أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Systolic Arrays on a Heterogeneous MPSoC-FPGA Cluster

Students: Fadhlallah Almohammed, Sultanah Almutairi

fadhlallah.almohammed@kaust.edu.sa, sultanah.almutairi@kaust.edu.sa,

Supervisor: Maynor Ballina

mballina@ictp.it

,

Abstract

Systolic arrays accelerate general matrix multiplication (GEMM), the dominant operation in deep neural network inference, but conventional designs are confined to a single device. When the computation exceeds one device the accelerator must be distributed, and the cost of moving data between nodes becomes the factor limiting the gain from additional hardware. Prior cluster work routes that data through the processors of the nodes and reports the resulting loss, without evaluating an alternative. This work builds a parameterizable GEMM systolic array on the HyperFPGA heterogeneous MPSoC-FPGA cluster, where the programmable logic of adjacent nodes is connected directly, so the processor mediated path and the direct path can be compared on the same workload.
Five matrix multiplication formulations were compared on measured hardware cost, and the streaming design was selected at 29,114 cycles and 3,398 LUT. The array was then built on the programmable logic in three stages, each verified on hardware against a reference model: a transport structure confirming correct operand propagation at zero DSP, a compute stage carrying one multiply and accumulate unit per processing element, and a 2×2 grid of blocks in which each block learns its position from a packet header rather than from configuration, and in which all traffic shares one link in each direction so that a block boundary can become a node boundary without change. Every stage returned zero error, the grid at a median on node latency of 8.7 μs. The grid was then deployed across four nodes, with each link replaced by an Aurora 8B10B connection at 6.25 Gb/s and the block design left unmodified. The bitstream, the node mapping and the host procedure were verified, but the transceivers did not reach the operating state, since the 125 MHz reference does not reach the clock input of the bank, which is a platform setting outside user control. The two path comparison is therefore not completed.
The same computation was measured on the processor and on the graphics processing unit of the same node. One tile cost 1.133 ms on the CPU, 0.52104 ms on the GPU and 0.00025 ms on the FPGA, with the FPGA below both at every matrix size tested.

Distributed software experiments showed communication exceeding computation by up to an order of magnitude, with overlapped dispatch reducing total operation time from 43.843 s to 25.623 s. Data movement, not arithmetic, is the limiting cost on all three resources.

Date: August 25, 2026

Code: https://github.com/ICTP/Systolic-arrays-on-an-MPSoC-FPGA-Cluster

Contents

1 Introduction 2

2 Background 2

2.1 General Matrix Multiplication . . . 2
2.2 Systolic Arrays . . . 3
2.3 MPSoC and Heterogeneous Computing . . . 3
2.4 The HyperFPGA Cluster . . . 4

3 Related Work 4

3.1 Optimizing the Single Array . . . 4
3.2 FPGA Implementations . . . 5
3.3 Distributed FPGA Clusters . . . 5
3.4 Research Gap . . . 5

4 Methodology 6

5 Evaluation Procedure 8

6 Implementation 9

6.1 Selection of the Matrix Multiplication Algorithm . . . 9
6.2 Software Implementation of Systolic Array . . . 10
6.3 Hardware Implementation of the Systolic Array . . . 18
6.4 Deployment of the grid across four nodes . . . 25

7 Results 28

7.1 Worst case computation time and memory footprint . . . 29
7.2 Total operation time against matrix size . . . 29
7.3 Discussion . . . 30

8 Limitations & Future Work 30

8.1 Limitations . . . 31
8.2 Future Work . . . 31

9 Conclusion 31

References 32

1 Introduction

General matrix multiplication (GEMM) is the dominant operation in deep neural network inference, and the systolic array is one of the principal architectures used to accelerate it. Processing elements exchange operands only with their immediate neighbors, so an operand fetched once from memory is reused by many of them as it moves through the array, giving a high ratio of arithmetic to memory traffic [1]. Recent work improves the architecture within a single device by making the dataflow selectable per layer [2], shortening the critical path [3], adding reconfigurable precision [4], or matching heterogeneous processing elements to compact convolutional neural networks [5]. The paradigm has also been applied outside deep learning, for example to support vector machine classification on embedded devices [6]. Each of these assumes the computation fits on one chip, while the models that motivate the acceleration continue to grow beyond the capacity of a single device.

The case where the computation exceeds one device is largely unaddressed. Johnson et al. distribute a deep learning graph across a cluster of Zynq boards and report that adding nodes does not consistently reduce latency, attributing the loss to network bandwidth and to the involvement of the processor in moving data between boards [7]. The processor level path is identified as the limiting factor, but no alternative to it is evaluated.

The system built here is a GEMM accelerator for dense matrices, described as a parameterizable high level synthesis (HLS) systolic array and distributed across nodes of the HyperFPGA, a cluster of multiprocessor system on chip (MPSoC) devices, each combining a processor with field programmable gate array (FPGA) logic [8]. This platform offers the alternative that Johnson et al. lacked. The processors of the nodes communicate through a Gigabit Ethernet switch, while the programmable logic of adjacent nodes is connected directly through the carrier board, so an accelerator placed on the fabric can exchange operands without passing through either processing system. Comparing those two paths on the same workload, in throughput in tera operations per second (TOPS) and in cluster end to end latency, is the question this work is built to answer.

Four stages of that system were carried out. Five matrix multiplication formulations were compared in Vitis HLS on measured hardware cost, and the selected formulation was built on the programmable logic as a transport structure, then as a compute stage, then as a grid of four blocks whose interface is designed so that a boundary between blocks can become a boundary between nodes. Each of the three was verified against a reference model on hardware, returning zero error. The grid was then deployed across four nodes with the links between blocks replaced by Aurora connections over the board to board transceivers, and the same computation was expressed on the processor and on the graphics processing unit of the same node, so that the cost of moving data is measured on every resource rather than on the accelerator alone. The transceivers did not reach the operating state, for a reason traced to the provisioning of a reference clock outside user control, so the two path comparison is reported as not completed and the measurements presented are those of the single node.

2 Background

2.1 General Matrix Multiplication

GEMM multiplies an $M \times K$ input matrix by a $K \times N$ weight matrix to produce an $M \times N$ output [2]. It is the operation targeted for acceleration because the workloads of interest reduce to it. A neural network layer, whether a two dimensional convolution, a fully connected layer, a long short-term memory (LSTM) network or a multilayer perceptron (MLP), is transformed into one

or more GEMM operations before execution [2], and the MLP blocks built from such layers account for between 40 and 60 percent of the total computational cost in Transformer models [9]. Accelerating GEMM therefore accelerates the dominant part of these workloads rather than a single special case.

A hardware array has a fixed size, while GEMM dimensions are set by the workload and are usually larger. When the matrices exceed the size of the array, they are divided into tiles, and the tiles are mapped onto the array one after another until all outputs are produced [2]. Tiling makes a fixed array usable for arbitrary problem sizes, and it also makes distribution across several arrays possible, since independent tiles can be assigned to different arrays.

2.2 Systolic Arrays

A systolic array is a grid of processing elements, each containing a multiply and accumulate unit and local registers. Every processing element communicates only with its immediate neighbors. Operands enter at the boundary of the array and propagate rhythmically from one processing element to the next, so a value fetched once from memory is reused by many processing elements as it travels [10]. Figure 1 shows the structure.

The consequence is a high ratio of operations to memory accesses. Because data is passed between neighbors rather than fetched again, the array raises the number of operations performed per memory fetch and reduces off-chip memory traffic [10]. This property is the reason the architecture is suited to GEMM, where the same operands are needed repeatedly.

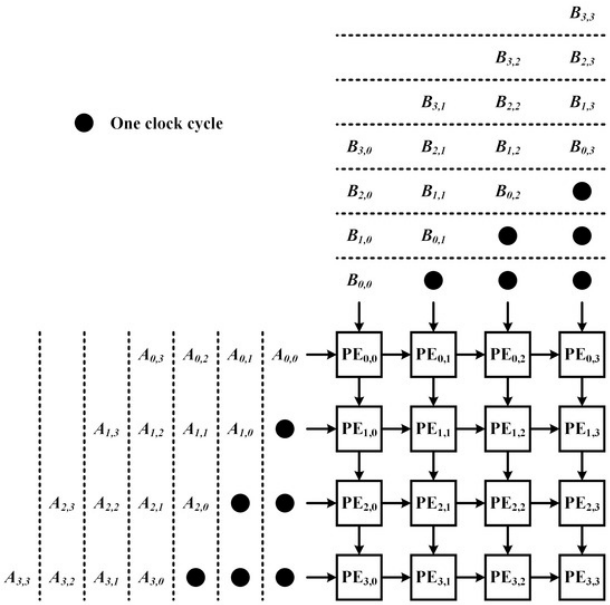


Figure 1 Structure of a systolic array. Processing elements exchange operands with their immediate neighbors, while data enters and leaves only at the boundary of the array.

2.3 MPSoC and Heterogeneous Computing

An MPSoC combines a hard processor and programmable logic on a single device. In the Zynq UltraScale+ family the processing system (PS) contains a quad-core ARM Cortex-A53 application processor, a dual-core Cortex-R5F real time processor, a Mali-400 MP2 graphics processing unit (GPU) and a memory controller, while the programmable logic (PL) contains lookup tables, flip flops, block RAM, UltraRAM and digital signal processing (DSP) slices.The two are connected by Advanced eXtensible Interface (AXI) ports.

Heterogeneity here means that four processing resources with different execution models are available on the same device. The same computation can therefore be expressed on more than one resource and compared, and a workload can be assigned to the resource suited to it.

The nodes used in this work are built on the XCZU4EG device, which provides 728 DSP slices, 87,840 lookup tables, 175,680 flip flops and 128 block RAM tiles of 36 Kb in the programmable logic.

2.4 The HyperFPGA Cluster

A cluster is a set of independent nodes connected by a network and operated as one machine. In a heterogeneous cluster each node is itself heterogeneous, so a workload is distributed twice, once across the nodes and once within each node between its processing system and its programmable logic.

The HyperFPGA is an MPSoC-FPGA cluster developed at ICTP for research on computing paradigms [8]. In its current form it consists of 16 nodes, six built on UltraScale+ MPSoC ZU4EG devices and ten on ZU3EG devices, each node being a system on module mounted on a carrier board, as shown in Figure 2. The cluster is reached through a JupyterHub server that forwards sessions to individual nodes, and tasks are distributed from a central server over Secure Shell (SSH) using `ipyparallel`. The work reported here targets a ZU4EG node.



Figure 2 The HyperFPGA cluster. Adapted from [8].

3 Related Work

3.1 Optimizing the Single Array

A dataflow determines which of the three matrices remains resident in the processing elements while the other two move through them [10]. Weight Stationary preloads the weights and lowers

memory accesses but requires synchronization first in first out (FIFO) buffers, Input Stationary holds the input at the cost of higher weight bandwidth, Output Stationary accumulates partial sums locally but doubles the required memory bandwidth, and Row Stationary broadcasts operands at the cost of data duplication. Since no scheme is optimal for every workload, ReDas makes the choice adaptive per layer and reports a measurable reduction in execution time over a fixed 128×128 Weight Stationary array [2].

The shape of the array is a second variable of the same kind. An array that is too large or too symmetrical for its workload leaves processing elements idle [11], and once a matrix exceeds the array it must be tiled, so the shape also determines the resulting traffic [2]. ReDas reshapes the logical array at fine grain, HeSA matches heterogeneous processing elements to the depthwise convolutional layers of compact convolutional neural networks [5], and Fierro reports the area cost of supporting multiple configurations in a 28 nm flow [9].

Precision is the third variable. ADiP proposes reconfigurable processing elements supporting 8b×8b, 8b×4b and 8b×2b operations, reaching 32.768 TOPS at a 64×64 configuration for an area and power overhead of up to 2.3× [4]. The gains are workload dependent, with a 53.6% latency improvement on BitNet-1.58B attention but a 62.8% energy overhead on 8-bit GPT-2 medium [4]. Sanchez-Flores et al. address the measurement side, noting that the quantization literature reports power in inconsistent units, and measure a quantized accelerator on an Ultra96-v2 board by separating the processing system and programmable logic power domains [12].

Dataflow, shape and precision are three means to the same end, which is reducing the volume of data crossing the boundary of the array. All are evaluated on a single device.

3.2 FPGA Implementations

The works above target application specific integrated circuit (ASIC) technology, and their area, power and timing results are not directly comparable with FPGA implementations. A smaller body of work targets FPGA. MEISSA implements a stationary array on a Zynq XC7Z020, separating the multipliers from the accumulation logic and using a logarithmic depth adder tree to shorten the critical path, described in C and C++ with pragmas and synthesized with Vivado HLS [3]. Ramadurgam and Perera apply the paradigm outside deep learning to support vector machine classification, achieving a 107× speedup over a software baseline and 98.5% classification accuracy on an FPGA-based systolic array [6]. As a throughput reference, BISMO reports 6.5 TOPS at 1.4 TOPS/W on a PYNQ-Z1 and 15.4 binary TOPS under 8 W on an Ultra96 [13]. These results establish HLS as a viable description method for systolic arrays on Xilinx FPGA devices, but each remains confined to a single device.

