Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

MemPool: A Shared-Memory, Massively-Parallel Visual Processor

Project Conclusion

Matheus Cavalcante, Samuel Riedel, Prof. Luca Benini
matheus.cavalcante, sriedel, lbenini *at* iis.ee.ethz.ch

July 2021

MemPool

Image Signal Processors (ISPs), also known as Image Processing Units (IPUs), have been used for many years in mobile System-on-Chips (SoCs). The key function of an ISP is to deliver the most accurate and highest image quality when pulling data from an image sensor and processing each pixel. In the past, this task was significantly simpler, with the quality enhancements mostly based on traditional luma and chroma processing. With the advent of advanced computational photography and computer vision, the need for ISPs with higher computing power has emerged. At the same time, advanced computer vision applications also require a high flexibility in terms of pixel manipulation functions to be implemented. Hence, traditional ISPs are rapidly becoming obsolete. They need to be replaced with *Visual Processors* that are much more capable in terms of performance, while being flexible and energy efficient.

It is commonly believed that the number of cores in a single L1-shared cluster is bound to the low-tens limit. To scale the core count of many-core systems into the hundreds, memory sharing is usually done at some high-latency hierarchy level. Moving to multiple clusters creates challenges in terms of programmability: tile-based systems are usually connected by meshes that have long access latency and usually require non-uniform memory access models of computation [1]. However, there is also a push for high core-count in many-core architectures to tackle the embarrassingly-parallel problems of image processing and machine learning.

It is in this context that this project has implemented **MemPool**, a scaled-out manycore system with a low-latency shared L1 memory [2]. MemPool is a 32-bit RISC-V system with 256 small cores sharing a large pool of Scratchpad Memory (SPM), a core-count one order of magnitude higher than what was previously considered achievable. In the absence of contention, all SPM banks are accessible within at most 5 cycles.

MemPool achieves this high core-count with a low access latency by means of a highly hierarchical design. At the bottom of its hierarchy, we have *tiles*, shown in Figure 1, with four cores and 16 KiB of SPM, accessible within one cycle from the cores in the same tile. The cores and the memory banks are connected via a fully-connected crossbar, ensuring a low contention in the local accesses. Our hybrid addressing scheme [2] ensures that often-accessed data (e.g., the stack) are stored in those low-latency banks.

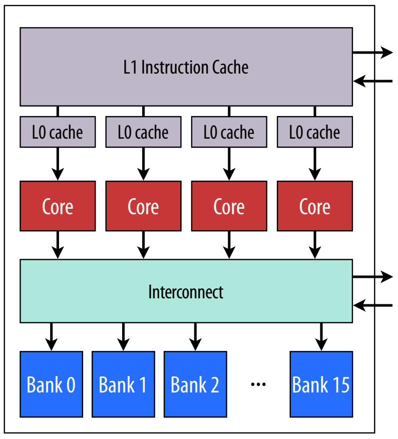

Figure 1: Architecture of the MemPool tile.

Tiles are interconnected with an interconnect topology which scales better than the fully-connected crossbar. In MemPool, we use radix-4 butterfly networks, ensuring connection between all tiles, while avoiding intense routing congestion in the back-end implementation. Sixteen tiles are tightly connected to form a *group*, from within all SPM banks can be accessed in three cycles. Four of those groups form the MemPool cluster, with 256 cores and 1 MiB of SPM, all accessible within five cycles of zero-load latency.

Performance

In terms of performance, when compared with an idealized (and unfeasible) cluster with 256 cores sharing the same amount of L1 memory through a fully-connected crossbar with one cycle of latency, our cluster performs well. We benchmarked MemPool with three real-world highly-parallelizable signal processing benchmarks:

matmul: a matrix multiplication of two 64×64 matrices, for which accesses are predominantly remote;

2dconv: a 2D discrete convolution with a 3×3 kernel, for which all accesses are local, except for cores working on windows that require data from two tiles;

dct: a 2D discrete cosine transformation operating on 8×8 blocks residing in local memory. It uses the stack to store intermediate results, i.e, all accesses are local, given the stack is mapped to local banks.

Figure 2 shows that MemPool achieves 80 % of the peak performance, when running a 64×64 matrix multiplication [2], which has a high number of remote accesses. For a highly local kernel such as *dct*, the performance of MemPool is almost the same as these of the idealized cluster.

