RETROSPECTIVE: Plasticine: A Reconfigurable Architecture For Parallel Patterns

Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao,
Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, Kunle Olukotun
Stanford University

I. Motivation

Rapid advances in modern machine learning algorithms have led to a surge in compute demand, sparking extensive research in computer architecture focused on specialized, programmable accelerators. By leveraging GPUs as the computing substrate and utilizing high-level programming frameworks such as PyTorch and Tensorflow, researchers can rapidly iterate through various deep-learning model architectures. However, hardware capabilities often limit the nature and scale of these AI models. Programmable accelerators come in two broad flavors: instruction-based and reconfigurable architectures. Instruction-based accelerators offer ease of programming but suffer from hardware overheads associated with instruction and thread state management. Alternatively, reconfigurable architectures, like FPGAs and CGRAs, avoid these overheads but typically exhibit limitations in compute density and programmability due to their fine-grained and fully static nature.

Plasticine aimed to answer the following question: *Can we develop a hardware architecture that eliminates the performance and hardware overheads of instructions and threads, while circumventing the programmability challenges of FPGAs and CGRAs?*

II. The Plasticine Architecture

Plasticine is a Reconfigurable Dataflow Architecture (RDA) with a programmable sea of compute and memory units in a programmable interconnect. We chose to build a dataflow architecture as it avoids the overheads of instruction and thread management in hardware. To address programmability concerns, the hardware primitives in Plasticine were built to accelerate composable software primitives called parallel patterns like *map*, *zip*, *reduce*, *filter*, and *groupby*. Prior research had shown that parallel patterns can express a broad variety of applications while capturing rich data locality and parallelism information that aids compilation. As a result, Plasticine's primary objective was to be an efficient compiler target that enables exploiting task, data, and nested pipelined parallelism exposed by parallel pattern primitives.

Plasticine features a mesh topology where the vectorized compute pipeline in the Pattern Compute Units (PCU) and distributed scratchpads in the Pattern Memory Units (PMU) are interconnected using programmable switches. Additionally, Address Generation and Coalescing Units (AGCUs) bridge the off-chip components and the I/O subsystem. The interconnect is statically configured and includes physically distinct vector, scalar, and control paths. The granularities of PCUs and PMUs were empirically selected to maximize utilization across a comprehensive set of representative benchmarks.

Plasticine exploits nested parallelism at multiple levels: SIMD parallelism across PCU ALU stages, fine-grained pipeline parallelism across PCUs, coarse-grained parallelism between compute and memory access, and task parallelism across kernels using dataflow-driven PCU and PMU groups. The PCUs act as a logical pipeline stage and the PMUs hold sharded and double-buffered logical tensors with software-orchestrated flow control.

III. Evolution and Impact

Since its publication in 2017, Plasticine has gained significant adoption and received enhancements from the research community, catering to a wider range of domains including databases [4], [5] and unstructured sparsity [3]. Additionally, research prototypes have been proposed to compile distributed [6] and imperative programs [7] on Plasticine.

Plasticine was commercialized as the SambaNova Systems SN10 Reconfigurable Dataflow Unit (RDU) [1], [2] to accelerate modern machine learning workloads. The SN10 RDU and its subsequent generations have demonstrated the feasibility of building and deploying RDUs at scale. The SambaFlow software stack enables compiling and running Large Language Models (LLMs) with trillions of parameters to achieve state-of-the-art or higher performance and accuracy.

Going from an academic prototype to an industrial product required us to add several features to improve performance and to handle complex real-world cases. PCUs added a systolic array to implement matrix multiply operations efficiently. PMUs added data manipulation capabilities to perform tensor transformations like transpose at full throughput. Data interconnect changed from fully static to packet-switched. Hardware support was added to handle outof-order data streams and unaligned memory accesses. We highlight observations and lessons learned from the SN10 RDU design below.

- *Software Ecosystem*: A substantial software stack is required to enable PyTorch applications to run efficiently on the SN10. Accurate cost models enable upper compiler layers to efficiently explore the large design space with operator fusion, tiling decisions, and parallelization. On-chip buffer management optimized on-chip memory usage and prevented hangs. A dataflow programming model allowed power programmers to build high-performance operators without worrying about hardware specifics. Bandwidth-aware place-and- route efficiently allocated on-chip bandwidth to multiple data streams. Control token-based instrumentation measured stage latencies to help identify performance bottlenecks. Iterative co-design helped strike the right balance between hardware and software.

