Towards Throughput-oriented Sparse Matrix Vector Multiplication on a Processing-in-Memory System

Lukas Zink Juan Gómez Luna Christina Giannoula Onur Mutlu

Motivation

Use Cases: Sparse Matrix Vector Multiplication (SpMV) is a fundamental kernel for sparse neural networks, semiconductor design, chemical/physical optimizations and graph applications

Problem: Random accesses of the input vector make the kernel fundamentally memory bound on commodity systems.
The result is a low hardware utilization and inefficient computation

Our Goal

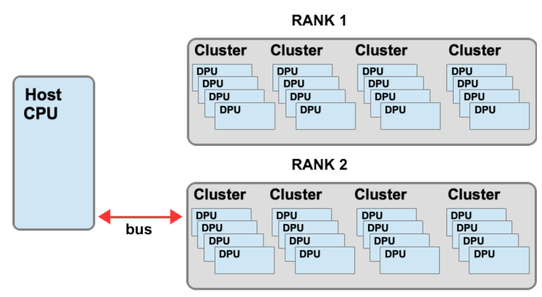
Perform many SpMVs in parallel on a PIM system to support concurrent tasks, users and SpMM workloads

Achieving a high throughput, high efficiency and low latency implementation

Taking advantage of the data locality, DRAM storage capacity and DPU Cores of the PIM System

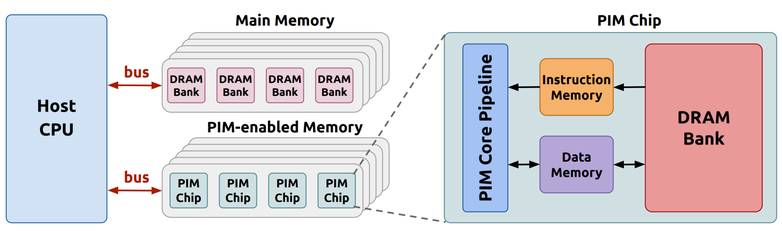
Analyzing transfer and kernel performance regarding matrix properties and partitioning formats and selecting the optimal kernel for a given SpMV

Cluster Mechanism



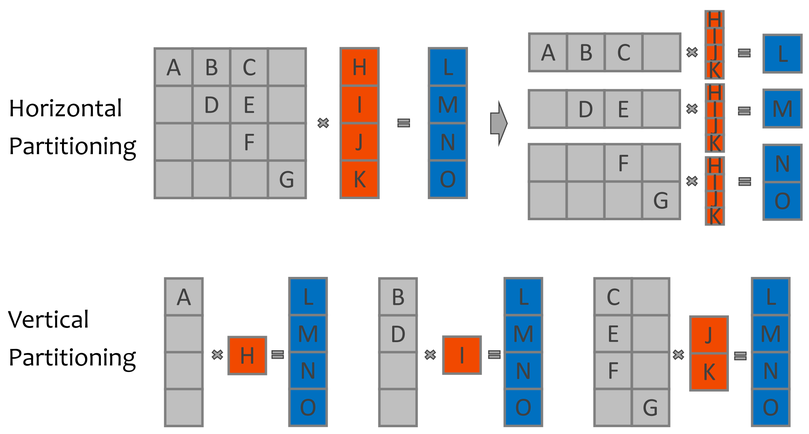
The mechanism separates the entire system into clusters of multiple DPUs where each Cluster then computes a single SpMV

State-of-the Art PIM System



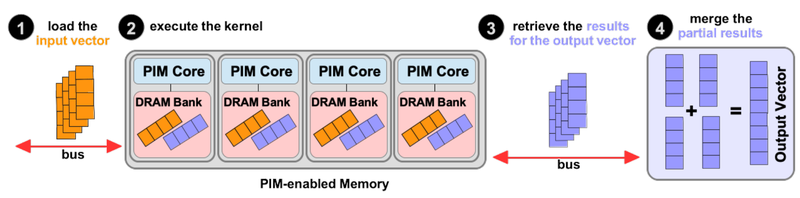
UPMEM PIM architecture with general-purpose processing cores called *DPUs*
- *64 DPUs /Rank , 2 Ranks /Module*
- 16 threads used
- 32-bit integer arithmetic
- 64-MB DRAM bank (*MRAM*)
- 64-KB scratchpad (*WRAM*)

Matrix Partitioning Techniques



SpMV on a PIM System

Matrix and/or Vector are partitioned before the transfer of matrix and vector data to the PIM memory can commence



