MemPool Meets Systolic: Flexible Systolic Computation in a Large Shared-Memory Processor Cluster

Samuel Riedel* Gua Hao Khov* Sergio Mazzola* Matheus Cavalcante* Renzo Andri† Luca Benini*‡
*IIS, ETH Zürich ‡DEI, University of Bologna
*{sriedel,khovg,smazzola,matheusd,lbenini}@ethz.ch †info@renzo.ch

***Abstract*—Systolic arrays and shared-memory manycore clusters are two widely used architectural templates that offer vastly different trade-offs. Systolic arrays achieve exceptional performance for workloads with regular dataflow at the cost of a rigid architecture and programming model. Shared-memory manycore systems are more flexible and easy to program, but data must be moved explicitly to/from cores. This work combines the best of both worlds by adding a systolic overlay to a general-purpose shared-memory manycore cluster allowing for efficient systolic execution while maintaining flexibility. We propose and implement two instruction set architecture extensions enabling native and automatic communication between cores through shared memory. Our hybrid approach allows configuring different systolic topologies at execution time and running hybrid systolic-shared-memory computations. The hybrid architecture's convolution kernel outperforms the optimized shared-memory one by 18%.**

***Index Terms*—manycore, RISC-V, systolic array**

I. Introduction

Systolic array architectures are widely used to tackle highly parallel, compute-intensive workloads. They are based on an array of specialized processing elements (PEs) communicating through a neighborhood-based interconnect forming an application-specific topology [1]. While they excel in their target domain, their rigid PE interconnection network limits their utilization in other use cases. In contrast, shared-memory manycore systems allow for flexible communication between all PEs via the memory, making them suitable for a wide range of workloads. However, explicit communication via memory lowers efficiency and PE utilization due to latency and contention.

In this work, we combine the best of both worlds by bringing the advantages of the systolic execution model to shared-memory architectures. We extend a shared-memory architecture with two lightweight hardware extensions to enable efficient core-to-core communication through memory-mapped queues. Namely, the *Xqueue* extension enables single-instruction access to any shared-memory queue, while the *queue-linked register (QLR)* extension eliminates communication overhead by managing queue accesses in parallel to the core's computation. Specifically, we extend the open-source, RISC-V-based MemPool architecture, which is a 32-bit general-purpose manycore system with 256 PEs sharing low-latency access to a large L1 memory [2].

The contributions of this paper are: 1) A new hybrid systolic-shared-memory architecture concurrently supporting both systolic topologies and shared-memory operation; 2) Two lightweight hardware extensions enabling fast and automatic communication between cores via memory-mapped queues.

The extensions have an 8.6% area cost and accelerate systolic implementations by up to 23×. Hybrid shared-memory-systolic kernels outperform shared-memory kernels by up to 18%.

II. Hybrid Architecture

In systolic architectures, neighboring PEs are interconnected and directly transfer data to any linked PE on some given topology [3]. The left of Fig. 1 shows an example architecture with a 2D mesh. In contrast, PEs of a shared-memory architecture, i.e., the cores, all communicate through shared memory, which they can all access as shown on the right of Fig. 1.

Due to their flexibility, shared-memory systems can be seen as a generic systolic array composed of a collection of PEs, i.e., the cores. By mapping the communication queues of systolic arrays into shared memory, each PE can communicate with every other PE through MemPool's all-to-all interconnect. For example, MemPool's 256 cores can be viewed as a 16×16 array of PEs where each PE is connected to its neighbors through a memory-mapped queue as illustrated in Fig. 1.

This mapping allows exploring systolic topologies and algorithms on shared-memory systems through software emulation by implementing them with memory-mapped software queues. The left of Figure 2 shows the *Baseline* software implementation, a simple example of a systolic algorithm in a shared-memory system. While the software queues allow for great flexibility, the cores must execute tens of instructions for each queue operation, including multiple memory transactions for queue bookkeeping, which is a significant overhead compared to traditional systolic execution, where data flows through PEs automatically.

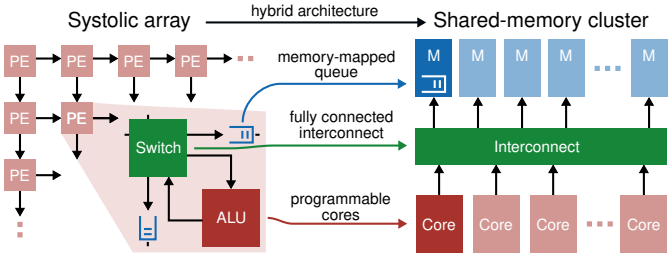

Fig. 1. A 2D mesh systolic array (*left*) and a shared-memory architecture (*right*) with a mapping from systolic to shared-memory to build a hybrid architecture.

