Trade-offs, and Significance

The final system's main strength was that it functioned as a complete, measurable pipeline rather than only as an isolated hardware kernel. Its modular separation between frontend, backend orchestration, board server, and programmable logic made redesigns manageable, especially when the communication path had to be replaced. The timing instrumentation was another major strength because it made optimization decisions evidence-based and exposed when the dominant bottleneck had shifted from setup overhead, to communication, and finally closer to compute.

The final design also reflected several important trade-offs. Larger tiles reduced dispatch overhead but increased memory pressure and integration difficulty. The binary TCP redesign added implementation complexity, but it was justified because the earlier REST/HTTP path was dominating runtime. The final broadcast-style HLS design was easier to verify and integrate than the original systolic RTL direction, but it also meant the final hardware differed from the proposal's initial architectural path.

The broader value of the project is that it showed why accelerator systems must be evaluated end to end. The largest performance gains did not come first from changing arithmetic throughput on the FPGA, but from removing software and communication overhead that had been masking hardware benefit. That result makes the project useful beyond this specific implementation: it demonstrates that practical FPGA acceleration depends on hardware/software co-design, measurement, and bottleneck-driven optimization rather than compute design alone.

### 4.4 Ethical Considerations and Broader Impact

Although this project does not raise immediate safety-critical concerns, it still has meaningful ethical implications. First, correctness matters more than raw speed. A fast accelerator has little value if tiling, numeric conversion, or software/hardware integration produces incorrect matrix outputs. For that reason, the system emphasized verification against trusted software results and treated correctness checks as part of the evaluation process rather than as an afterthought. The verifier scales its comparison tolerance with matrix size: 0.1 for inputs up to 1024 × 1024, and 0.5 once any dimension exceeds 1024. The widened tolerance reflects the fact that 16-bit fixed-point quantization noise accumulates roughly linearly along the inner dimension K, so a fixed 0.1 threshold would have produced false failures on 2048-class jobs. Disclosing this scaling in the verification path keeps the comparison honest rather than appearing to pass larger jobs by silent rounding.

Second, the project raised an issue of honest performance reporting. It would have been misleading to claim improvement based only on FPGA compute time while ignoring the costs of serialization, transfer, and host-side orchestration. Because the system was evaluated as a full pipeline, the reported gains more accurately reflect real user-visible performance rather than isolated kernel behavior. This is especially important for acceleration projects, where selective reporting can exaggerate the practical value of the hardware.

Third, there are security and data-exposure considerations in any distributed service model. In its project form, the system used unsecured network communication between components, so input matrices and results could be exposed in transit if deployed beyond a trusted environment. A more production-ready deployment would therefore place encryption and authentication in front of the service, such as HTTPS or a similarly protected communication layer, to reduce unnecessary exposure.

More broadly, accelerated matrix multiplication is a building block for machine learning inference and large-scale data processing. Improving access to lower-cost acceleration platforms can broaden who is able to experiment with such systems, which has educational and research value. At the same time, higher-throughput compute can support both beneficial and harmful applications. For this reason, the most defensible ethical stance for this project is to emphasize correctness, transparent measurement, reproducibility, and responsible reporting. Energy efficiency is also relevant: reducing unnecessary communication overhead avoids wasted transfer and wasted computation, which becomes more important as systems scale beyond the classroom setting.

### 4.5 Project Reflection and Lessons Learned

Several aspects of the system design worked well. The staged milestone structure helped the team validate the stack incrementally, and the Milestone 1 proof-of-concept established a stable end-to-end path before the full matrix multiplication system was introduced. The final three-tier separation between frontend, backend orchestration, and board-side execution also worked well because it made the system easier to debug and allowed performance bottlenecks to be isolated more clearly. In addition, the timing instrumentation framework proved especially valuable because it turned performance debugging into a data-driven process rather than guesswork.

At the same time, several limitations and challenges remained. Vivado automation could not always be trusted, so manual inspection of block connections, AXI addressing, and integration details was often necessary. Fixed-point arithmetic required careful host-side scaling and validation, and small mistakes in that process could produce misleading results. Toolchain behavior, especially the Vitis 2024.2 co-simulation issue, also directly affected architecture choices and prevented the team from keeping the original systolic-array plan unchanged. End-to-end runtime remained constrained by communication overhead and by the fact that the system ultimately depended on a single FPGA endpoint.

If the system were extended further, the first improvements would be reducing the remaining communication overhead, strengthening security through authenticated and encrypted communication, and moving more accumulation or scheduling intelligence closer to the hardware so that less orchestration work stayed on the host. The project would also benefit from a cleaner repository structure, a more reproducible deployment flow, and stronger automation for testing and setup. These changes would make the design easier to maintain, benchmark, and scale beyond the capstone setting.

The most important lesson from the project was that successful system design depends on measuring the real bottleneck rather than optimizing around assumptions. The project began with a compute-centered view of acceleration, but the final results showed that serialization, transfer, and orchestration were just as important as the FPGA core itself. More generally, the work reinforced the value of building a working vertical slice early, validating correctness continuously, and using instrumentation to guide redesign decisions. Those lessons were as important as the final speedup, because they shaped how the system was evaluated and why the final architecture took its eventual form.