3.3 Distributed FPGA Clusters

Few works distribute an accelerator across multiple FPGAs. Johnson et al. build a cluster of up to 12 Zynq-7020 boards and five Zynq UltraScale+ MPSoC boards connected through a 1 Gb/s Ethernet switch, and distribute a ResNet-18 graph across nodes running the Versatile Tensor Accelerator [7]. Adding nodes did not consistently reduce latency, and the authors attribute the loss to network bandwidth and to the involvement of the processor in transmitting data packets between devices. Their finding identifies the processor level path as the limiting factor but does not evaluate an alternative, since the platform used offers none.

3.4 Research Gap

Two observations follow from the above. First, the architectural literature optimizes a single array within a single device and is silent on the case where the computation exceeds one device. Second,

the cluster literature distributes the computation but routes the data through the processors of the nodes, and reports this as the factor limiting the gain from additional devices [7].

This work addresses the space between them. A single large matrix multiplication is distributed across a systolic array spanning several nodes of an MPSoC-FPGA cluster, with the objective of making the exchange of data between nodes as fast as the platform permits. The array is held fixed as a parameterizable HLS description, while the communication path between nodes is the variable under study. The available paths are compared using throughput in TOPS, the metric reported by the architectures reviewed above, so that the cost of moving data between nodes can be separated from the cost of computing within them.

4 Methodology

The accelerator is a parameterizable HLS systolic array performing GEMM, distributed across nodes of the HyperFPGA cluster [8]. The array is held fixed as a single design, and the path used to move tiles of the input and output matrices between nodes is the independent variable.

The same computation is expressed on the three processing resources of the node, so that the hardware implementation is compared against software references measured on the same device. Every implementation is developed and deployed through the flow shown in Figure 3.

1. A reference model is written in Python and NumPy. It defines the expected output and is the correctness reference for all three platforms.
2. The computation is then expressed on each resource in turn.
    a. On the CPU the systolic array is modeled in Python and executed on the ARM Cortex-A53 cores, one engine per core.
    b. On the GPU the computation is expressed through the graphics pipeline, with the operands held as images, and executed on the Mali-400 MP2.
    c. On the FPGA the array dimensions and the dataflow are first profiled in SCALE-Sim, so that they are chosen from evidence rather than assumption. The kernel is then written in C++ and synthesized in AMD Vitis HLS, applying directives for latency and resource targets and verifying with C simulation and C/RTL co-simulation before exporting the IP. The exported IP is integrated in Vivado alongside the processing system and one or more AXI DMA engines, and packaged into a bitstream.
3. The implementation is deployed to a single node, driven from Python, and its output is checked against the reference model.
4. The workload is distributed to multiple nodes from a central server, and throughput and latency are measured across the cluster.

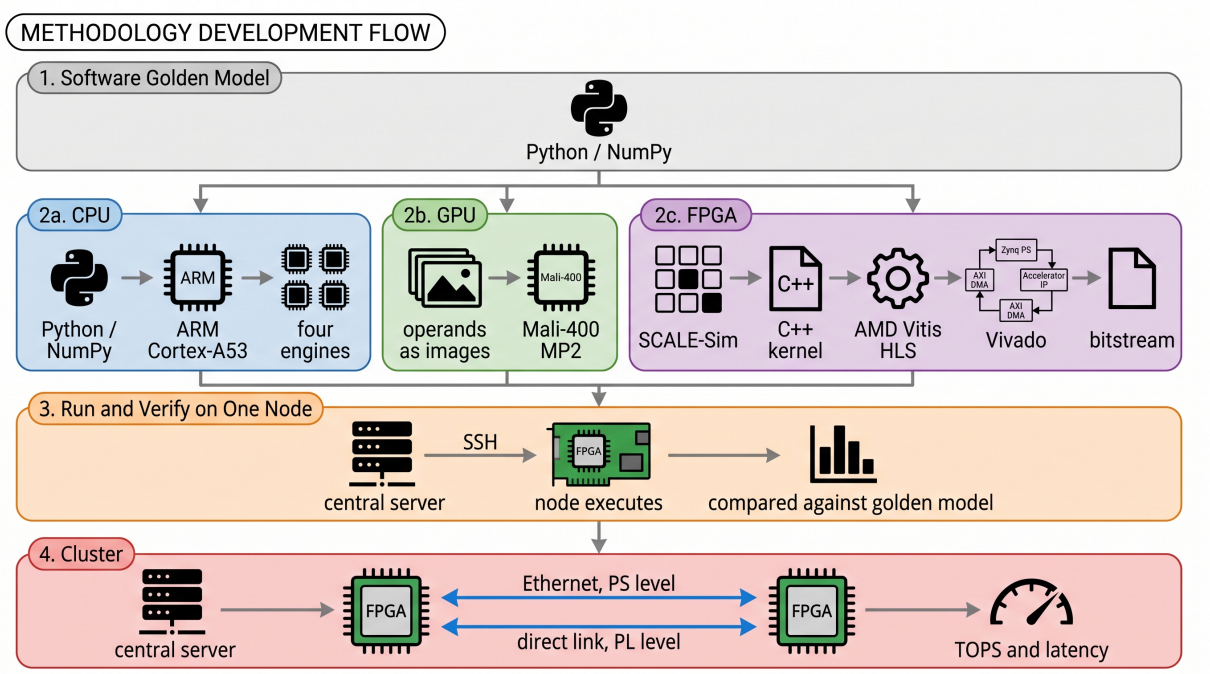


Figure 3 Methodology development flow. A single reference model in Python and NumPy serves the three platforms.

The hardware implementation follows the same principle within itself. The array is built in four stages, shown in Figure 4, each adding one mechanism to the one before it, so that a fault can be attributed to what was added. The digital signal processing count reported by synthesis identifies which stage is present, since it is zero where the processing elements only move operands and one per processing element where they compute.

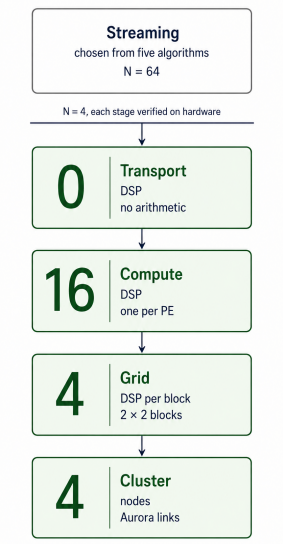


Figure 4 The array is built in four stages. The digital signal processing count identifies the stage, being zero in transport, one per processing element in compute and four per block in the grid. The first three stages were verified on hardware. The fourth was deployed across four nodes, where the links did not reach the operating state.

The three platforms are reached in the same way at node level. Work is dispatched from a

central server over SSH using `ipyparallel`, and results are returned over the same connection. They differ in how data is exchanged between nodes. The CPU and the GPU are reachable only through the processing system, so both use the Gigabit Ethernet network, over which tiles and results travel as SSH traffic between the central server and the nodes. The programmable logic is not restricted to that route, since the fabric of adjacent nodes is connected directly through the carrier board [8]. On the FPGA the exchange between nodes is therefore carried by an Aurora link over the board to board transceivers, without passing through either processing system. These two routes, the processor level path shared by the CPU and the GPU and the programmable logic level path available only to the FPGA, are the paths compared in this work.

Figure 5 summarizes the arrangement. The workload reduces to a matrix product on all three platforms, the same product is expressed once on each resource, and the resource determines the route taken between nodes.

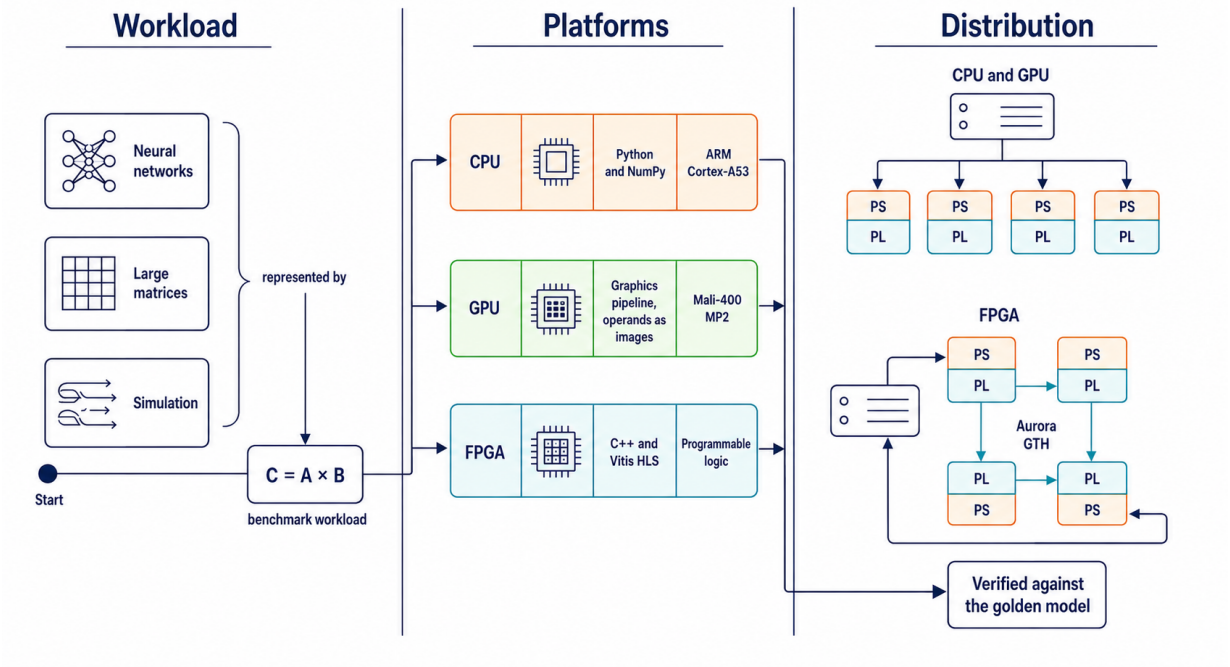


Figure 5 The same matrix product is expressed on the three resources. The CPU and the GPU are reached through the processing system and exchange data over Ethernet, while the programmable logic of adjacent nodes exchanges operands directly over Aurora. Every result is checked against the reference model.

5 Evaluation Procedure

The workload is a set of synthetic dense matrices at several sizes, generated to exceed the array of a single node so that tiling and distribution are exercised. Every run is checked against the reference model before any performance number is recorded, and the maximum absolute error against that model is reported alongside the timing.

Table 1 states what is recorded on each platform. On the CPU and the GPU the algorithm time is the time spent computing and the communication time is the time spent moving operands and results between the central server and the node, and the two are separated so that the fraction of the total spent moving data rather than computing is visible. The FPGA is measured at three levels. Co-simulation gives the compute alone. The single node measurement adds the transfer across the PS-PL boundary, and records the on node end to end latency, defined as the interval

Table 1 Quantities recorded on each platform.

Platform	Level	Quantities recorded
CPU	Node	Algorithm time, communication time, total operation time, maximum peak memory
GPU	Node	Algorithm time, communication time, total operation time, maximum worst case memory footprint
FPGA	Co-simulation	Latency in cycles, LUT, FF, BRAM, DSP
	Single node	On node end to end latency
	Cluster, Aurora	Cluster end to end latency, throughput in TOPS, communication time separated from compute time

from the first operand entering the accelerator through the AXI DMA engine to the last result leaving it through the same path. The cluster measurement adds the transfer between nodes. Each level therefore isolates one further cost.

6 Implementation

6.1 Selection of the Matrix Multiplication Algorithm

Before the systolic array was written, five matrix multiplication algorithms were implemented and compared, so that the formulation carried into the hardware design would be selected from measured cost rather than from asymptotic complexity. Table 2 states their asymptotic cost and their on chip working set. Each was first profiled in software and then synthesized in Vitis HLS.

Table 2 The five algorithms compared. Memory footprint is the working set held on chip at one time, in words.

Algorithm	Time complexity	Memory footprint
Standard	$O(n^3)$	$3n^2$
Strassen	$O(n^{2.807})$	$\sim O(n^2 \log n)$
Block / Tiled	$O(n^3)$	$3n^2 + 3\,\text{tile}^2$
Winograd	$\sim O(n^{2.81})^\dagger$	$3n^2 + 2n$
Streaming	$O(n^3)$	$3\,\text{tile}^2$

†Winograd reduces the number of multiplications, not the total operation count.

6.1.1 Software implementation

The five algorithms were implemented in Python and run on square matrices from $n = 4$ to $n = 128$. Wall clock time and peak memory were recorded for every size.