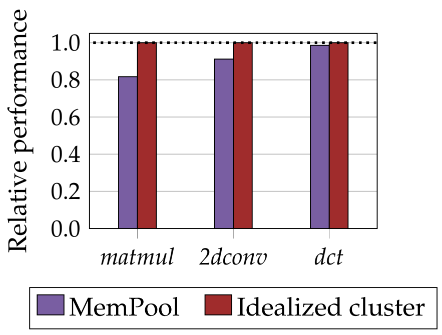

Figure 2: Performance of the three benchmarks, relative to the baselines.

The MemPool cluster was implemented as a 4.6 mm × 4.6 mm macro, with 55 % of the area being covered by the tiles. The area overhead was driven by congestion, which is the main constraint of the design, particularly at the center of the design. MemPool runs at 700 MHz in typical conditions—and at 480 MHz in worst-case conditions—with the advanced GLOBALFOUNDRIES 22FDX FD-SOI technology node.

In terms of power—in typical conditions, at 500 MHz, while running the *matmul* kernel—the MemPool cluster consumes 1.55 W, 86 % on that in the tiles. Each local load (i.e., within the same tile) uses 8.4 pJ. About half of this energy consumption, 4.5 pJ, is spent at the local interconnect. Remote loads use the global interconnect, which raises their energy consumption to 16.9 pJ. In this case, the interconnects consume 13.0 pJ, or 2.9× the energy consumed at the interconnects for a local load.

As a comparison with arithmetic instructions, a local load uses about as much energy as a complex instruction such as mul, or 2.3× the energy consumed by a simple add. Remote loads have the highest energy requirements, but even then that is only 4.5× the energy of an add. This result confirms that MemPool is a balanced design that is not severely interconnect-dominated.

Scalability of the design

With its hierarchical design and parametrized implementation, MemPool can easily be scaled to different numbers of cores. A scaled-down version of MemPool was recently taped-out in TSMC's 65 nm technology[1]. It features 16 cores, which are distributed across four groups of one tile each. All memory banks are accessible within three cycles, in the absence of contention.

[1]`http://asic.ee.ethz.ch/2021/Minpool.html`

A more involved question concerns how to scale MemPool up to the thousands of cores. Since wire-delay limited the size of MemPool, it is not possible to increase the number of groups in a MemPool cluster without impacting either the memory latency of remote accesses or the throughput of the global interconnect. We can hide the long memory latency through instruction scheduling and supporting multiple outstanding instructions. Instruction scheduling is supported by GCC and LLVM and can be done automatically with little support by the programmer, e.g., unrolling loops to provide enough independent instructions.

After a certain latency threshold, however, it is no longer possible to rely on the compiler to hide the latency. An idea would be to rely on double-buffering, by using a Direct Memory Access (DMA) engine to copy data to local memory before the cores use them. The programming model does become more tedious since the programmer needs to think about two iterations of its loop in parallel.

It is possible to scale up the core count of MemPool through the integration of more groups, in a 2D array interconnected with a mesh network. MemPool becomes similar to other ISPs, which also use meshes to connect its processing elements. However, MemPool does so at an upper hierarchy level, using a high-latency highly-scalable mesh network to connect processing elements clusters with 64 cores. Within a group, the accesses would still be done through a high-throughput low-latency network.

From the point of view of physical feasibility, we could scale up MemPool almost indefinitely. In this design, we use a physically feasible block, the MemPool group, and only connect it to its immediate neighbors. This greatly alleviates the routing congestion issues on the interconnect between groups. In terms of the programming model, this approach to scaling up MemPool keeps its shared view of memory, which allows implementing a wider set of algorithms compared to more rigid architectures such as systolic arrays. This also simplifies programming, when compared to clustered architectures, since the global/shared L1 simplifies the memory hierarchy and takes tedious memory allocation off the programmer's hands compared to multi-cluster approaches. Parallelizing is much easier if there is only one hierarchy of cores. The problem only has to be parallelized in one dimension instead of split into sub-problems that have to be parallelized.

Bibliography

[1] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1654 – 1663, 2013, 2013 International Conference on Computational Science.

[2] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A shared-L1 memory many-core cluster with a low-latency interconnect," in *Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, Mar. 2021.