- *On-chip Memory Capacity*: RDUs packed 3-10x more on-chip memory capacity compared to architectures with similar compute capability partially due to reduced instruction and thread management overhead. Consequently, RDUs captured more temporal locality (larger data tiles) and spatial locality (intermediate results between stages), drastically reducing off-chip bandwidth requirements for high utilization. Specifically, RDUs could achieve the same or higher compute utilization as GPUs on large language models in spite of having 10x lesser off-chip memory bandwidth.
- *On-chip Memory Bandwidth*: To sustain high utilization for compute graphs parallelized within and across PCUs, it is crucial to have (1) hardware that enables full throughput vectorized memory access within a PMU and (2) software that shards the logical tensor across distributed PMUs. High address bandwidth motivated specialized ALUs in PMUs for efficient address calculation. High data bandwidth necessitated banked scratchpads. While more banks reduced bank conflicts, it also decreased effective memory capacity due to SRAM overheads. Architectural support enabling software-controlled interleaving across banks effectively reduced bank conflicts without requiring a large number of memory banks.
- *Kernel Fusion*: Dataflow flexibility in RDUs enables easier automatic kernel fusion. Specifically, fusion translated to a graph partitioning/clustering algorithm in the compiler on the input compute graph, obviating the need to build and maintain libraries of manually written fused kernels. The flexibility to configure each PCU independently, along with large address and data bandwidth in PMUs helps sustain high utilization for fused subgraphs requiring regular as well as irregular accesses.
- *Specialize for Composability*: Fundamental to dataflow flexibility is architectural hooks that enable software to compose larger logical units by combining smaller units. For example, composing larger systolic arrays out of many PCUs, or larger buffers out of many PMUs. Composing units is rarely free, as they often introduce additional many-to-1 data streams and a data reordering mechanism, which impacts overall utilization. Specialized hardware to handle common communication patterns that arise out of composition can significantly mitigate the negative impact on utilization.
- *Programmability and Interconnect*: A fully static interconnect makes it difficult for software to achieve high compute utilization for arbitrary compute graphs. Packet-switch interconnects with end-to-end credited data streams lowered the barrier for software to map arbitrary graphs for utilization. Furthermore, hardware hooks to configure bandwidth allocation between several concurrent streams allowed software to achieve better placement and routing, which led to higher overall performance.
- *Graph Setup Overheads*: Greater RDU flexibility translates to added area overhead and configuration latency, where the latter can quickly underwhelm the overall performance improvement. Architectural mechanisms to program units in parallel and minimize software driver intervention was critical to mitigate these overheads.

IV. Future

AI and ML constitute the most important workloads driving computing in the world today. In the last decade, we have witnessed the rise of a few dominant model architectures like the Transformer and UNet, and an exponential increase in model parameters and compute requirements. Given this unsustainable trend, we are now at an inflection point. AI researchers and practitioners alike are exploring alternative architectures like Mixture-of-Experts and retriever-based models that trade-off compute requirements for storage and memory bandwidth. The dataflow paradigm in Plasticine and RDU provides the flexibility to make such trade-offs programmatically, thus making them well-positioned to handle future computing workloads. Dataflow with high on-chip memory capacity and bandwidth also enables new training methods with sparse kernels that require operator fusion for higher performance. Broader community adoption requires building a mature software ecosystem with debuggers, profilers, and best practices to extract the full potential of dataflow architectures.

References

[1] R. Prabhakar and S. Jairath, "Sambanova sn10 rdu: Accelerating software 2.0 with dataflow," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–37.

[2] R. Prabhakar, S. Jairath, and J. L. Shin, "Sambanova sn10 rdu: A 7nm dataflow architecture to accelerate software 2.0," in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 350–352.

[3] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector rda for sparsity," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1022–1035.

[4] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 402–415.

[5] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 309–321.

[6] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 615–628.

[7] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, "Sara: Scaling a reconfigurable dataflow accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1041–1054.