Standard was the fastest at every size tested, despite having the highest operation count, and peak memory was close to identical across the algorithms, since all of them allocate the full $n \times n$ output on the heap. The software profile therefore does not separate the algorithms. They are separated instead by the working set held on chip at one time, which the Python measurement does not expose. Standard requires $3n^2$ words, 192 kB at $n = 128$ in 32 bit words, whereas Streaming requires $3\,\text{tile}^2$ words, 3 kB at a tile size of 16. Streaming was therefore the candidate carried into synthesis.

6.1.2 Hardware implementation

Each algorithm was described in C++ and synthesized in Vitis HLS 2024.2. For each one the dominant bottleneck was identified from the synthesis report and only the directive attacking that bottleneck was applied.

Table 3 reports the synthesis results. In its deployed configuration Streaming is both the fastest at 29,114 cycles and the smallest in logic at 3,398 LUT and 4,944 FF, and it is the only configuration to close timing with positive margin, estimating 3.104 ns against the 3.33 ns target. The comparison against the optimized Block configuration isolates the reason. Both perform the same 16 wide multiply and accumulate on the same 16×16 tiles and differ only in that Streaming overlaps the tile loads with the compute stage, and Block completes in 42,857 cycles against 29,114.

Table 3 Synthesis results, all configurations are $N = 64$ except Strassen, which is $N = 8$. Configuration names the directive set applied to the algorithm.

Algorithm	Configuration	LUT	FF	BRAM	DSP	Latency (cycles)
Standard	baseline	4,464	3,278	30	4	270,369
	optimized	14,829	21,691	142	256	12,350
Strassen	$N = 8$	7,980	5,408	6	4	789
Block	baseline	17,000	54,299	6	4	404,649
	optimized	21,176	50,880	6	64	42,857
Winograd	three pass	11,191	13,678	6	12	266,287
Streaming	baseline	84,437	159,485	0	4	391,514
	unrolled	88,671	172,515	0	64	29,786
	deployed	**3,398**	**4,944**	34	64	**29,114**

The synthesis results also revise the ordering suggested by the operation counts of Table 2. Strassen performs fewer multiplications than Standard but uses the same four DSP slices, since the count of multiplications is the number of times a shared multiplier is invoked rather than the number of multiplier circuits instantiated. Winograd halves the multiplications but reads four array elements per iteration against two available memory ports, which cancels the saving in full. In both cases the operation count is not the quantity that determines hardware cost. Streaming was selected on the measured result of Table 3, and is the formulation used in the systolic array reported in the following subsections.

6.1.3 Deployment of the selected core

The streaming core was exported to the IP catalog and integrated in Vivado with the processing system and two AXI DMA engines, as shown in Figure 6. The two DMA read channels drive the two input streams, and the output stream returns through the second DMA. The bitstream was deployed to a HyperFPGA node and driven from a Jupyter notebook.

A maximum error of zero was returned against a NumPy reference across all 4,096 elements of the 64×64 output.

6.2 Software Implementation of Systolic Array

6.2.1 Overview and Scope

This section characterizes communication overhead in a distributed, tiled systolic array simulation across two compute backends available on each HyperFPGA node: the quad-core ARM processors and the on-board Mali-400 GPU. Both backends were driven from a Jupyter notebook using `ipyparallel`, a Python library for dispatching work to remote processes, and both were evaluated

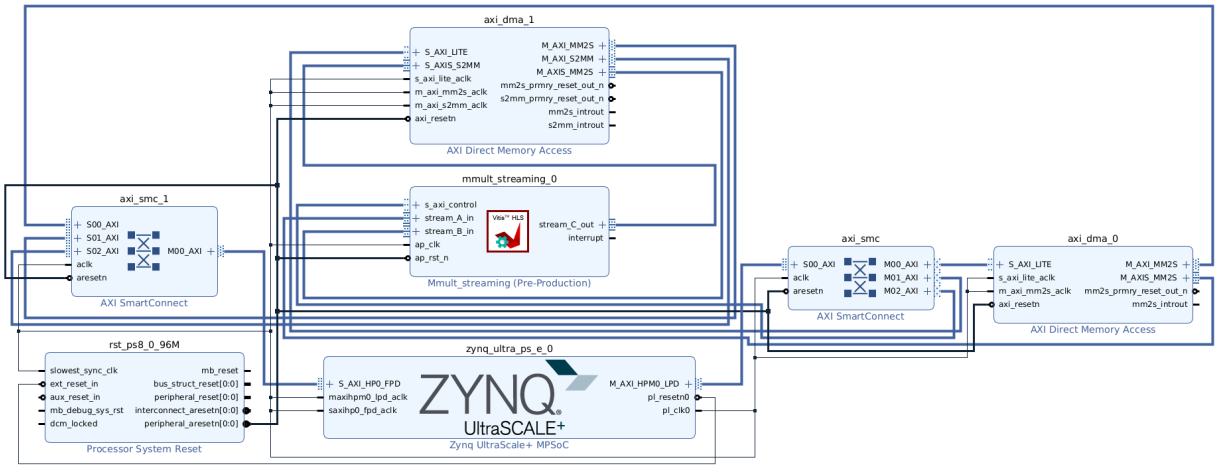


Figure 6 Vivado block design of the streaming matrix multiplication core with two AXI DMA engines.

against a single question: how does communication overhead scale relative to the computation being performed, as a function of tiling strategy, dispatch pattern, and cluster topology?

The remainder of this section first explains a platform limitation that shaped the choice of distribution tool (Section 8), then presents the CPU-side experiments (Sections 6.2.3–6.2.7) before turning to the GPU-accelerated path (Sections 6.2.8–6.2.10).

6.2.2 Limitations: Choosing How Engines Communicate

MPI (Message Passing Interface) was evaluated first as the mechanism for coordinating work across a node's four cores, since it is the conventional choice for this kind of distributed numerical computation. On this cluster, however, every core failed with the same communication error.

Investigation traced the cause to how the cluster starts each core's process. Rather than launching all four cores together as a single coordinated job, the cluster's engine launcher starts them one at a time, each over its own SSH connection. As a result, each core comes up as an isolated process with no awareness that three sibling cores exist. MPI depends on all participating processes being launched together so they can recognize one another as members of the same communication group; when that condition is not met, any attempt to divide work among them fails, because there is no group for MPI to coordinate.

This is a limitation of the cluster's launch mechanism rather than an error in the code being run, and it motivated the adoption of `ipyparallel` as the distribution layer for all subsequent work in this project. Unlike MPI, `ipyparallel` does not require its engines to be aware of one another: all coordination happens through the controller, which pushes data to engines and collects results back individually.

No direct engine-to-engine communication exists anywhere in this project. This constraint defines the shared architecture used throughout the remainder of the section, described next.

6.2.3 Shared Architecture

Every experiment uses the same topology: the controller process, running in the notebook, dispatches work to a set of `ipyparallel` engines over SSH and collects the results. Engines never communicate with one another directly. This hub-and-spoke structure applies equally to the CPU and GPU tracks that follow, differing only in what each engine does with the data it receives.

With the distribution mechanism established, the next step was to generalize the systolic array simulation itself, which had initially been implemented only for a fixed 4×4 case.

6.2.4 Generalizing the Systolic Array

The array simulation was generalized to accept matrices of any depth and width, so that the array's dimensions no longer needed to match one another. Each engine was also modified to process however many rows it received, rather than assuming exactly one, so the simulation remains correct when the number of rows does not divide evenly across the four engines.

Across four engines, baseline process memory totaled 282.02 MB, with an additional 40.50 MB attributable to importing the numerical library used for the computation. Against this, the array computation itself cost 2.812 KB for a 4×4 case and 8.106 KB for a wider 4×8 case — a gap of roughly five orders of magnitude between the infrastructure carrying the simulation and the computation it performs. Both cases reproduced the correct matrix product exactly.

This generalized array, however, still grew to match whatever matrices it was given — a convenience of simulation with no counterpart in physical hardware. The following experiment removes that convenience by fixing the array's size independently of the problem.

6.2.5 Tiling for a Fixed-Size Array

A physical systolic array has a fixed number of processing elements, decided when the hardware is built; it cannot grow to accommodate a larger problem. To reflect this constraint, the array's dimensions were fixed independently of the matrices being multiplied, and matrices exceeding the array's size were decomposed into tiles dispatched one at a time.[1]

Three kinds of split are required, for distinct reasons. Splitting a matrix's rows or columns produces separate, non-overlapping pieces of the final result, which are assembled side by side once computed. Splitting the shared middle dimension is different: every piece from that split contributes to the *same* output values, so the resulting partial results must be summed rather than placed side by side.

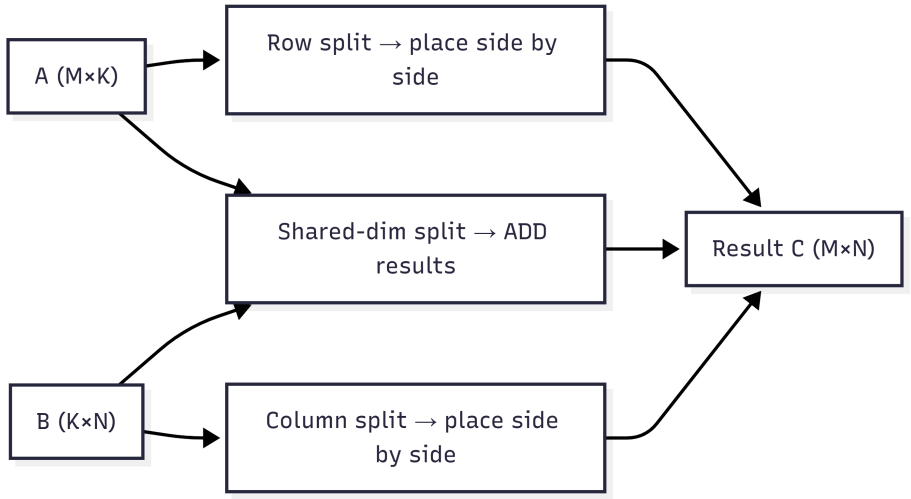


Figure 7 Decomposition of a tiled matrix multiplication: row and column splits are concatenated; splits along the shared dimension are summed.

[1] In the tiling parameters $\text{arr}_M \times \text{arr}_K \times \text{arr}_N$, only arr_K (array depth) and arr_N (array width) are physical constraints on the systolic array itself; arr_M controls how many rows of A are batched to each engine per tile and is not a physical array dimension.

At a $20 \times 40 \times 100$ matrix multiplication against a fixed $4 \times 6 \times 8$ array (Figure 7), the problem required 455 separate tiles. Dispatching them one at a time — waiting for each tile's result before sending the next — produced a total operation time of 43.843 s, of which 41.589 s, roughly nineteen times the compute time, was spent on communication. The per-tile communication cost closely matched the cost of a single, un-tiled round trip measured in an earlier experiment, confirming that the overhead scales with the number of separate trips made, not with the total volume of data transferred.

This result raised a natural question: since real streaming hardware avoids exactly this kind of serialized wait, could the same idea be applied in software to reduce the communication cost observed here?

6.2.6 Overlapped (Streaming) Dispatch

FPGA streaming implementations avoid waiting for one data transfer to complete before beginning the next, using double buffering: while the array computes using data already loaded, the next tile's data is loaded into a second buffer in parallel. This technique was reproduced in software using two alternating buffers and a non-blocking dispatch call, so that tile i+1's data is sent while tile i is still being computed (Figure 8). Alternating between two buffers ensures that a buffer is never overwritten while its contents are still in use.

Table 4 Blocking versus 2-stage overlapped dispatch, 455 tiles.

	Standard (blocking)	2-stage overlap
Algorithm time	2.254 s	1.640 s
Communication time	41.589 s	23.983 s
Total operation time	43.843 s	25.623 s

Applying this overlap to the same 455-tile workload reduced total operation time from 43.843 s to 25.623 s, a reduction of approximately 40%, with the correctness of the result unchanged. Having established that dispatch strategy reduces communication cost, attention turned next to whether the physical placement of engines across the cluster's boards could offer a further, independent improvement.

6.2.7 Topology and Array Size

Engine placement and array size were examined together using the overlapped dispatch strategy, sweeping array size at a fixed matrix size of $20 \times 40 \times 100$, with four engines placed either on one board or across four separate boards.

Table 5 Total operation time versus array size, one board versus four boards, four engines in both cases.