978-3-9819263-7-8/DATE23/© 2023 EDAA

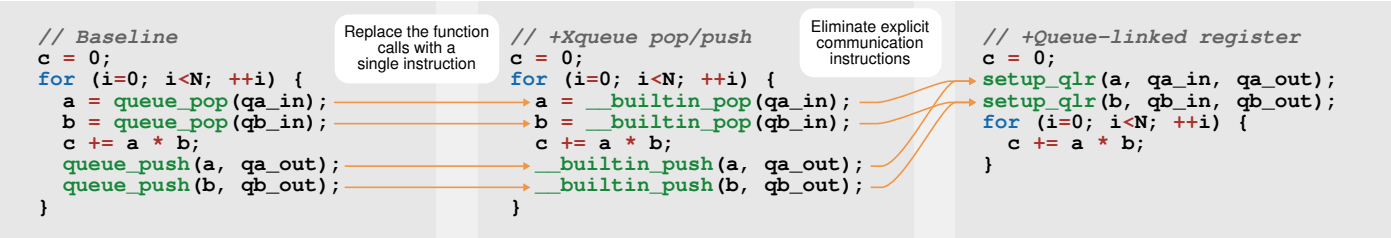

Fig. 2. Simplified code executed on the hybrid architecture's PEs to implement a matrix multiplication with each hardware optimization. The *Baseline* (left) interfaces with the queues through function calls. The Xqueue extension (middle) replaces those function calls with single instructions. Finally, the additional queue-linked register (QLR) extension (right) eliminates the communication instructions entirely. Instead, it requires a small overhead to set up the QLRs.

To overcome the overhead of software-based queue bookkeeping, we propose the *Xqueue* hardware extension. It enhances the RISC-V instruction set architecture with two instructions, a `push` and a `pop`, for efficient access to memory-mapped queues. Instead of explicitly managing the queues in software, the hardware takes care of head and tail pointer updates, queue boundary checks, and queue access, all in a single instruction. In the middle code snippet in Fig. 2, the function calls are replaced by their hardware intrinsics of the Xqueue instructions. The Xqueue extension significantly reduces the number of instructions required to manage inter-core communication. However, the extension requires the number and size of queues to be fixed as hardware parameters. Nevertheless, which PEs connect via which queue remains fully runtime configurable.

Even more aggressively tuned, the *queue-linked register (QLR)* extension brings two additional major communication advantages of systolic computation to shared-memory systems: Communication happens implicitly in parallel to computation, and operands flow directly to the compute-unit. The QLR extension consists of a small configurable unit in each core that automatically pops or pushes from memory-mapped queues. Interfacing with the core's register file, it directly reads/writes from/to pre-defined registers to/from the queue, depending on its configuration. QLRs completely eliminate instructions spent on communication and allow creating a queue network that, once configured, runs entirely autonomously and remains in sync with the associated systolic computations. The rightmost snippet in Fig. 2 illustrates the concept.

III. Results

We implement the systolic MemPool architecture with four queues per core in GlobalFoundries 22nm FD-SOI technology. The extensions come with a hardware overhead of only 8.6% and do not impact MemPool's operating frequency.

Different implementations of *matmul* and *2dconv* kernels, evaluated in cycle-accurate register-transfer level simulation, serve as benchmarks. Both kernels are well-suited for systolic execution due to their regular data flow. The *matmul* implementation is based on a 2D mesh systolic architecture, while the *2dconv* implements a 1D chain topology.

We quantify the benefits of our extensions by running systolic kernels with different extensions enabled. Compared to the baseline systolic software kernel, the Xqueue extension improves performance by 3× for *matmul* and 13× for *2dconv* by removing the overhead of managing the queues in software. Combined with the QLR extension, they achieve speedups of 5× and 23× for *matmul* and *2dconv*, respectively.

The hybrid architecture allows fusing systolic and shared-memory schemes when implementing kernels. We take advantage of those capabilities by implementing hybrid systolic-shared-memory versions of both kernels. The *matmul* kernel optimizes data reuse by having one matrix move through the PEs in a systolic fashion while the other matrix is loaded in a classical shared-memory regime. Furthermore, we explore different mappings of MemPool's cores to a 2D mesh. Similarly, the *2dconv* kernel's data reuse is optimized by simultaneously moving multiple input rows through the systolic topology. These kernels outperform the kernels tuned for the shared-memory architecture, which use a tiled approach to maximize data reuse, by 15% and achieve a multiply–accumulate (MAC) unit utilization of 64% for the *matmul*. Similarly, the hybrid implementation of the *2dconv* outperforms the shared-memory implementation by 18% with a utilization of 77%.

IV. Conclusion

We present a flexible and novel hybrid architecture that combines systolic computation and shared-memory systems by using two lightweight hardware extensions with only 8.6% area overhead. The resulting architecture can implement any systolic topology and still operate in a shared-memory mode, thus enabling hybrid implementations that simultaneously use systolic dataflow and shared-memory concepts. The proposed hybrid architecture outperforms the specialized shared-memory implementations by up to 18% and achieves a MAC unit utilization of up to 77%.

Acknowledgment

This work was supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies.

References

[1] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[2] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A shared-L1 memory many-core cluster with a low-latency interconnect," in *2021 Design, Automation, and Test in Europe Conf. and Exhib.*, Grenoble, France, Mar. 2021, pp. 701–706.

[3] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982.