DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures

Geraldo Oliveira, Alain Kohli, David Novo, Juan Gómez-Luna, Onur Mutlu

▶ **To cite this version:**

Geraldo Oliveira, Alain Kohli, David Novo, Juan Gómez-Luna, Onur Mutlu. DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures. PACT 2023 - 32nd International Conference on Parallel Architectures and Compilation Techniques, Oct 2023, Vienna, Austria. , 2023, 10.48550/arXiv.2310.10168 . lirmm-04423308

HAL Id: lirmm-04423308

https://hal-lirmm.ccsd.cnrs.fr/lirmm-04423308

Submitted on 29 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures

Geraldo F. Oliveira⋆ Alain Kohli⋆ David Novo‡ Juan Gómez-Luna⋆ Onur Mutlu⋆
⋆ETH Zürich ‡LIRMM, Univ. Montpellier, CNRS

arXiv:2310.10168v1 [cs.AR] 16 Oct 2023

1. Motivation & Problem

The increasing prevalence and growing size of data in modern applications have led to high costs for computation in traditional *processor-centric computing* systems. To mitigate these costs, the *processing-in-memory* (PIM) [1–6] paradigm moves computation closer to where the data resides, reducing the need to move data between memory and the processor. Even though the concept of PIM has been first proposed in the 1960s [7, 8], real-world PIM systems have only recently been manufactured [9–13]. The UPMEM PIM system [9, 10, 14] is the first PIM architecture to become commercially available. It consists of *UPMEM modules*, which are standard DDR4-2400 DIMMs with 16 PIM chips. A *PIM chip* consists of eight small multithreaded general-purpose in-order processors called DPUs. Each *DPU* has exclusive access to a 64 MB DRAM bank (called *MRAM*), a 24 kB instruction memory (called *IRAM*), and a 64 kB scratchpad memory (called *WRAM*). A common UPEM-capable system has 20 DRAM modules with 128 DPUs and 8 GB of memory each, totaling 2,560 DPUs with 160 GB of memory.

To program the DPUs in a UPMEM-capable system, UPMEM has developed a single-instruction multiple-thread (SIMT) programming model. The programming model uses a C-like interface and exposes to the programmer a series of APIs to manage data allocation and data movement between the host CPU/DPUs and within the memory hierarchy of the DPUs. A programmer needs to follow four main steps to implement a given application targeting the UPMEM system. The programmer needs to: (*i*) partition the computation (and input data) across the DPUs in the system, *manually* exposing thread-level parallelism (TLP) to the system; (*ii*) distribute (copy) the appropriate input data from the CPU's main memory into the DPU's memory space; (*iii*) launch the computation kernel that the DPUs will execute; and (*iv*) gather (copy) output data from the DPUs to the CPU main memory once the DPUs execute the kernel.

Even though UPMEM's programming model resembles that of widely employed architectures, such as GPUs, it requires the programmer to (*i*) have prior knowledge of the underlying UPMEM hardware and (*ii*) manage data movement at a fine-grained granularity *manually*. Such limitations can difficult the adoption of PIM architectures in general-purpose systems. Therefore, our **goal** in this work is to ease programmability for the UPMEM architecture, allowing a programmer to write efficient PIM-friendly code *without* the need to manage hardware resources explicitly.

2. DaPPA: A Data-Parallel PIM Framework

To ease the programmability of PIM architectures, we propose DaPPA (data-parallel processing-in-memory architecture), a framework that can, for a given application, *automatically* distribute input and gather output data, handle memory management, and parallelize work across the DPUs. The *key idea* behind DaPPA is to remove the responsibility of managing hardware resources from the programmer by providing an intuitive data-parallel pattern-based programming interface [15, 16] that abstracts the hardware components of the UPMEM system. Using this key idea, DaPPA transforms a data-parallel pattern-based application code into the appropriate UPMEM-target code, including the required APIs for data management and code partition, which can then be compiled into a UPMEM-based binary *transparently* from the programmer. While generating UPMEM-target code, DaPPA implements several code optimizations to improve end-to-end performance.

