# Hardware-Software Co-Design of Iterative Filter-Update Numerical Methods Using Processing-In-Memory

Eric Tang
Carnegie Mellon University
Pittsburgh, USA
erictang@cmu.edu

Tianyun Zhang
Carnegie Mellon University
Pittsburgh, USA
tianyun2@andrew.cmu.edu

William Bradford
University of Virginia
Charlottesville, USA
wbradford@virginia.edu

Farzana Siddique
University of Virginia
Charlottesville, USA
farzana@virginia.edu

James Hoe
Carnegie Mellon University
Pittsburgh, USA
jhoe@andrew.cmu.edu

Kevin Skadron
University of Virginia
Charlottesville, USA
skadron@virginia.edu

Franz Franchetti
Carnegie Mellon University
Pittsburgh, USA
franzf@andrew.cmu.edu

## Abstract

Data movement is a key bottleneck in applications such as machine learning and scientific computing. Some software techniques address this by computing on subsets of data but this still requires reading the entire dataset to determine the subset. We propose a hardware-software co-design approach for iterative methods centered around two operations–filtering and updating. We introduce a domain-specific language that supports these computational patterns to enable PIM programming. Since filter and update are simple pointwise operations, PIM hardware requires only limited compute capability.

In this work, we investigate gradient descent for an ill-conditioned convex optimization function using this approach and map it to a PIM architecture using the PIMEval architectural simulator. Filter load and update store operations sparsify the data set by 83% while requiring as few as 1.5x more iterations to converge compared to traditional gradient descent approaches, with a net reduction in data movement of 3.9x.

## Keywords

in-memory computing, gradient methods

**ACM Reference Format:**
Eric Tang, Tianyun Zhang, William Bradford, Farzana Siddique, James Hoe, Kevin Skadron, and Franz Franchetti. 2025. Hardware-Software Co-Design of Iterative Filter-Update Numerical Methods Using Processing-In-Memory. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3731599.3767489

## 1 Introduction

Modern workloads such as neural networks, graph applications and (sparse) scientific computation are memory bound due to the large datasets and massive data movement that are required between main memory and the CPU. As a result, architects have turned to Processing-in-Memory (PIM) [18] — where memory-bound computations are done in DRAM — to take advantage of the larger internal memory bandwidth and reduce data movement needs [11]. However, due to tight area and power constraints and the high costs of logic in DRAM, it may be infeasible for these hardware platforms to support more complex arithmetic operations such as integer multiplication or double precision operations [6]. These limitations can be ameliorated by investigating workload requirements and restricting algorithms to leverage the benefits of PIM while placing more complex computations on the host processor (CPU or GPU).

**Related Work**. Optimization algorithms present in many applications have been structured using various approaches to reduce the need for computation and data movement. For optimization problems that utilize iterative methods to converge to a minimum, the core computation is a sparse matrix multiplied by a dense vector (SpMV) which is a memory-bound problem [2, 10]. To alleviate this, much prior work [12, 14] focuses on splitting the vector into subvectors, each associated with different subspaces that can be updated in parallel and independently. Even though the update computed in each iteration is no longer follows the globally optimal descent direction, the reduced data movement still enables the linear system to converge faster. However, since this approach is only explored in software, the sparsification step still requires transferring the vector across the memory bus in order to apply the filter.

By considering the memory bandwidth limitations and data requirements for iterative solvers, we recast some examples of these problems to reduce data movement and use PIM to perform some simple filter and update operations. For example, in gradient descent, each iteration only requires the update computation for the top 0.17% most significant components of the gradient to converge

[15]. While this update computation can be performed on the CPU, filtering the gradient to find these components must occur in memory as otherwise the entire vector would need to be transferred across the memory bus. Similarly, to avoid transferring the entire vector back and forth, the updates to both the solution vector and gradient in each iteration must be performed in memory. These filter and update operations are simple pointwise computations that can be performed by PIM hardware with limited compute capability.

