Towards a Java Virtual Machine for Processing-In-Memory

Kazuki Ichinose[1]
Fixstars Corporation, Tokyo, Japan

Shigeyuki Sato
The University of Electro-Communications, Tokyo, Japan

Tomoharu Ugawa
The University of Tokyo, Japan

Abstract

Processing-in-Memory (PIM) is a computing paradigm in which computation takes place in or near memory devices, offering high-bandwidth yet energy-efficient data-parallel processing. Real-world PIM systems have recently emerged, and SPMD-style programming in C is supported there. However, high-level object-oriented programming in managed languages has never been studied. Pursuing high-level programming for offloading Java applications to PIM processors, we are developing a Java framework to support it. As a status report on our project, we present our prototype Java VM built upon a real-world PIM system and experimentally demonstrate its scalability. The experimental results showed the potential of our Java VM on the PIM system with thousands of PIM processors.

2012 ACM Subject Classification Computing methodologies → Parallel programming languages; Hardware → Memory and dense storage; Software and its engineering → Interpreters

Keywords and phrases Java VM, Processing-in-Memory, Offloading, Data parallelism

Digital Object Identifier 10.4230/OASIcs.Programming.2025.2

Category Extended Abstract

Funding This work was supported by JSPS KAKENHI Grant Number 23K24822.

1 Introduction

Processing-in-Memory (PIM) [1] is a computing paradigm in which computation takes place in memory devices. It attracts much attention because of its high memory bandwidth and energy-efficient data-parallel processing capabilities. The recent emergence of the first real-world commercial PIM system, UPMEM PIM [11], has driven attention even more. UPMEM PIM can be modeled as a distributed system in a single computer, consisting of around 2500 PIM processors called DPUs, each of which has its own DRAM called MRAM of 64 MB though DMA. CPU can read/write this MRAM and let DPU execute small programs. Unfortunately, software systems for PIM are still limited. Although an MPI-like library [2] for low-level C programming in the SPMD (single program, multiple data) style was developed, high-level object-oriented programming in managed languages has never been studied.

For high productivity, we pursue high-level programming for offloading Java applications to DPUs. Specifically, Figure 1 shows our motivating example, which consists of three key concepts: 1) single user-defined classes are executable on both CPUs and DPUs; 2) objects are created and kept in DPU memory; 3) handlers for results on DPUs are controlled on CPUs. These three concepts are natural demands on using PIM systems for Java applications. However, they pose non-trivial challenges in Java because the existing efforts on computation

[1] This work was done at the University of Tokyo.

```
class MVContext
        extends DpuGroupContext {
  final int n;
  MVContext(int p, int n) {
    super(p); this.n = n;
  }
}
class Matrix {
  Matrix(MVContext ctx) {
    this(ctx.n);
  }
  Matrix(int size) { ... }
}
class Vector {
  Vector(int size) { ... }
  Integer dot(Vector rhs) { ... }
}
class MV {
  final MVContext ctx;
  MV(MVContext ctx) { ... }
  Vector mult(Matrix mat,
              Vector vec) { ... }
  Integer sqnorm(Vector vec) { ... }
}
```

(a) User-defined classes.

```
// Prepare a context of DPUs.
MVContext ctx = new MVContext(p, n);

// Create an object on DPUs.
MultiplexProxy<Matrix> mat
  = ctx.newObject(Matrix::new);

// Prepare an object on CPUs
Vector src = new Vector(n);
// Transfer an object from CPUs to DPUs
MultiplexProxy<Vector> vec
  = ctx.makeProxy(partition(src));

// Invoke MV.mult on DPUs, where the referents
// of result product are distributed over DPUs.
MultiplexProxy<Vector> prod
  = ctx.kernel(MV:mult, mat, vec);

// Invoke MV.sqnorm DPUs, where the results
// on DPUs are transferred to CPUs as a list.
List<Integer> partials
  = ctx.invoke(MV:sqnorm, prod);

// Aggregate partial results to the squared norm.
return partials.stream().sum();
```

(b) Host code of offloading to DPUs.