2.1. DaPPA Overview

Figure 1 shows an overview of our DaPPA framework. DaPPA takes as input C/C++ code, which describes the target computation using a collection of data-parallel patterns and DaPPA's programming interface, and generates as output the requested computation. DaPPA consists of three main components: (*i*) DaPPA's data-parallel pattern APIs, (*ii*) DaPPA's dataflow programming interface, and (*iii*) DaPPA's dynamic template-based compilation.

Data-Parallel Pattern APIs. DaPPA's data-parallel pattern APIs (❶ in Figure 1) are a collection of pre-defined functions that implement high-level data-parallel pattern primitives. Each primitive allows the user to express how data is transformed during computation. DaPPA supports five primary data-parallel pattern primitives, including: (*i*) `map`, which applies a function f to each individual input element i, producing unique output elements $y_i = f(x_i)$; (*ii*) `filter`, which selects input elements based on a predicate; (*iii*) `reduce`, which reduces input elements to a scalar; (*iv*) `window`, which *maps* and output element as the *reduction* of W *overlapping* input elements; (*v*) `group`, which *maps* and output element as the *reduction* of G *non-overlapping* input elements. The user can combine all five data-parallel primitives to describe complex data transformations in an application. DaPPA is responsible for translating and parallelizing each data-parallel primitive to efficient CPU and UPMEM code.

Dataflow Programming Interface. DaPPA exposes a dataflow-based programming interface to the user (❷ in Figure 1). In this programming interface, the main component is the `Pipeline` class, which represents a sequence of data-parallel patterns that will be executed on the DPUs. A given `Pipeline` has one or more `stage`s. Each `stage` utilizes a given data-parallel pattern primitive to transform input operands following a user-defined computation. `Stage`s are executed in order, in a pipeline fashion.

Dynamic Template-Based Compilation. DaPPA uses a dynamic template-based compilation (❸ in Figure 1) to generate DPU code in two main steps. In the first step, DaPPA creates a base DPU code based on a basic skeleton of a DPU application. In the second step, DaPPA uses a series of transformations to (*i*) extract the required information that will be fed to the DPU code template from the user program; (*ii*) calculate the appropriate offsets used when managing data across MRAMs and WRAMs; and (*iii*) divide computation between CPU and DPUs.

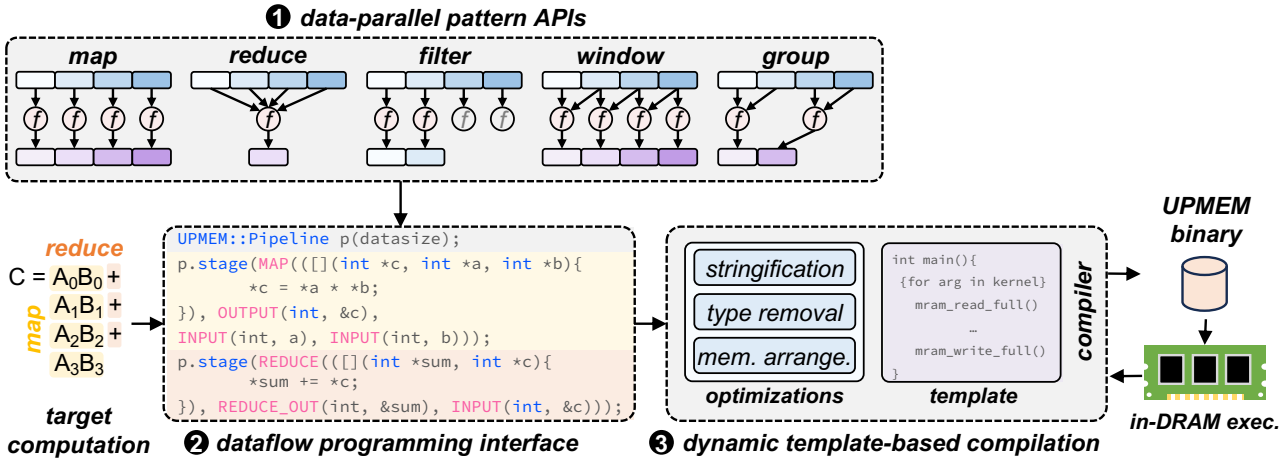

Figure 1: Overview of the DaPPA framework.