Array size (tiles)	1 board (s)	4 boards (s)	Improvement
$4 \times 8 \times 20$ (125)	10.89	9.22	15.4%
$6 \times 12 \times 30$ (64)	8.00	7.26	9.3%
$8 \times 16 \times 40$ (27)	4.71	4.28	9.2%
$10 \times 20 \times 50$ (8)	3.72	3.58	3.7%
$20 \times 40 \times 100$ (1)	2.85	2.73	4.1%

Two patterns emerge together. First, distributing engines across separate boards improves total time at every array size tested (Table 5), with the advantage largest where communication forms

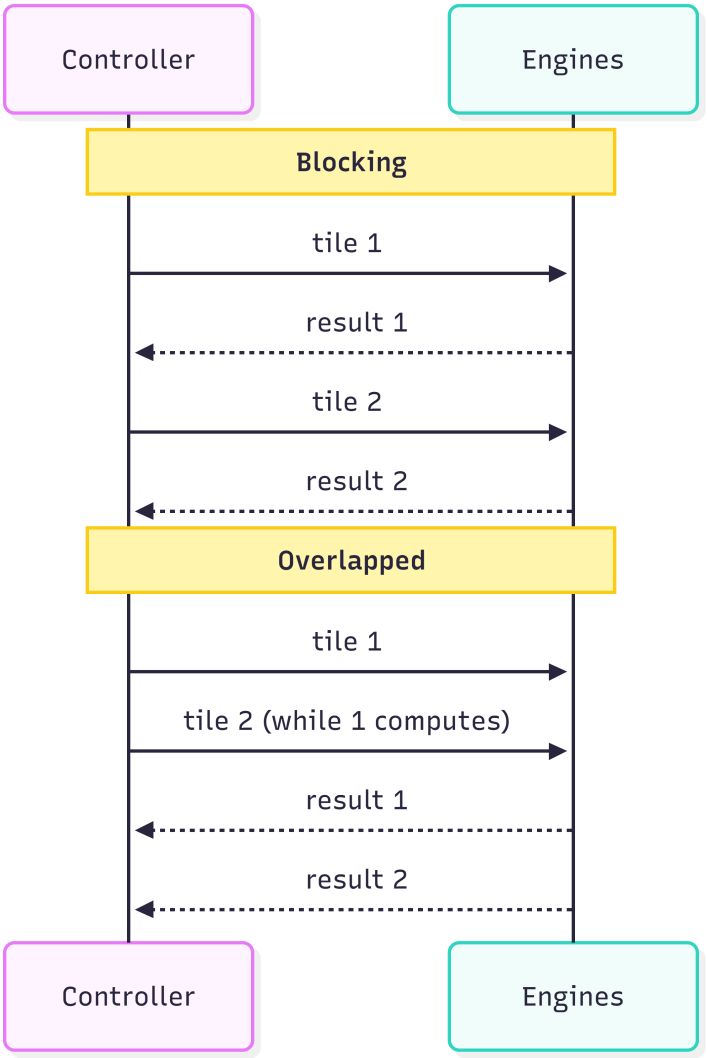


Figure 8 Blocking versus overlapped dispatch: in the overlapped case, the next tile is sent while the current tile is still being computed.

the greatest share of total time and diminishing as tile count falls — consistent with contention on a shared network interface, relieved when engines use independent connections. Second, communication time itself falls sharply with array size, from 6.80 s at 125 tiles to 0.34 s at a single tile, while algorithm time remains largely unchanged throughout the sweep, between roughly 2.4 and 4.1 s (Figure 9). This reflects that the total volume of multiply-accumulate work does not depend on how it is partitioned; only the number of round trips does. Beyond the point where the array matches the matrices themselves, further enlargement of the array provides no additional benefit, since both configurations reduce to a single tile.

The CPU experiments established that communication overhead, rather than computation, is the dominant cost in this distributed simulation, and that this overhead can be reduced through dispatch strategy and engine placement.

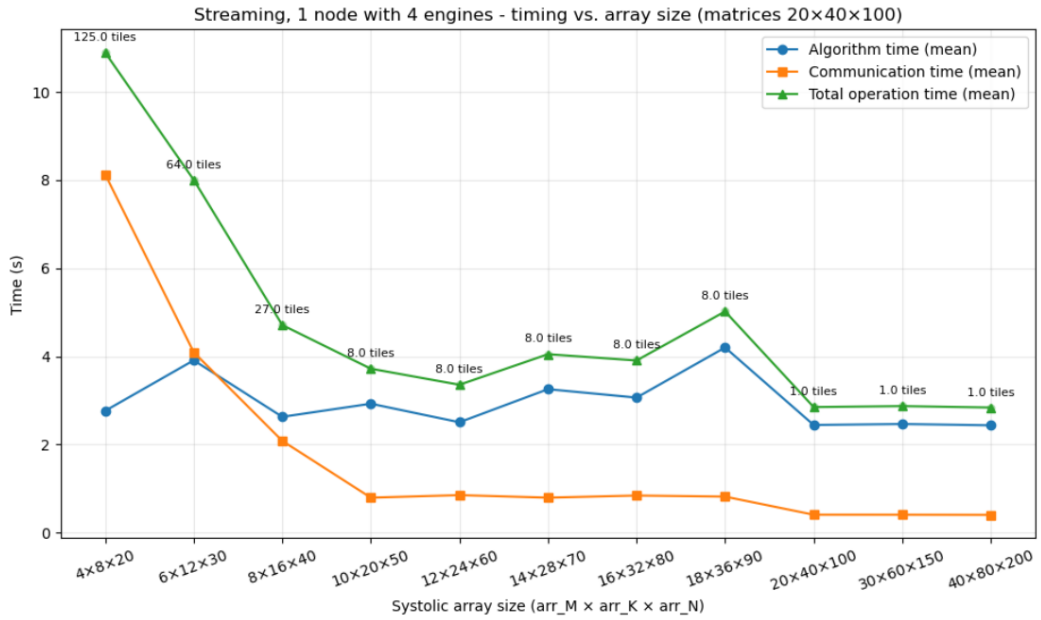


Figure 9 Algorithm, communication, and total time versus array size, one board with four engines.

6.2.8 GPU-Accelerated Tile Computation

The software-simulated per-engine computation was replaced with a version that dispatches each tile to the board's GPU, which performs the multiplication as a single hardware operation rather than a simulated cycle-by-cycle loop. The surrounding tiling logic was left unchanged, so any difference in behavior can be attributed to the compute backend rather than the distribution strategy.

Because the GPU path encodes numerical values using a compressed color format, results carry a rounding error noticeably larger than the CPU path's floating-point precision, in the range of approximately 0.2 to 5.7 across the configurations tested. This error remained consistent across dispatch strategies, confirming it as an inherent property of the GPU computation rather than of how tiles were sent to it, and is an accepted characteristic of the GPU generation used on this platform rather than an implementation defect.

Table 6 Two-engine GPU baseline, one tile per round trip.

Array size	Tiles	Algorithm (s)	Communication (s)	Total (s)
$4 \times 8 \times 20$	125	0.885	5.072	5.957
$6 \times 12 \times 30$	64	0.473	2.739	3.212
$8 \times 16 \times 40$	27	0.225	1.180	1.405
$10 \times 20 \times 50$	8	0.095	0.374	0.469

Communication accounted for roughly 80–85% of total time across this sweep (Table 6), a less favorable ratio than the CPU path, because the GPU computes each tile in under one millisecond, leaving proportionally more of every round trip as overhead. This finding motivated the same amortization strategy already shown effective on the CPU path.

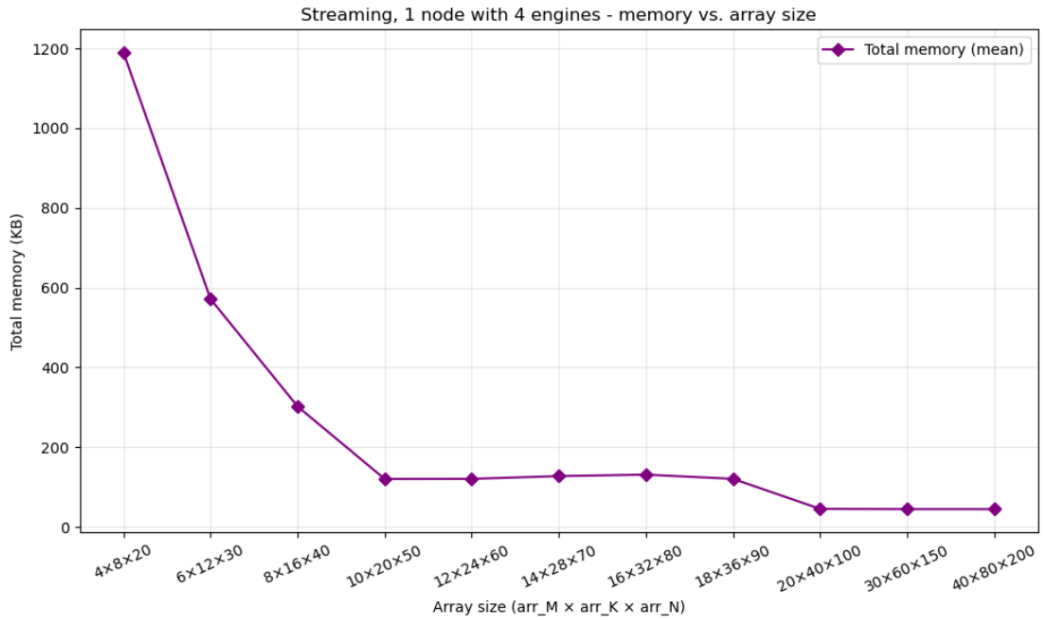


Figure 10 Total memory versus array size, one board with four engines.

6.2.9 Batched Dispatch

Multiple tiles were grouped into a single dispatch, with each engine iterating over its assigned tiles before returning all results together, so that the fixed cost of each round trip could be spread across more work (Figure 11).

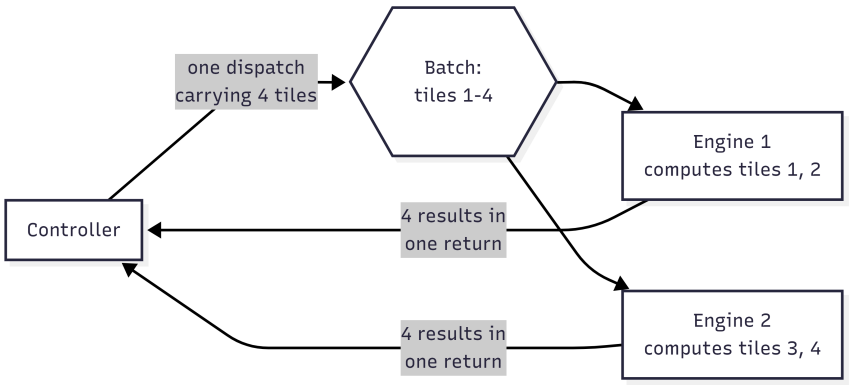


Figure 11 Batched dispatch: a single round trip carries several tiles, split across engines, with all results returned together.

An initial implementation of this grouping contained a defect: one component of a tile's data was correctly divided across engines, while the remaining components were sent, unmodified, to every engine. This produced mismatched data whenever a group of tiles crossed a boundary between differently sized pieces of the tiling pattern. The defect was identified when two of five test configurations failed outright, and confirmed by a marked reduction in rounding error at a configuration that had previously appeared to succeed. The correction bundles every component of a tile together before distribution, ensuring the pieces cannot become separated regardless of how tiles are grouped.

Grouping four tiles into each dispatch reduced total operation time by a factor of roughly three across every configuration tested (Table 7), with correctness and compute time unaffected. This

Table 7 Two engines, one tile per dispatch versus four tiles per dispatch.

Array size	Tiles	Unbatched (s)	Batched (s)	Speedup
$4 \times 8 \times 20$	125	5.957	1.815	3.28×
$6 \times 12 \times 30$	64	3.212	0.937	3.43×
$8 \times 16 \times 40$	27	1.405	0.439	3.20×
$10 \times 20 \times 50$	8	0.469	0.175	2.68×

established batching as an effective remedy on two engines; the next question was whether the same fix would resolve a scaling limitation observed when additional engines were introduced.

6.2.10 Multi-GPU Scaling

Before batching was introduced, increasing the number of engines from two to four made total time worse rather than better at matching problem sizes. A closer examination ruled out one plausible explanation — that some engines were left idle with no assigned work — since configurations in which every engine had genuine work still showed the same regression. This pointed instead to a fixed coordination cost incurred each time the controller communicates with an engine, a cost that accumulates faster as more engines are added.

Applying batching resolved this. Holding the workload per engine constant at two tiles per round trip in both configurations, doubling engine count produced a consistent reduction in total time:

Table 8 Two engines versus four engines, equal workload per engine per round trip.

Array size (tiles)	2 engines (s)	4 engines (s)	Speedup
$4 \times 8 \times 20$ (125)	1.815	1.107	1.64×
$6 \times 12 \times 30$ (64)	0.937	0.498	1.88×
$8 \times 16 \times 40$ (27)	0.439	0.330	1.33×
$10 \times 20 \times 50$ (8)	0.175	0.123	1.42×

The gain is real but sub-linear: batching reduces coordination cost without eliminating it entirely, so doubling engine count does not fully double throughput. A further sweep tested whether increasing the workload per round trip beyond two tiles per engine would close this gap.

Table 9 Four engines: two tiles per engine versus four tiles per engine.