Figure 1 Example of high-level Java programming for PIM offloading: matrix-vector multiplication followed by squared norm.

offloading in Java were made for GPU computing[5, 3] and mobile computing [9], of which the characteristics differ from UPMEM PIM. For example, in GPU computing, it is unusual to keep data in GPU memory after computation, and rich computing capabilities are available; in mobile computing, thousands of processor cores do not work cooperatively in parallel for a single computing task – UPMEM PIM systems are designed for these very things.

This paper is a status report on the development of our Java framework for PIM offloading. We design and implement a prototype Java VM that enables offloading Java methods written in the SPMD style to DPUs on UPMEM PIM to perform scalable data-parallel processing (Section 2). We also experimentally demonstrate the scalability of our Java VM with a simple benchmark on the UPMEM PIM system (Section 3).

2 Basic Design and Prototype Implementation

Our programming framework consists of the compiler, the runtime system for the CPU and DPU sides (Figure 2). The compiler identifies the *entry methods* of Java programs (`app.java` in Figure 2), which are directly called from the CPU, and compiles each of them into separate binary for the DPUs. Each binary also contains the methods that may be directly or indirectly called from the entry method. For example, `MV.mult` and `MV.sqnorm` in Figure 1 are entry methods, and their binaries respectively contain `Vector.dot` as it is called therein. The runtime system for the DPU side is statically linked with each binary. The main program runs on the CPU. When the program calls an entry method through the host APIs, such as `ctx.kernel` and `ctx.invoke` in Figure 1, the CPU-side runtime system loads the corresponding binary to the DPUs and launches the DPUs.

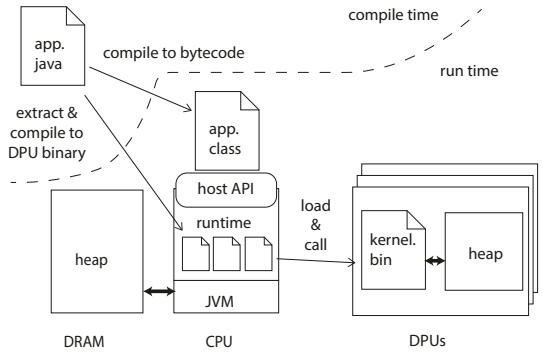

Figure 2 System Overview.

We implemented the prototype of the compiler and the runtime system for the DPU side. The implementation of the runtime system for the CPU side is limited. It only supports a single DPU, and the context is not supported. In the prototype, SPMD support for DPU programming is low-level; a high-level API will be designed on top of it in the future.

2.1 Heap Management

Each DPU has its own heap, which is managed by the DPU's runtime system. Objects created by the methods running on a DPU are created in the DPU's heap, located in MRAM. The contents of the DPU's heap are preserved across the calls to the entry methods. For example, in Figure 1, the vector object and its elements created in `mult` are stored in the DPU's heap. They can later be used in the call to `dot`.

2.2 Compiler

We compile each method into a C function. Then, the C functions and the runtime systems for the DPU side are compiled into a binary for DPUs by using the Clang compiler in UPMEM SDK [10].

The compilation of a method is straightforward. We concatenate the fragments of C code that correspond to the bytecodes of the method. Constants are embedded in the C code rather than implementing the constant pool. Java's local variables and entries of the operand stack are mapped to C's local variables as Proebsting et al. [8] did. To support garbage collection, checkpoints are inserted at method calls and backward jumps.

2.3 SPMD Support in DPU Programming

A DPU has 24 hardware threads, and using them is essential to achieve high performance. A DPU has a single in-order pipeline, and instructions of the same thread are dispatched 11 cycles apart [4]. Therefore, the pipeline cannot be filled with few threads.

To utilize hardware threads, we provide support for SPMD programming in DPU. When DPU is launched, the predefined number of hardware threads starts executing the entry method from the beginning. We provide APIs to obtain the number of threads and the ID of the calling thread. We also provide a barrier synchronization primitive.

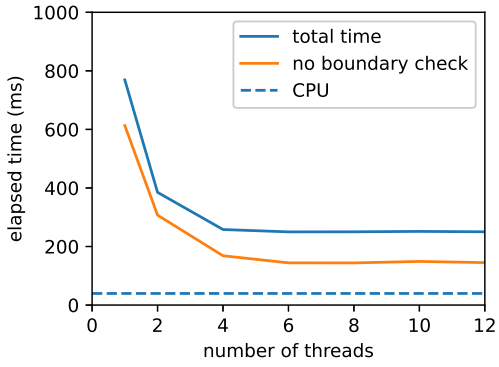

Figure 3 Execution Time for Matrix Vector Product.