# 5. Project Management

## 5.1 Planned Versus Actual Deliverables

The project delivered the major system elements proposed at the start of the semester, but several of them changed in implementation as the design evolved. The following table compares the original planned deliverables with the final completed system.

| Planned deliverable | Actual final deliverable |
|---|---|
| PYNQ-based matrix multiplication accelerator | Completed; evolved into a tiled accelerator that produced a 16 × 16 output tile using a 128-MAC broadcast-based implementation in the final optimized hardware path. |
| Host-side orchestration and board-side driver | Completed; expanded into a full frontend/backend/board stack with progress and benchmark views. |
| Detailed performance evaluation | Completed; strengthened through seven-phase timing instrumentation, benchmark sweeps, and milestone-to-milestone comparisons. |
| Hybrid DSP/LUT systolic implementation and double-buffered DMA emphasis | Partially completed / revised; the final project did not deliver the exact original systolic/hybrid path, and instead delivered a broadcast-style compute core, binary TCP communication, persistent connections, and revised overlap/prefetch optimizations. |

## 5.2 Timeline and Milestones

The project followed the semester milestone structure closely, but the technical emphasis changed as implementation results and measurement data accumulated. Early work remained aligned with the original plan: the proposal and architecture phase established the system direction, and Milestone 1 successfully demonstrated a working end-to-end hardware/software path through a simple DMA proof-of-concept. Milestone 2 then produced the first complete full-stack matrix multiplication service.

Later milestones remained on schedule in terms of course deliverables, but the design direction changed in response to what the team learned during implementation. Milestone 3 shifted the project toward instrumentation and benchmarking once it became clear that performance could not be improved responsibly without detailed timing visibility. Milestone 4 then redirected the optimization effort from the originally emphasized DMA-centered approach to protocol redesign and persistent TCP communication, since measured results showed that serialization and network overhead were dominating end-to-end latency. Milestone 5 focused on hardware refinement within the final architecture rather than returning to the original systolic direction.

As shown in Figure 8, the team adhered to the overall semester schedule and completed each major milestone in sequence, even though some of the technical goals changed relative to the proposal. The final phase concentrated on integration, final demonstration, and report preparation, which was consistent with the planned course timeline.

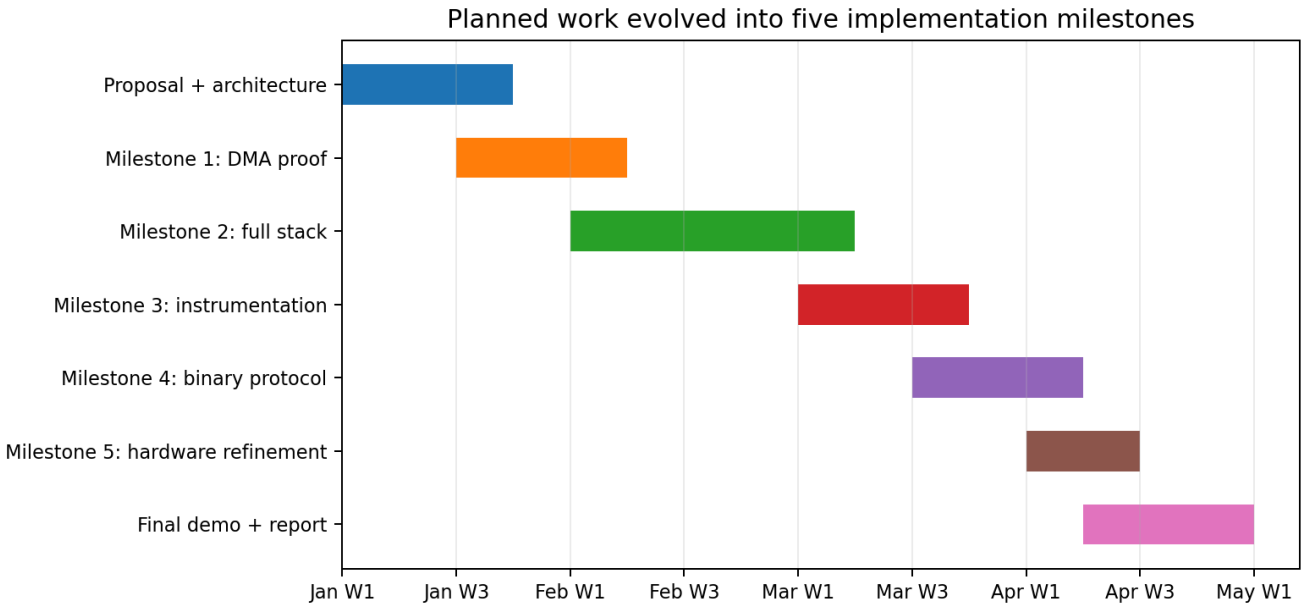


*Figure 8. High-level project timeline aligned to the semester milestone structure.*

## 5.3 Team Contributions

The table below summarizes the division of responsibilities reflected in the repository history, milestone documentation, and final integration work. It is written to identify concrete contributions instead of generic statements.