Array size (tiles)	Batch = 8 (s)	Batch = 16 (s)	Change
$4 \times 8 \times 20$ (125)	1.107	0.800	28% faster
$6 \times 12 \times 30$ (64)	0.498	0.393	21% faster
$8 \times 16 \times 40$ (27)	0.330	0.221	33% faster
$10 \times 20 \times 50$ (8)	0.123	0.131	6.5% slower

Increasing the workload per round trip further improved total time at every configuration except the smallest, where only eight tiles are distributed across four engines (Table 9, Figure 12). There, a batch size of sixteen exceeds the available work, so no further round trips remain to eliminate, and the added grouping overhead is not repaid. This indicates that batch size should

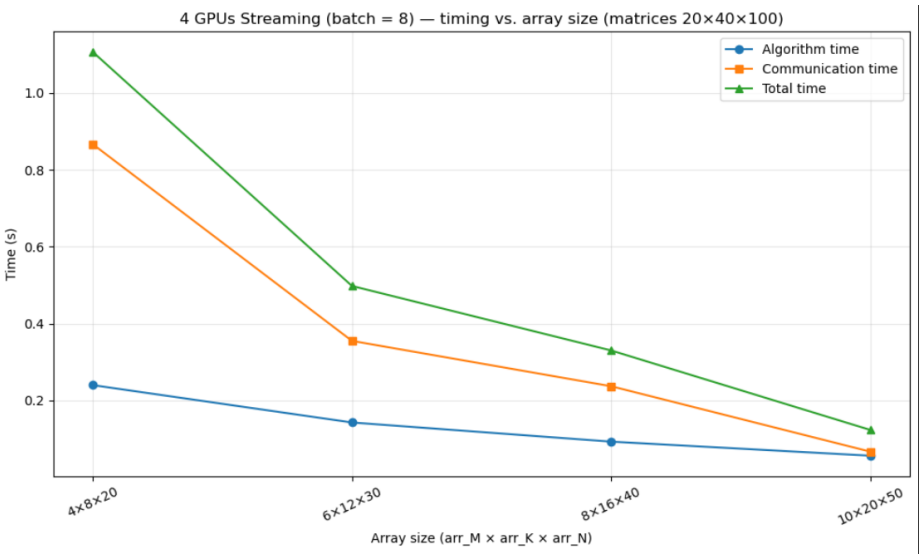


Figure 12 Algorithm, communication, and total time versus array size, four engines, two tiles per engine (eight tiles per round trip).

be chosen relative to the total number of tiles: larger batches are beneficial only up to the point where each engine still receives multiple tiles to group.

The largest single memory footprint observed on any engine also increased with batch size, reflecting that a larger batch holds more tiles' data resident at once — a real and expected tradeoff for the communication savings it provides.

6.2.11 Summary

Table 10 Cross-cutting summary of CPU and GPU findings.

Finding	CPU	GPU
Communication dominates computation	up to 19× compute time	up to 90× compute time
Overlap/batching reduces communication	~40% (overlap)	up to 3.4× (batching)
Added hardware requires prior amortization	4–15% gain from placement	regression before batching, 1.3–1.9× after
Batch size has a problem-size-dependent optimum	—	up to 33% further gain, or a small loss if oversized

Across both independently implemented compute backends, the evidence points to a single, consistent conclusion: dispatch that sends one tile at a time and waits for each to complete is dominated by the fixed cost of each round trip, and this cost must be actively amortized — through overlapped dispatch, tile batching, or both — before additional parallel hardware provides a net benefit. Left unaddressed, this overhead can cause additional hardware to measurably worsen performance, a regression directly observed and subsequently resolved on the GPU path.

6.3 Hardware Implementation of the Systolic Array

The array was implemented on the programmable logic in three stages. In the first the processing elements move operands but perform no arithmetic, so that the transport structure can be verified independently of the computation. In the second each processing element performs the multiply and accumulate. In the third the array is partitioned into four blocks whose interface is designed so that a boundary between blocks can later become a boundary between nodes.

6.3.1 Transport stage

A 4×4 Output Stationary array was described in HLS. Every operand carries a tag identifying the row or column it came from, so that any value recovered from the array can be traced back to its origin. The behavior of a single processing element is shown in Figure 13. A read from an empty stream blocks, so a processing element cannot proceed until both of its operands are present, and the diagonal skew of the array follows from the channel semantics rather than being applied explicitly. Operands enter and results leave through AXI-Stream interfaces at the boundary.

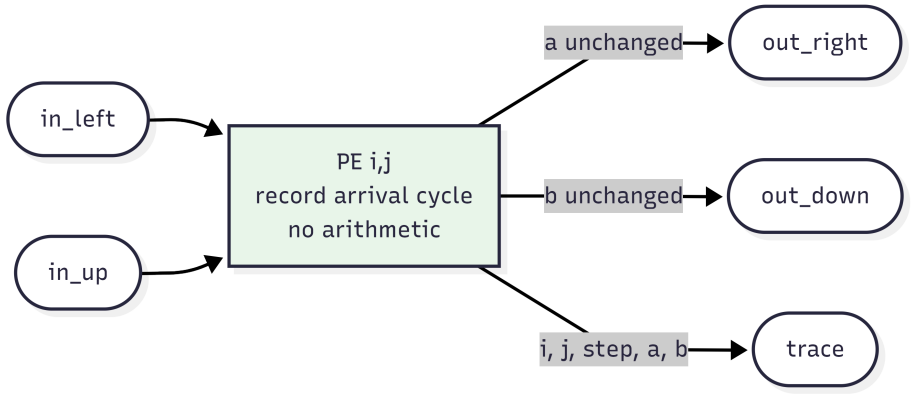


Figure 13 A processing element in the transport stage. Both operands are forwarded unchanged, the arrival is recorded, and no arithmetic is performed.

The array is built from registers and unrolled, so it scales by changing N alone. Each row and each column is delayed on entry so that operands meet at the intended processing element, and the step counter is derived from the cycle and the position of the processing element rather than from the operand values, so it does not drift on data.

Table 11 reports the synthesis result.

Table 11 Synthesis result for the transport array, $N = 4$.

Target clock	3.33 ns
Estimated clock	2.878 ns
Latency (cycles)	122
LUT	1,514
FF	744
BRAM	40
DSP	0

The DSP count of zero confirms that no arithmetic was synthesized into the array, so the result reports the transport structure alone. The block RAM is the trace storage, held as one bank per processing element. Separate banks were required because the trace writes contended for memory ports and held the array loop above an initiation interval of one. Partitioning the trace array into one bank per processing element removed the contention and returned the array loop to an initiation interval of one.

The core was exported to the IP catalog and integrated in Vivado with a single AXI DMA engine carrying both the operands and the trace, as shown in Figure 14. The MM2S channel sends A followed by B into the block on a single stream, and the trace returns on S2MM packed into 64 bit words. Control reaches the block over AXI-Lite through an AXI SmartConnect, and the DMA

reaches DDR through a high performance port.

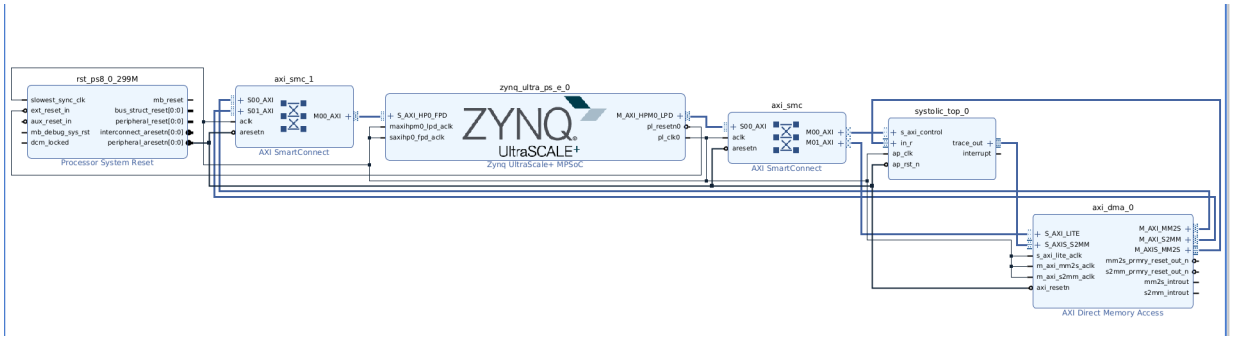


Figure 14 Vivado block design of the transport array.

The bitstream was deployed to a HyperFPGA node and driven from a Jupyter notebook. The matrices A and B were concatenated into one sequence and sent over the DMA, and the trace was read back from the output. Every record was checked against the schedule an Output Stationary array requires, namely that the element at position (i, j) holds $A_{i,k}$ and $B_{k,j}$ at step k. The number of records returned was N^2K, all 64 were returned, and every one matched. Table 12 reproduces the first records of two processing elements.

Table 12 Records recovered from the transport array, showing the first four steps of two processing elements. Each operand value encodes the position it came from in the source matrix. All 64 records were returned and every one matched the expected schedule.

PE	step k	a	b
0,0	0	100	200
0,0	1	101	210
0,0	2	102	220
0,0	3	103	230
1,0	0	110	200
1,0	1	111	210
1,0	2	112	220
1,0	3	113	230

6.3.2 Compute stage

In the second stage the trace was replaced by arithmetic. Each processing element holds its own accumulator and performs one multiply and accumulate every cycle, while A shifts to the right and B shifts downward. The trace banks are removed, and the store stage sends out the C matrix in place of the records. The operands are signed 16 bit values and the accumulator is 32 bit, wide enough for the sum over K. The load stage, the entry delay and the shift are unchanged from the transport stage, so the difference between the two designs is confined to what each processing element does with the operands it receives.

The top level is a DATAFLOW region of three stages, load, compute and store. Table 13 reports the synthesis result and Table 14 the breakdown across the three stages.

The DSP count of 16 is one multiply and accumulate unit per processing element, which is the expected result for a 4×4 array and confirms that the arithmetic was synthesized. A count of

Table 13 Synthesis result for the compute array, $N = 4$.

Target clock	3.33 ns
Estimated clock	2.782 ns
Uncertainty	0.17 ns
Latency (cycles)	86
LUT	1,721
FF	1,851
BRAM	8
DSP	16

Table 14 Latency and resources of the three DATAFLOW stages of the compute array, $N = 4$.

Stage	Latency (cycles)	Latency (ns)	Interval	BRAM	DSP	FF	LUT
systolic_top	86	287.000	41	8	16	1,851	1,721
load	40	133.000	40	0	0	36	316
compute	25	83.325	25	0	16	1,631	1,138
store	19	63.327	16	0	0	19	196

zero, as in Table 11, would indicate that no computation had been built. The compute stage itself accounts for 25 of the 86 cycles, the remainder being the load and the store.

The integration in Vivado is that of Figure 14, with one AXI DMA engine carrying operands and results. The input and output stream widths are both 32 bit, since the elements of C are plain integers, in place of the 64 bit packing used for the trace.

The bitstream was deployed to a HyperFPGA node and driven from a Jupyter notebook. Random A and B matrices were sent over the DMA and C was read back as signed 32 bit integers. The result was compared against the NumPy product of the same operands and matched exactly, with the values

$$C = \begin{bmatrix} 11021 & 15900 & 13885 & 8500 \\ 2347 & 2611 & 2387 & 2828 \\ 12670 & 19471 & 17526 & 3320 \\ 14025 & 20989 & 18153 & 6165 \end{bmatrix}$$

returned from hardware and identical to the reference for every element.

6.3.3 Grid of blocks within one node

The single array of Section 6.3.2 was then partitioned into four blocks arranged in a 2×2 grid, each block holding a 2×2 array of processing elements. The four blocks behave as one 4×4 array rather than as four independent ones, since a value leaving the right edge of one block enters the left edge of its neighbor on the following cycle. The same IP is instantiated in all four positions.

The partition was made because the four blocks are later to be placed on separate nodes. What is a wire between two blocks within one device becomes an Aurora link between two nodes, so the interface between blocks is designed under the constraint that it must survive that substitution.

In a systolic array A enters from the left edge and moves to the right, and B enters from the top edge and moves downward. Once the array is divided, only the blocks on the edges of the grid receive raw data, and block $(1, 1)$ receives none at all. Figure 15 shows the order in which data passes through the grid.