3 Preliminary Evaluation

We evaluate the performance and the scalability of our prototype framework. We develop a matrix-vector multiplication program, which is offloaded to a single DPU. The matrix is a $24 \times 16,384$ matrix of non-negative integers. This is a submatrix obtained by dividing a $60,000 \times 16,384$ matrix evenly for 2500 DPUs. In this experimentation, we use the $(\mathcal{N}, \max, +)$-algebra instead of the classic $(\mathcal{N}, +, \times)$-algebra. This is because the DPU does not support multiplication. This program was compiled with UPMEM SDK version 2023.2.0 and executed in a single DPU running at 350 MHz of the UPMEM PIM system [11].

Figure 3 shows the execution time. The "total time" indicates the entire execution time of the multiplication. This shows that performance was improved up to 4 threads. One of the reasons why it did not scale up to 11 threads was that the DPU had a single DMA controller. Because objects were in MRAM, all access to objects involved DMA operations. Thus, the DMA controller was a bottleneck.

We also measured the execution time of the program that did not perform array-boundary checks, whose results are the "no boundary check" in Figure 3. This shows that the overhead of array-boundary checks was significant. Array boundary check accesses the size field of the array, which is in MRAM. The fact that the performance scaled up to 6 threads supports that the bottleneck was the DMA controller.

The dashed line in Figure 3 shows the execution time of the same computation using the standard Java 17 HotSpot JVM, but the size of the matrix was $60,000 \times 16,384$ because around 2500 DPUs are available in the UPMEM PIM system. For this measurement, we used two Intel Xeon Gold 6354 CPUs running at 3.0 GHz. This system has 72 hardware threads in total. This shows that offloading to DPU was 6.3× slower even if we had used all available DPUs. However, considering our implementation is straightforward, we believe that there is a large room for improvement. For example, eliminating the array boundary checks [12] would improve the performance significantly.

4 Conclusion

This paper has reported our prototype Java VM for PIM offloading. Although our Java VM has not yet sufficiently reached our goal of high-level Java programming, the experimental results showed its potential on UPMEM PIM. We plan to implement higher-level APIs like Java 8 Stream as part of our software stack by using Babylon [7] and PIM-oriented in-memory databases [6] as the primary application. Development of a more sophisticated compiler and runtime systems to achieve high performance is also listed in our future work.

References

1 Kazi Asifuzzaman, Narasinga Rao Miniskar, Aaron R. Young, Frank Liu, and Jeffrey S. Vetter. A survey on processing-in-memory techniques: Advances and challenges. *Memories - Materials, Devices, Circuits and Systems*, 4:100022:1–100022:11, 2023. doi:10.1016/j.memori.2022.100022.

2 Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu. SimplePIM: A software framework for productive and efficient processing-in-memory. In *Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, pages 99–111. IEEE, 2024. doi:10.1109/PACT58117.2023.00017.

3 Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. *Programming Heterogeneous Hardware via Managed Runtime Systems*. Springer Briefs in Computer Science. Springer, 2024. doi:10.1007/978-3-031-49559-5.

4 Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10:52565–52608, 2022. doi:10.1109/ACCESS.2022.3174101.

5 Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, PACT '15, pages 419–431. IEEE, 2015. doi:10.1109/PACT.2015.46.

6 Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. PIM-Tree: A skew-resistant index for processing-in-memory. *Proc. VLDB Endow.*, 16(4):946–958, 2022. doi:10.14778/3574245.3574275.

7 Oracle. Project babylon. https://openjdk.org/projects/babylon/, 2025.

8 Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (wat) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, USA, 1997. University of Arizona. URL: http://www.usenix.org/publications/library/proceedings/coots97/proebsting.html.

9 Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1:1–1:40, 2009. doi:10.1145/1555392.1555394.

10 UPMEM. Software development kit (sdk), 2024. URL: https://sdk.upmem.com/.

11 UPMEM. Upmem, 2024. URL: https://www.upmem.com/.

12 Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot™ client compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 125–133, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1294325.1294343.