Putting All Together. Using DaPPA's data-parallel pattern APIs, data-flow programming interface, and dynamic template-based compilation, the user can quickly implement and deploy applications to the UPMEM system without any knowledge of the underlying architecture. Figure 1 showcases an example of implementing a simple vector dot product application using DaPPA. In this example, the user defines a `Pipeline` with two `stages`: a `map` stage and a `reduce` stage. DaPPA generates the appropriate binary for the UPMEM system, executes the target computation in the DPUs, and copies the final output from the DPUs to the CPU.

3. Key Results & Contributions

Methodology. To demonstrate DaPPA's benefits, we implement a subset of the workloads (i.e., vector addition, select, reduce, unique, imagine histogram small, and gemv) presented in the UPMEM-based PrIM benchmark suite [17] using our data-parallel pattern model. We conduct our evaluation on a UPMEM PIM system that includes a 2-socket Intel Xeon Silver 4110 CPU at 2.10 GHz (host CPU), standard main memory (DDR4-2400) of 128 GB, and 20 UPMEM PIM DIMMs with 160 GB PIM-capable memory and 2560 DPUs. We compare DaPPA's performance and programming complexity to that of the hand-tuned implementations present in PrIM.

Key Results. First, compared to the hand-tuned PriM workloads, DaPPA improves end-to-end performance by 2.1×, on average across all six workloads (min. 0.8×, max. 10.6×). DaPPA's performance improvement is due to code optimizations, such as parallel data transfer and workload partition between CPU and DPUs. Second, DaPPA *significantly* reduces programming complexity (measured using line-of-code) on average by 94.4% (min. 92.3%, max. 96.1%). We conclude that DaPPA is an efficient framework that eases the programmability of PIM architectures.

We make the following key contributions:

- To our knowledge, this is the first work to propose a data-parallel pattern-based framework to generate code for the UPMEM architecture *automatically*.
- We propose DaPPA (data-parallel processing-in-memory architecture), a framework that *automatically* distributes input and gathers output data, handles memory management, and parallelizes work across DPUs.
- We equip DaPPA with a series of code optimizations that improve the performance of workloads running on the UPMEM system.
- We evaluate DaPPA using six workloads from the PrIM benchmark suite, and we observe that DaPPA improves performance by 2.1× and reduces line-of-code by 94.4%, on average, compared to the hand-tuned PrIM workloads.

References

[1] S. Ghose *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, 2019.
[2] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021.
[3] G. F. Oliveira *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, 2021.
[4] S. Ghose *et al.*, "The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption," in *Beyond-CMOS Technologies for Next Generation Computer Design*, 2019.
[5] O. Mutlu *et al.*, "Enabling Practical Processing in and Near Memory for Data-Intensive Computing," in *DAC*, 2019.
[6] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[7] W. H. Kautz, "Cellular Logic-in-Memory Arrays," *IEEE TC*, 1969.
[8] H. S. Stone, "A logic-in-memory computer."
[9] UPMEM, "UPMEM Website," https://www.upmem.com, 2023.
[10] UPMEM, "Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper)," 2018.
[11] Y.-C. Kwon *et al.*, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.
[12] S. Lee *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021.
[13] L. Ke *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2021.
[14] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," arXiv:2105.03814 [cs.AR], 2021.
[15] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman London, 1989.
[16] M. Cole, "Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel Computing*, 2004.
[17] SAFARI Research Group, "PrIM Benchmark Suite," https://github.com/CMU-SAFARI/prim-benchmarks.