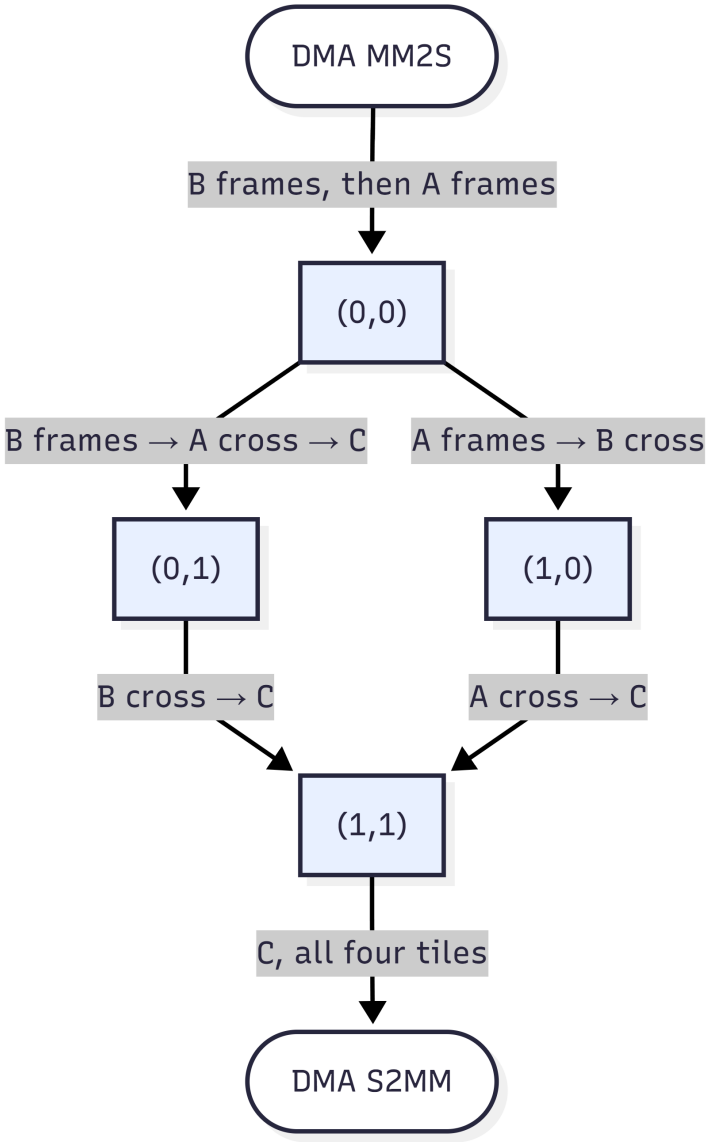


Figure 15 Order of traffic through the grid. Frames are distributed first, crossings follow during the computation, and the tiles of C are gathered last. Only block $(0, 0)$ receives from the DMA and only block $(1, 1)$ returns to it.

Two kinds of traffic follow from this arrangement. A frame is a group of raw values sent once, before the computation begins, and is used to fill the local buffer of a block. A crossing is a single beat sent on every step of the computation, carrying the values that leave the edge of one block and enter the next. The two travel in opposite directions: frames of A travel downward and frames of B to the right, because a frame must reach every block of the column or row that keeps it, while crossings follow the direction in which the operands flow during the computation, A to the right and B downward. Every flow moves to the right or downward, so the graph of blocks is acyclic.

Between any two neighboring blocks there is a single stream in each direction, and it carries three kinds of traffic in sequence: first the frames being passed on to blocks further away, then the crossing values at one beat per step, and last the tiles of C being gathered. The three never overlap, since each belongs to a different phase of the work, and the header identifies which one

is arriving. Each block is a DATAFLOW region of three stages, shown in Figure 16.

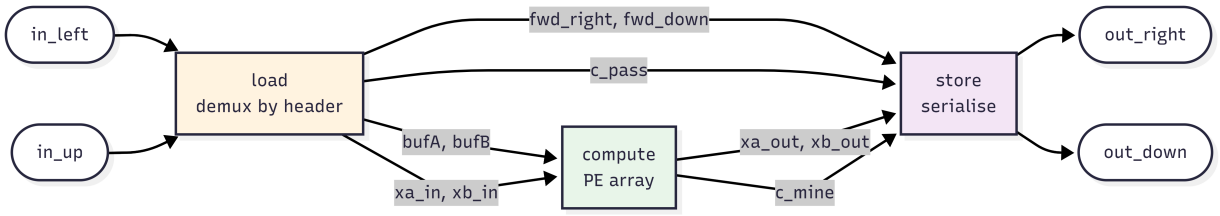


Figure 16 The three DATAFLOW stages of one block. The load stage separates frames, crossings and results by header, the compute stage runs the processing element array, and the store stage merges what is forwarded, what is computed and what is gathered onto the two output ports.

Every group of data is preceded by one 32 bit word, whose fields are given in Table 15.

Table 15 Fields of the 32 bit header that precedes every group of data.

Bits	Field	Meaning
[31:24]	`magic`	fixed pattern, always `0x53`
[23:20]	`type`	which kind of data follows
[19:16]	`dst_r`	row of the block that keeps this data
[15:12]	`dst_s`	column of the block that keeps this data
[11:0]	`len`	number of beats that follow

The magic pattern is a fixed value carried for detection. If a word is lost on the link, the following word read as a header will not contain the pattern and the fault is detected at once, which matters most on the Aurora link. The type field distinguishes an A frame, a B frame, an A crossing, a B crossing and C data, and determines the direction in which a block forwards data it does not keep, A downward and B to the right. The length field states how many beats follow, which removes any dependence on TLAST for counting.

The four blocks are instances of the same IP, so nothing within a block states which position it occupies. The destination fields supply it. Every header states the position of the block that is to keep the data, so a block that keeps a frame has, in the same moment, been told where it is, and no configuration is written from the host.

Each block produces its own tile of C, and the tiles are gathered so that a single DMA reads the complete result, as shown in Figure 17. The tiles move to the right along a row of blocks and then downward along the last column, so the block in the bottom right corner is the only one that returns data to the DMA.

Table 16 reports the synthesis result for one block and Table 17 the breakdown across its three stages. The DSP count of four is one multiply and accumulate unit per processing element of a 2×2 block, the same relation observed for the single array in Table 13.

Each of the four positions was verified separately in C/RTL co-simulation, one transaction per run, and all four passed. Table 18 reports the measured result for each. The computation itself is 11 steps in every block, and the remainder of each figure is time spent waiting for operands to arrive.

The four blocks were instantiated from the same IP and connected as shown in Figure 18. Each instance exposes the two input ports, the two output ports and an AXI-Lite control interface, and each link between blocks carries an AXI4-Stream Data FIFO. A single AXI DMA engine serves the grid, its MM2S channel driving the input port of block $(0, 0)$ and its S2MM channel receiving

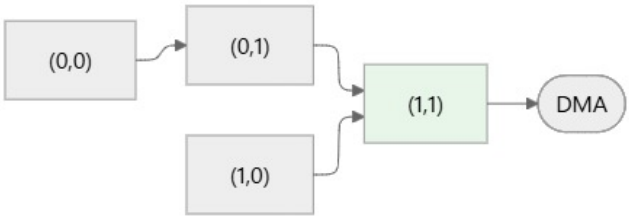


Figure 17 Gathering of the four tiles of C. Tiles move to the right and then downward, and only the corner block returns data to the DMA.

Table 16 Synthesis result for one block of the grid.

Target clock	3.33 ns
Estimated clock	2.794 ns
Uncertainty	0.17 ns
Latency (cycles)	112
LUT	4,257
FF	2,497
BRAM	0
DSP	4

Table 17 Latency and resources of the three DATAFLOW stages of one block.

Stage	Latency (cycles)	Latency (ns)	Interval	BRAM	DSP	FF	LUT
sa_grid	112	373.000	113	0	4	2,497	4,257
load	112	373.000	112	0	0	157	1,096
compute	42	140.000	42	0	4	950	1,094
store	75	250.000	75	0	0	166	1,222

Table 18 Co-simulation result for each block position, at the 300 MHz programmable logic clock. Beats received and beats sent count the words crossing the input and output ports, the latter comprising the crossings and the tiles of C. The stage figures include the cycles a stage spends waiting for its neighbors, so they do not sum to the total.

Block	Total (cycles)	load	compute	store	Beats received	Beats sent	Time (μs)
$(0,0)$	67	60	63	66	20	39	0.223
$(0,1)$	45	36	38	44	22	21	0.150
$(1,0)$	41	32	38	40	17	17	0.137
$(1,1)$	44	35	29	43	38	17	0.147

the gathered result from block $(1,1)$. Control reaches the programmable logic over AXI-Lite through an AXI SmartConnect on the `M_AXI_HPM0_LPD` port, and the DMA reaches DDR through a second SmartConnect on the `S_AXI_HP0_FPD` port. The programmable logic runs at 300 MHz.

The bitstream was deployed to a HyperFPGA node and the grid was driven from a Jupyter notebook. The header returned with the result carried the expected magic pattern, and the

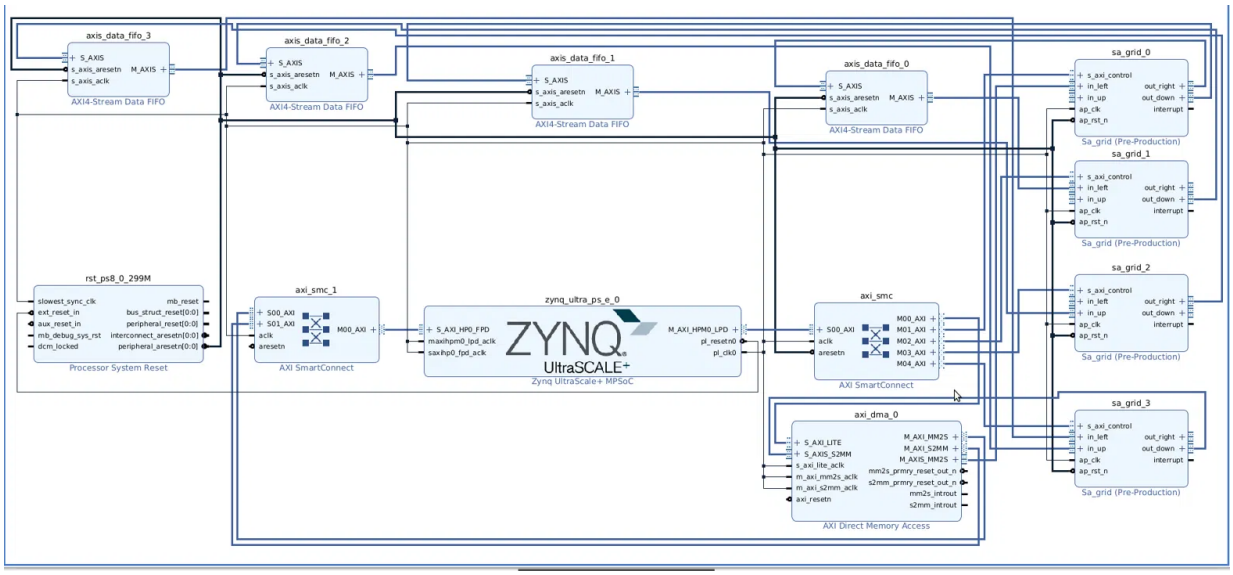


Figure 18 Vivado block design of the grid. The four instances of the block IP are on the right, each pair joined by an AXI4-Stream Data FIFO, with one AXI DMA engine driving block $(0, 0)$ and receiving the gathered result from block $(1, 1)$.

matrix C read back was compared against the NumPy product of the same operands, returning a maximum absolute error of zero over all four tiles. The on node end to end latency was measured from the host over ten runs, giving a minimum of 8.5 μs, a median of 8.7 μs and a mean of 10.3 μs with a standard deviation of 3.4 μs. The synthesis figure for one block is 0.373 μs, so the measured interval is dominated by the transfer across the PS-PL boundary and the driver overhead on the host rather than by the computation.

The grid is the last configuration that fits within one device. Every mechanism that a distributed array requires is present in it and verified on hardware: the operands are addressed rather than placed, the position of a block is carried in the data rather than configured, and the three kinds of traffic are merged onto one link in each direction. Replacing each link with an Aurora connection therefore changes the medium between blocks without changing the design on either side of it.

6.4 Deployment of the grid across four nodes

The grid of Section 6.3.3 was deployed across four nodes of the cluster, one block per node. The block IP was not modified. Each AXI4-Stream Data FIFO joining a pair of blocks inside the device was removed and replaced by an Aurora 8B10B link carried over the GTH transceivers, so the change is confined to the medium between blocks and the logic on either side of it is that already verified on hardware.

Four Aurora cores are instantiated on every node, one for each edge of the block, and the configuration applied to all of them is stated in Table 19. The framing interface is used rather than the streaming interface, so that the boundary of a group of data is preserved across the link and the header of Table 15 is delivered at the head of the frame that carries it. The user interface is four bytes wide, which matches the width of the stream carried between blocks, so no width conversion is required. A cyclic redundancy check (CRC) is enabled on every link, which places a second detection mechanism alongside the magic pattern already carried in the header.

The four cores are named by the direction of the edge they serve, up, left, right and down, and each presents one AXI4-Stream input port and one AXI4-Stream output port to the rest of

Table 19 Configuration of the Aurora cores. The same settings are applied to all four links of every node.

Parameter	Value
Core	Aurora 8B10B v11.1
Lanes per link	1
Lane width	4 bytes
Line rate	6.25 Gb/s
Dataflow mode	Duplex
Interface	Framing
Flow control	None
Back channel	Sidebands
Scrambler, descrambler	Disabled
CRC	Enabled
Shared logic	Included in the core
GT quad	X0Y1
GT lane	X0Y4
GT reference clock	MGTREFCLK1 of quad X0Y1, 125 MHz
Initialization clock	156.25 MHz

the design, as shown in Figure 19. The shared logic, being the transceiver quad phase locked loop (PLL), the differential reference clock buffer and the reset logic, is held in the core serving the upward link, and the reference clock, the transceiver reset and the initialization clock are distributed from it to the other three. Each core is wrapped with two first in first out buffers, one on the transmit path and one on the receive path, which cross between the Aurora user clock and the programmable logic clock, as shown in Figure 20.