Latency-Oriented Version Scaling Limitation

Less DPUs: execution time dominated by computation on the DPUs
More DPUs: execution time dominated by input vector transfers

Problem: Latency-Oriented Implementation does not scale beyond 64 DPUs per SpMV

Testing: Impact of Block Pattern on Performance

If a Matrix exhibits a block-pattern, a blocked partitioning format consistently performs better

For matrices without a block-pattern, traditional partitioning formats outperform

Testing: Prior knowledge about the matrix exhibiting a block pattern leads to a **13.3% lower execution time** on average

The matrix analyzer tool of the library can classify the block pattern property with an accuracy of 92% on the matrix test set

Recommendation: If matrix properties are known in advance, use according format. Analyzing properties prior to SpMV only beneficial if matrix is reused

Key Takeaways: SpMV on a PIM System

1. Balancing of non-zero matrix elements across DPUs crucial for low kernel execution time

2. Minimize total amount of data transferred to PIM memory to lower the overall execution time

3. Parallel transfers should be similar size and lock-free synchronization should be used on the UPMEM system

4. Between 2 and 64 DPUs should be used for a given SpMV depending on matrix size and performance goals

5. The CSR-NNZ and CSR-BLOCK kernels achieve the highest performance for SpMVs with 8 DPUs

Evaluation

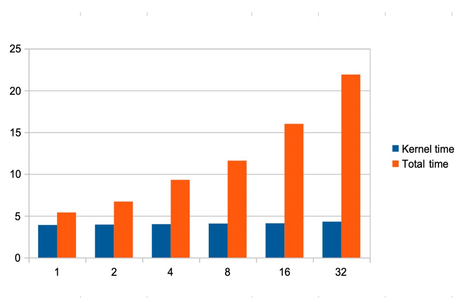
Testing Methodology

Testing was performed on a set of 12 matrices from the Texas A&M sparse matrix, with vastly different properties. The UPMEM system was configured with 20 PIM modules, resulting in 2496 DPUs running at 400mhz

Throughput Improvement

Comparison against the latency-oriented SpMV implementation by SparseP using the COO-NNZ Kernel
- throughput increases from **57x to 259x** depending on matrix size
-average throughput increase of **93x**
-latency increase by 3x

System Scalability

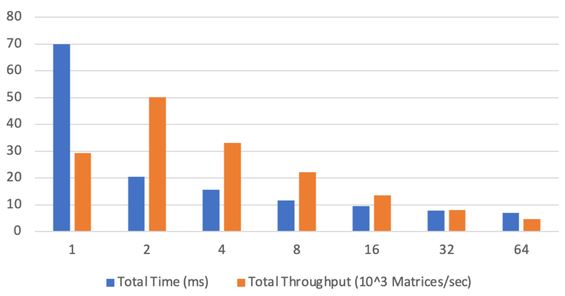


When keeping the workload per rank constant and scaling up the system it can be observed that total execution time increases by 400% when going from 1 to 32 ranks.
This can be attributed to the narrow memory bus

Throughput vs. Latency

Varying the cluster sizes changes the performance characteristic of the SpMV.
Decreasing the cluster size allows for **more throughput**
Increasing the cluster size can **improve latency**

Large cluster size: insufficient load per DPU
Small cluster size: many input vector transfers



Comparison Against CPU/GPU

GPU: NVIDIA A100@1095, cuSparse cuda 11.2:	182 Gflops
CPU: Intel Xeon 8280 Intel MKL:	5.4 Gflops
UPMEM 2496 DPU 400mhz s 32-bit float:	3.3 Gflops
UPMEM 2496 DPU 400mhz 8-bit int:	10.8 Gops

GPU: NVIDIA A100@1095, cuSparse cuda 11.2:	0.9% (0.002% tensor)
CPU: Intel Xeon 8280 Intel MKL:	16.35%
UPMEM 2496 DPU 400mhz kernel:	58-79%
UPMEM 2496 DPU 400mhz total:	32%

Conclusion

In conclusion, the throughput-oriented implementation succeeds at parallel computation of multiple independent SpMVs with a performance that rivals a state-of-the art CPU-based system. The PIM system executes the selected kernels at a high utilization and efficiency.
Limitations of the implementation are the arithmetic throughput limit of the DPU cores, narrow memory bus and emulated operations for larger data types. These can largely be attributed to the early stage of PIM technology development and are not fundamental limits of the technology itself.

Acknowledgement: We acknowledge support from the UPMEM Corporation for enabling access to their infrastructure