| Team member | Primary contributions |
|---|---|
| Amani Alqaisi | Integrated backend, frontend, and board-side workflows; corrected and rewired portions of the Vivado block design during later hardware integration; handled cross-platform deployment issues around the PYNQ environment; coordinated team logistics including scheduling meetings and helping keep milestone work on track; prepared the written project materials including major report drafting and revision work across the proposal, milestone reports, and final report; and led presentation preparation by building milestone and final presentation slides, organizing practice runs, and shaping speaking flow and content for demos. |
| Mack Liao | Served as lead developer for the HLS compute path; conceptualized and implemented the 256-bit AXI memory interface and compute-and-prefetch pipelining; spearheaded system-wide benchmarking and the seven-phase timing instrumentation framework; developed the binary TCP wire protocol to resolve communication bottlenecks; directed the technical execution by coordinating task distributions, structuring project agendas, and leading the final iterative performance optimization. Reviewed and proofread the final written report. |
| Minh La | Performed hardware validation and synthesis-oriented debugging on RTL paths; helped collect implementation results from the hardware flow; contributed board-side hardware testing and benchmarking during milestone development. |
| Chuhan Qiao | Worked separately on RTL-path exploration, synthesis, and hardware testing; contributed to initial board-side integration and verification activities. |

# 6. Conclusion and Future Work

## 6.1 Conclusion

This capstone produced a complete FPGA-accelerated matrix multiplication system spanning the frontend, backend, board server, and programmable-logic compute core. Across the semester, the project reduced the runtime of a representative 2048 × 2048 multiplication from a projected 368 s baseline to 7.30 s in the final system, while maintaining correct tiled execution and verification against a software reference. The final implementation delivered a working three-tier architecture, detailed timing instrumentation, and a communication redesign that replaced HTTP/JSON with binary TCP, which proved more important to end-to-end performance than additional arithmetic optimization alone.

The most important finding of the project was that FPGA acceleration had to be evaluated as a full pipeline rather than as an isolated compute kernel. The project began with a compute-centered proposal, but the results showed that communication, serialization, and orchestration overhead were the dominant bottlenecks for much of the semester. In that sense, the most valuable outcome was not only a functioning accelerator, but a clearer understanding of how full-system FPGA performance depends on measurement, hardware/software co-design, and bottleneck-driven optimization rather than raw compute throughput alone.

## 6.2 Future Work

Several extensions would improve both the technical scope and practical maturity of the system. First, a more complete benchmark campaign should be conducted across a wider range of matrix sizes, tile sizes, and network conditions to better characterize end-to-end performance. Second, the current single-board execution model could be extended to a multi-board or PCIe-based design to reduce communication overhead and improve scalability. Third, the team would revisit a true systolic implementation, including the earlier RTL-oriented direction, once the toolchain path is more stable, so the original architectural idea can be evaluated without the co-simulation issues that affected this project.

Additional hardware/software refinements would also improve the final system. Moving more accumulation and scheduling logic closer to the hardware would reduce host-side orchestration cost and could improve end-to-end efficiency. Further hardware optimization could also target remaining memory-transfer overheads, especially on the board-side path between staged data and compute execution.

To make the system more production-ready, the communication path should be secured with authentication and encryption rather than relying on an unsecured prototype deployment model. The deployment flow should also be packaged into a cleaner public repository structure with reproducible setup scripts, stronger automation, and a more unified release layout. Together, these changes would make the project easier to benchmark, reproduce, extend, secure, and deploy beyond the capstone setting.

## References

[1] NVIDIA Corporation, "Matrix Multiplication Background," NVIDIA Developer Documentation, 2023.

[2] "Computational complexity of matrix multiplication," Wikipedia, The Free Encyclopedia.

[3] Xilinx, Inc., "PYNQ-Z2 Board Reference Manual," UG1460, v1.0, 2021.

[4] A. Alqaisi, M. La, M. Liao, and C. Qiao, "Scalable Hardware Accelerator for Large-Scale Matrix Multiplication," CSE 4602 formal proposal, Feb. 10, 2026.

[5] Team Amani-Mack, "Capstone Milestone Report: FPGA-Accelerated Matrix Multiplication," Apr. 2026.

[6] Team repository, "cse4602-team-amani-mack-capstone-main," private Git repository used for implementation artifacts and code excerpts, accessed Apr. 15, 2026.

## Appendix A. Repository and Implementation Notes

The primary project repository for this capstone is available at:

https://github.com/WashU-CSE4602-SP26/cse4602-team-amani-mack-capstone

The repository includes the final implementation as well as milestone-era directories preserved from the semester. The Final_release directory provides the cleaned final version, while the milestone folders document intermediate development stages and supporting implementation artifacts. Repository contents include HLS source files, board-side Python services and drivers, backend orchestration code, frontend components, and generated hardware artifacts such as bitstreams and hardware descriptors.

An earlier standalone client repository was also created during development at:

https://github.com/MackLiao/CapstoneClient

Its functionality was later incorporated into the main capstone repository as part of the final integrated frontend/backend system.