The node level design is shown in Figure 21. The block receives on the upward and leftward ports and drives the downward and rightward ports, which is the direction of travel established for frames and crossings in Section 6.3.3. Each node runs the same bitstream. Block $(0, 0)$ receives its operands from the AXI DMA engine of its node through a stream multiplexer, selected by a single ComBlock register, and block $(1, 1)$ returns the gathered result to the DMA engine of its node, so the host drives the cluster through the same two transfers used on the single node. Position is therefore supplied at run time in two ways, in the destination fields of the header within the computation and in the ComBlock register for the attachment of the DMA engine, and no node carries logic that states which position it occupies. The assignment of grid positions to nodes is fixed by the transceiver cabling of the carrier and was confirmed against the exported hardware description: block $(0, 0)$ on node 0-2, $(0, 1)$ on 0-3, $(1, 0)$ on 1-2 and $(1, 1)$ on 1-3.

Two defects were found and resolved during deployment. The device tree overlay produced by the unpacking tool requested a firmware file whose name carried a spurious suffix, so the kernel could not find the bitstream and rejected the overlay with ENOENT. The overlay was recompiled from its source and programming then succeeded on all four nodes. Separately, the block netlist shows that the downward output of each block passes through a stream broadcaster whose second output feeds the local DMA receive channel, so that channel must be armed on every node, not only on the node of block $(1, 1)$, for traffic to flow. The host procedure was corrected accordingly.

The design carries a static register file exposing fifteen status signals per transceiver and three clock frequency counters, and a host side readout of these registers was built from the exported

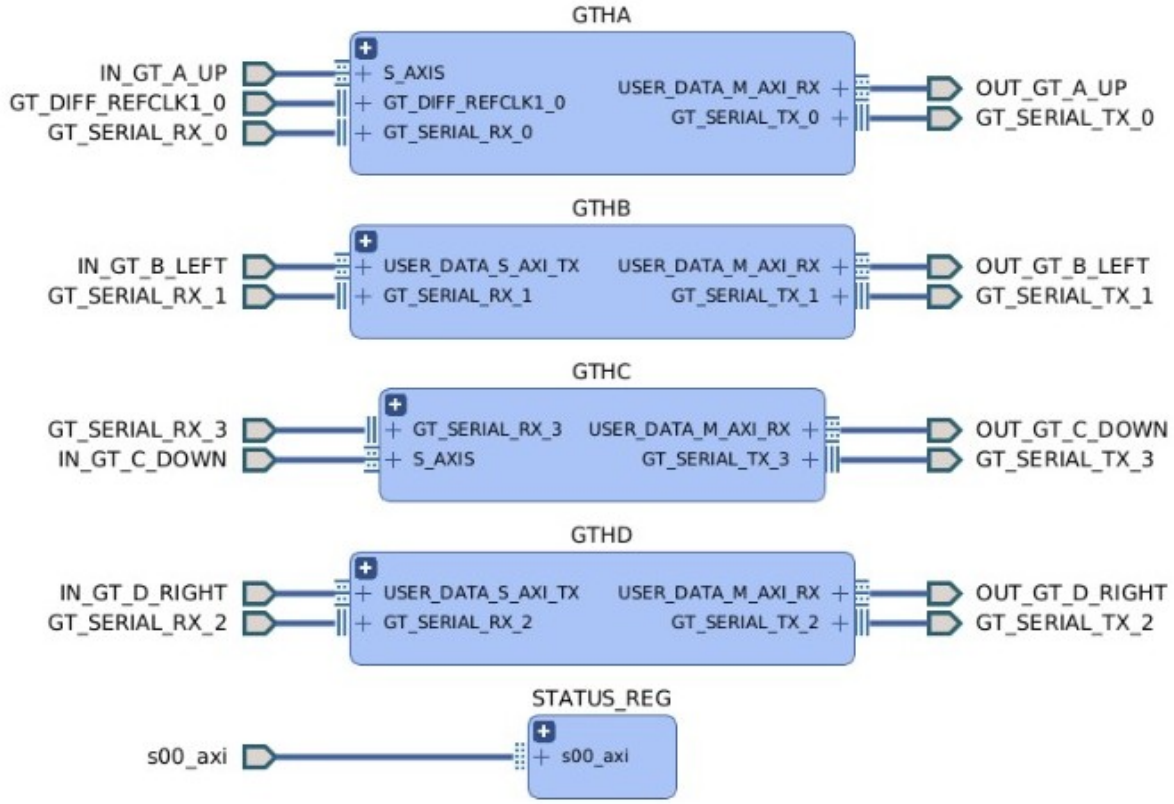


Figure 19 The four Aurora links of one node, one for each edge of the block. Each core presents one AXI4-Stream input and one AXI4-Stream output to the design, and the status register file is read over AXI-Lite.

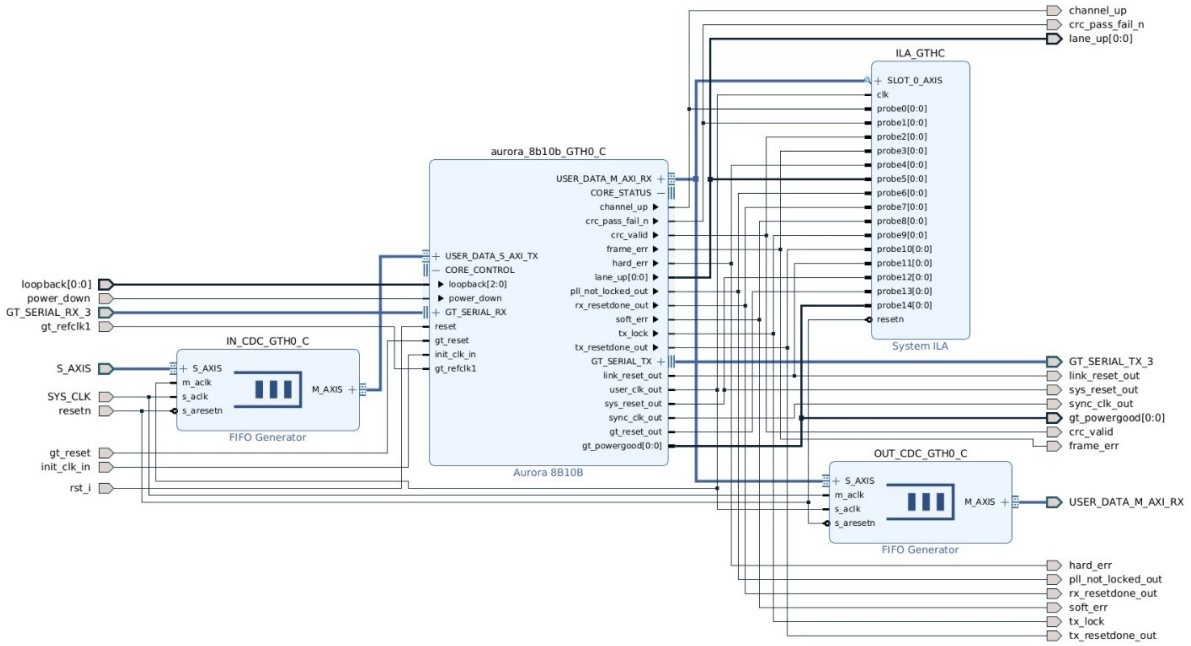


Figure 20 One Aurora link in detail. The two first in first out buffers cross between the Aurora user clock and the programmable logic clock, and the status outputs of the core are brought to an integrated logic analyzer and to the status register file.

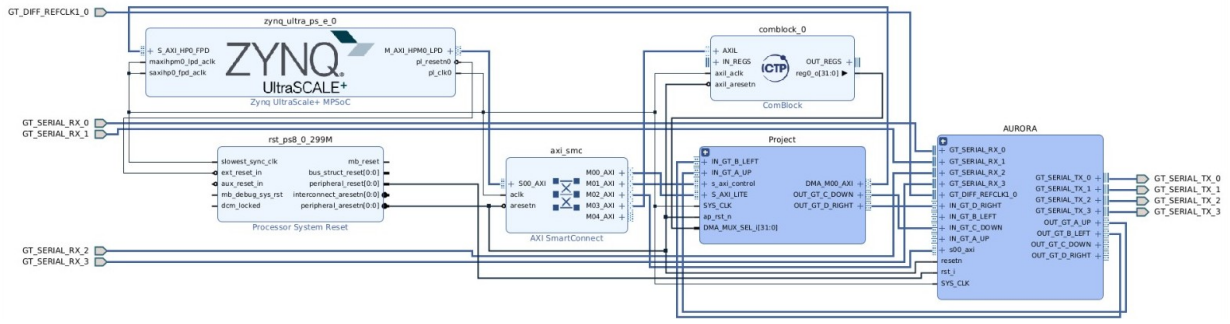


Figure 21 Node level block design. The processing system, the ComBlock carrying the position register and the four Aurora links surround the block, which is the same IP verified in Section 6.3.3.

hardware description, requiring no debug cable. Table 20 summarizes the readings, which are identical on all sixteen transceivers of the four nodes.

Table 20 Transceiver status and measured clocks at deployment, identical across all sixteen transceivers on the four nodes.

Signal	Reading
`gt_powergood`	1
`pll_not_locked`	1
`channel_up`, `lane_up`	0
`hard_err`	1
Measured init clock	156.25 MHz
Measured system clock	300.00 MHz
Measured transceiver user clock	0

The pattern is consistent with a single cause. The transceivers are powered and out of reset, the on fabric clocks are present and the frequency counters read them correctly, but the user clock generated by the transceiver PLL never starts and the PLL never locks, which indicates that the 125 MHz reference is not reaching the clock input of the bank. The board schematic shows this reference is produced by a programmable clock synthesizer whose configuration is loaded at boot. The synthesizer was read over the Inter-Integrated Circuit (I2C) bus from the processing system and responds with all outputs enabled and its drivers configured, so the boot initialization did run. The remaining candidates are the frequency programmed on the relevant output and the selection of the bank clock input itself, both of which are platform provisioning settings outside user control.

The deployment is therefore complete up to this dependency. The bitstream, the node mapping, the position register and the host procedure are verified, and the links do not reach the operating state, so the cluster scale measurement defined in Section 5 was not carried out. Only the single node configuration was measured with the array, and the comparison between the processor level path and the programmable logic level path is therefore not completed.

7 Results

The three platforms are compared on two quantities, the worst case cost of one tile and the total operation time as the matrix grows. The same systolic array algorithm is executed on each, so the comparison isolates the architecture rather than the algorithm. The FPGA figures are those of the

single node configuration of Section 6.3.3, since the links of the four node deployment did not reach the operating state.

7.1 Worst case computation time and memory footprint

Figure 22 reports the worst case computation time for one tile and the peak memory footprint on each platform. The computation time per tile is 1.133 ms on the CPU, 0.52104 ms on the GPU and 0.00025 ms on the FPGA. The ordering follows the execution model of each resource. The CPU evaluates the array as a sequential loop over processing elements, the GPU evaluates one tile as a single graphics operation, and the FPGA holds one multiply and accumulate unit per processing element and advances all of them on the same cycle.

The memory footprint reverses that ordering. The CPU holds 0.64 GB, the GPU 2.04 GB and the FPGA 2.28 GB. The FPGA footprint is the largest of the three, and the cause is on the host rather than in the array. The array itself holds only the operands of one tile in registers, while the direct memory access driver allocates a separate host buffer for every transfer, so the quantity recorded grows with the number of transfers issued rather than with the size of the array.