The programmability of this novel architecture requires a new abstraction layer in order to allow hardware and software developers to innovate independently. The irregular patterns with which data is stored in DRAM and incomplete set of arithmetic operations that may be supported across different PIM architectures make it difficult to define an ISA which is portable across PIM devices. In order to port a single application such as a SpMV across architectures, hardware developers would need to support a large number of different data structures or the software developer would need to reimplement the application for each architecture at a wide range of sizes. Due to the need for managing the long latency of each instruction and large amount of concurrent and independent operations, we take inspiration from databases when designing this API. To this end, we have developed the Memory Query Language (MQL) which provides a set of basic operations that allow programmers to specify the computation to be done in memory. This language introduces two new operations - Filter and Update which are needed to obtain performance in software but also provides flexibility in how these operations are supported in hardware.

We investigate the effectiveness of this approach by using gradient descent to find the minima of an ill-conditioned objective function as a first example. Each iteration of the gradient descent algorithm is sparsified using a filter to dramatically reduce the amount of data required to compute the update for each iteration. For these filters, the amount of data moved per iteration and overall (i.e., to reach the specified convergence criterion) is reduced by 5.8-10x and 3.9x respectively. Meanwhile the number of iterations required to converge is within 1.5x of the baseline gradient descent algorithm. This demonstrates that for memory-bound iterative algorithms where data transfer is the limiting factor, PIM hardware with limited computed capability can be utilized to efficiently sparsify the data. The reduced data per iteration leads to an increase in the number of iterations to converge but the volume of data that is transferred to the host is significantly reduced.

**Contributions**. This paper makes the following contributions:

- Proposes iterative numerical methods be recast into filter-update operations to leverage PIM hardware and proposes an API to map these examples to a range of PIM architectures.
- Analytic assessment of data movement and performance for filter-update operations, using gradient descent as a case study. These results show substantial savings in data movement while requiring only relatively cheap operations in PIM hardware.

## 2 Background

Modern PIM architectures are programmed in a manner that forces application developers to be aware of the PIM hardware. This largely stems from the fact that depending upon where in memory the compute unit is placed (rank level, bank level, subarray level) greatly changes the types of instructions the programmer is allowed to use.

**PIMeval**. PIMeval [17] is an extensive modeling framework for PIM that provides a high-level C++ API to write PIM applications. These APIs are then mapped to PIM-specific execution time and energy model. PIMeval currently comes with three different general-purpose PIM architecture: 1. subarray-level bit-serial [4, 7, 9, 17], 2. subarray-level bit-parallel [8], and 3. bank-level [3] PIM. This paper uses PIMeval as the modeling framework due to its flexibility and generality in capturing a wide range of PIM architectures. In addition, PIMbench [17] – a diverse benchmark suite for PIM includes *filter-by-key* – a benchmark that filters a set of key-value pairs. PIMbench implements the *filter* operation on PIM and the *aggregate* operation on the host, attributing this design choice to the high cost of inter-PIM communication. Given the similarity between the *filter* phase of our target application and that in PIMbench, we adopt a similar decomposition. Furthermore, since PIMbench is built on top of PIMeval, using PIMeval as our modeling framework not only enables comparison across a wide range of PIM architectures, but also facilitates direct comparison with PIMbench.

**Gradient Descent**. Gradient descent is a first-order iterative optimization algorithm used to minimize a differentiable objective function. Given a function $f(x)$, with parameters $x \in \mathbb{R}^d$, the fundamental gradient descent iteration updates the parameters in the direction opposite of the gradient of the objective function $\nabla f(x_t)$ at the current parameter values $x_t$:

$$x_{t+1} = x_t - \eta_t \nabla f(x_t)$$

where $\eta_t > 0$ is the learning rate (step size) at iteration $t$.

In practice, computing the full gradient can be expensive, especially when the dataset is large. Stochastic gradient descent (SGD) addresses this by approximating the gradient using a single data sample or a small subset of the data (a "mini-batch"). Denoting the stochastic gradient at iteration $t$ as $g_t$, the update rule becomes: $x_{t+1} = x_t - \eta_t g_t$. To accelerate optimization, SGD is often parallelized by distributing work across multiple nodes. The overall objective function is the average of objective functions computed on local data partitions at each of the $N$ nodes:

$$f(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x)$$

At each iteration $t$, every node $i$ computes a local stochastic gradient $g_{i,t}$ based on its local data and the current parameters $x_t$. Then, local gradients are aggregated across nodes to compute the update applied to $x_t$.