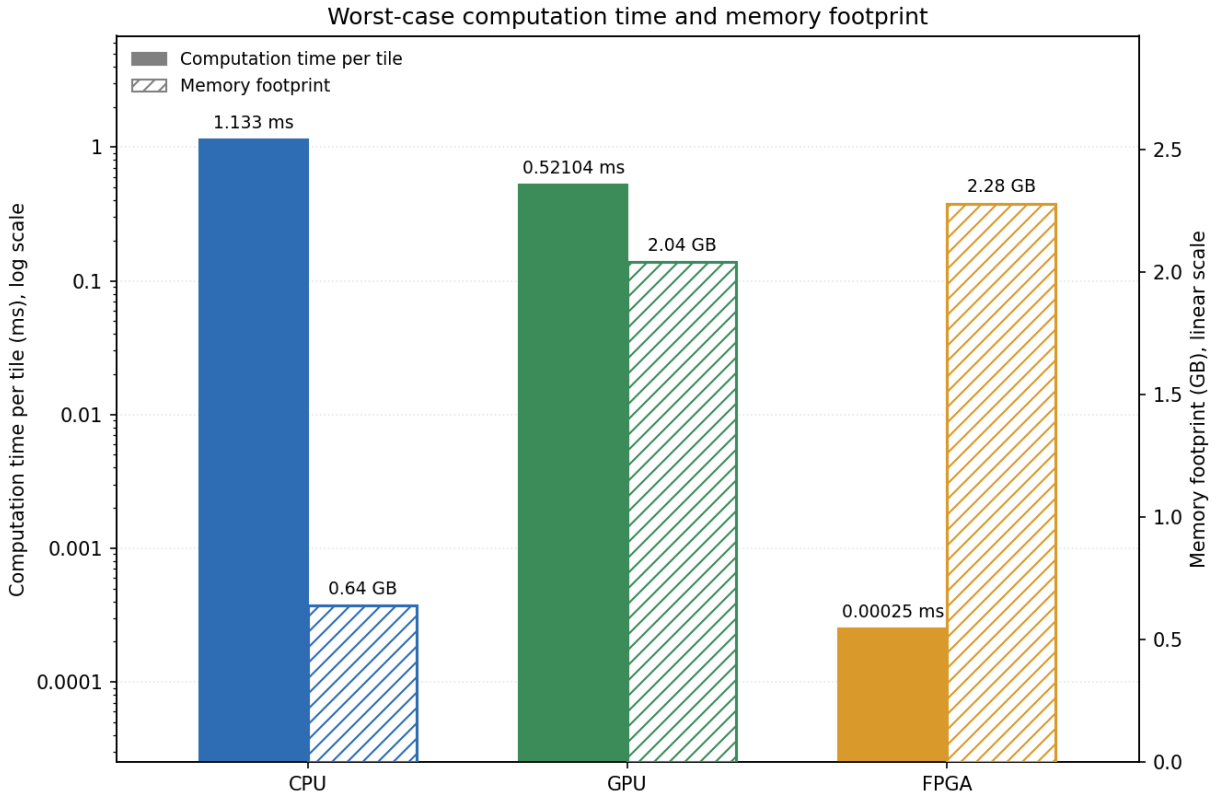


Figure 22 Worst case computation time per tile and peak memory footprint on the three platforms. The computation time is drawn on a logarithmic scale and the memory footprint on a linear scale. The FPGA memory footprint is the largest because the direct memory access driver allocates a separate host buffer for each transfer.

7.2 Total operation time against matrix size

Figure 23 reports the total operation time on square matrices from 4×4 to 128×128. The FPGA lies below the GPU and the GPU below the CPU at every size tested, and the separation between the three widens as the matrix grows, so the advantage of the array is not a fixed offset but

increases with the size of the problem.

Matrices larger than the array are decomposed into tiles and mapped onto it one after another, so the growth of the FPGA curve reflects the number of tiles issued rather than any change in the array. Every point on all three curves was checked against the reference model before the timing was recorded.

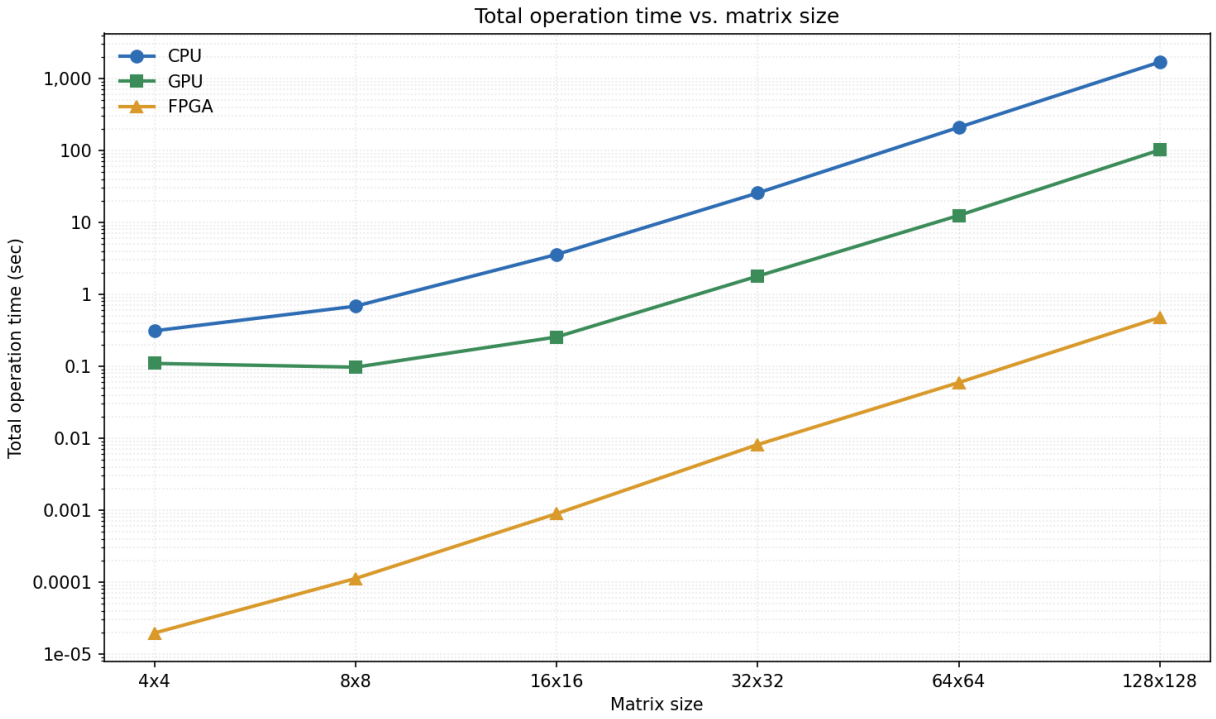


Figure 23 Total operation time against matrix size on the three platforms, on a logarithmic scale. The same systolic array algorithm is executed on each, so the comparison isolates the architecture rather than the algorithm.

7.3 Discussion

Two observations follow. First, the cost of one tile on the FPGA is smaller than on either software platform by a wide margin, and the array occupies a small fraction of the programmable logic of the device, so the size of the array is not what limits the result. Second, the quantity that grows on every platform is the number of transfers, not the arithmetic. On the CPU and the GPU this appears as the communication time separated in Section 6.2.11, which exceeds the computation time by up to an order of magnitude. On the FPGA it appears as the on node end to end latency of Section 6.3.3, measured at a median of 8.7 μs against a synthesis figure of 0.373 μs for one block, and as the memory footprint of Figure 22, which is set by the number of host buffers allocated.

Data movement is therefore the limiting cost on all three platforms, which is the constraint the distributed FPGA literature identifies [7]. Whether the direct path between programmable logic recovers that cost is not established here, since only the processor mediated path was measured with the array.

8 Limitations & Future Work

8.1 Limitations

Five limits apply to the results reported.

1. All hardware measurements of the array were taken at $N = 4$ on a single node, so no figure establishes how the design scales with array size. The array is parameterizable and scales by changing N alone, but that property is verified in synthesis rather than on hardware.
2. No throughput in tera operations per second is reported. That figure is defined for the cluster configuration in Table **??**, and the links of the four node deployment did not reach the operating state, so no measurement across nodes was taken.
3. No power or energy measurement was taken on any platform, so the results are expressed in time and area alone and cannot be compared against the energy figures reported by the architectures reviewed in Section **??**.

8.2 Future Work

The work continues in the following order.

1. The cluster end to end latency and the throughput in tera operations per second are measured across the four nodes, and the communication time is separated from the compute time, which discharges the third limit above.
2. The same workload is distributed over the processor level path and over the programmable logic level path, holding the array design and the node count fixed, which discharges the fourth limit and answers the question the work was set to address.
3. The grid is measured on hardware with an integrated logic analyzer, which resolves the bound of the second limit into a figure and gives the first on node throughput.
4. The array is scaled beyond $N = 4$ toward the capacity of the device, which provides 728 digital signal processing slices against the 16 used by the array at $N = 4$, and the scaling behavior is measured rather than inferred from synthesis.
5. The partitioning strategy and the topology are varied once the two path comparison is established, and the accelerator is exercised on a workload drawn from neural network inference in place of synthetic dense matrices.

9 Conclusion

This report covers an eight week project on distributing a systolic array across an MPSoC-FPGA cluster. The literature optimizes single arrays inside single devices, and the closest multi-board FPGA system found scaling limited by communication rather than compute, since all data between boards moved through the processors [**johnson**]. That gap is the reason communication, rather than the array, is the variable under study.

Five matrix multiplication algorithms were compared on measured hardware cost. The streaming formulation was selected at 29,114 cycles and 3,398 LUT, and returned zero error against a NumPy reference. In software, communication exceeded computation by up to an order of magnitude, and overlapped dispatch cut total operation time from 43.843 s to 25.623 s.

On the programmable logic the array was built in three stages, each verified on hardware. The transport stage returned all 64 expected records at zero DSP. The compute stage added one multiply and accumulate unit per processing element, 16 in total, and matched the reference exactly. The array was then split into a 2×2 grid of blocks. Each block learns its position from the header rather than from configuration, and all traffic shares one link in each direction, so a block

boundary can become a node boundary without change. The grid returned zero error at 8.7 μs median over ten runs.

The grid was then deployed across four nodes, with each link replaced by an Aurora 8B10B connection and the block IP left unmodified. The bitstream, the node mapping, the position register and the host procedure were verified on all four nodes. The transceivers did not reach the operating state, since the 125 MHz reference does not reach the clock input of the bank, which is a platform setting outside user control. The cluster measurement waits on that alone.

Across the three platforms one tile cost 1.133 ms on the CPU, 0.52104 ms on the GPU and 0.00025 ms on the FPGA, and the FPGA stayed below both at every matrix size, with the gap widening as the matrix grew. On all three the dominant cost is moving data, not arithmetic. Whether the direct path between programmable logic recovers what the processor level path costs remains open, and the design needed to answer it is built and in place.

References

[1] Rui Xu et al. “A Survey of Design and Optimization for Systolic Array-based DNN Accelerators”. In: *ACM Computing Surveys* 56.1 (2023). Article 20, pp. 1–37.

[2] Meng Han et al. “ReDas: A Lightweight Architecture for Supporting Fine-Grained Reshaping and Multiple Dataflows on Systolic Array”. In: *IEEE Transactions on Computers* 73.8 (2024), pp. 1997–2011.

[3] B. Asgari, R. Hadidi, and H. Kim. “MEISSA: Multiplying Matrices Efficiently in a Scalable Systolic Architecture”. In: *Proc. 2020 IEEE 38th Int. Conf. on Computer Design (ICCD)*. Hartford, CT, USA, 2020, pp. 130–137. DOI: 10.1109/ICCD50377.2020.00036.

[4] Ahmed Abdelmaksoud et al. “ADiP: Adaptive-Precision Systolic Array for Matrix Multiplication Acceleration”. In: *IEEE Open Journal of the Solid-State Circuits Society* (2026). DOI: 10.1109/OJSSCS.2026.3684844.

[5] Rui Xu et al. “Heterogeneous Systolic Array Architecture for Compact CNNs Hardware Accelerators”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2022), pp. 2860–2871.

[6] S. Ramadurgam and Darshika G. Perera. “A Systolic Array Architecture for SVM Classifier for Machine Learning on Embedded Devices”. In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023, pp. 1–5.

[7] Hans Johnson et al. “Reconfigurable Distributed FPGA Cluster Design for Deep Learning Accelerators”. In: *Proc. 2023 IEEE Int. Conf. on Electro Information Technology (eIT)*. 2023.

[8] W. Florian Samayoa et al. “HyperFPGA: A Scalable Reconfigurable Platform for Computing Paradigms Research”. In: *IEEE Access* 13 (2025), pp. 201377–201391. DOI: 10.1109/ACCESS.2025.3637300.

[9] S. I. Fierro. “Design and Evaluation of Reconfigurable Systolic Arrays for Neural Networks”. MA thesis. Politecnico di Torino, 2024–2025.

[10] Ahmed J. Abdelmaksoud, Shady Agwa, and Themis Prodromakis. “DiP: A Scalable, Energy-Efficient Systolic Array for Matrix Multiplication Acceleration”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 73.1 (2026), pp. 283–293. DOI: 10.1109/TCSI.2025.3591960.

[11] A. Cem Yüzügüler et al. “Scale-out Systolic Arrays”. In: *ACM Transactions on Architecture and Code Optimization* 20.2 (2023). Article 27, pp. 1–25. DOI: 10.1145/3572917.

[12] Alejandra Sanchez Flores et al. “Energy and Precision Evaluation of a Systolic Array Accelerator Using a Quantization Approach for Edge Computing”. In: *Electronics* 13.14 (2024), p. 2822. DOI: 10.3390/electronics13142822.

[13] Yaman Umuroglu et al. “Optimizing Bit-Serial Matrix Multiplication for Reconfigurable Computing”. In: *ACM Transactions on Reconfigurable Technology and Systems* 12.3 (2019), pp. 1–24. DOI: 10.1145/3337929.