While distributed SGD can speed up computation, it introduces significant communication overhead, since nodes need to exchange local gradients at each iteration. For functions with millions or billions of parameters, this communication may become a performance bottleneck [16][1].

## 3 PIM Compute Stack

To program these novel architectures, a compute stack must be designed in order to manage the complexity of using the hardware,

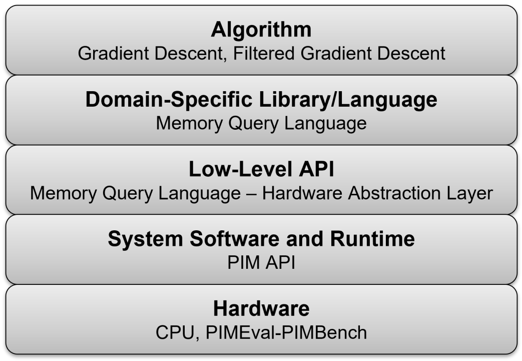

Algorithm
Gradient Descent, Filtered Gradient Descent
Domain-Specific Library/Language
Memory Query Language
Low-Level API
Memory Query Language – Hardware Abstraction Layer
System Software and Runtime
PIM API
Hardware
CPU, PIMEval-PIMBench

**Figure 1: PIM Compute Stack**

simplify programming and enable scalability and portability. When looking at iterative methods, these programs begin with the algorithm design which is done using the foundational mathematics for analyzing the data. Following this, the mathematical operations in the algorithm must be translated into a program that the machine can execute.

Iterative methods often feature recurring compute patterns that are well-suited for a domain-specific language (DSL) to target PIM architectures. Inspired by database queries agnostic to how data is stored in memory, we call this DSL the Memory Query Language (MQL). MQL is then broken down into a lower-level representation, the MQL Hardware Abstraction Layer (MQL-HAL), which represents an abstract PIM device. This layer exposes key in-memory compute operations to the program, and is subsequently lowered to a PIM API [17] that can target a range of PIM hardware.

**Memory Query Language (MQL).** To ensure portability across different PIM devices, the Memory Query Language (MQL) isolates the application description from the low-level data representation and computation details. Inspired by SQL, MQL utilizes a declarative programming model where users select regions of memory (instead of tables) and describe the computation to be performed on the data at those addresses. The system captures these semantics in an abstract syntax tree, which can then be used to target a code generation system like SPIRAL to generate instructions for a specific PIM architecture.

While PIM systems allow bandwidth-limited algorithms to exploit high internal DRAM bandwidth, a method is needed to efficiently handle the sparse results that must be read or written by the CPU. MQL solves this by introducing two operations: *filter*, which outputs sparse data from a dense region, and *update*, which applies a sparse input to a dense region.

**MQL Hardware Abstraction Layer.** To achieve good performance, high-level MQL commands are implemented through the Memory Query Language Hardware Abstraction Layer (MQL-HAL). MQL-HAL targets an abstract PIM device, exposing pertinent hardware details to the program. In addition to filter and update operations, this layer introduces an API for promoting and demoting data to PIM-enabled memory.

At the MQL-HAL level, the filter and update operations require specific parameters, such as the size of the memory region, to expose the necessary hardware details. The filter operation applies

```
CSR filtered_grad, filtered_grad_update; // Host side CSR struct
for (int iter = 0; iter < N_ITERATIONS; iter++) {
    // SELECT *
    // FROM gradient_pim
    // WHERE x > THRESHOLD_VAL
    filtered_grad = pimFilter(gradient_pim, ABS_GREATER_THAN,
      THRESHOLD_VAL);

    // Host side update computation
    filtered_grad_update.nnz = filtered_grad.nnz;
    filtered_grad_update.keys = filtered_grad.keys;
    filtered_grad_update.values = (GD_Precision*)malloc(
      filtered_grad.nnz * sizeof(GD_Precision));
    for (int i = 0; i < filtered_grad.nnz; i++) {
        filtered_grad.values[i] *= LR;
        filtered_grad_update.values[i] = filtered_grad.values[i] *
      gd_coeffs[filtered_grad.keys[i]];
    }

    // UPDATE x_pim
    // SET value = value - filtered_grad.value
    // FROM filtered_grad
    // WHERE x_pim.index = filtered_grad.index
    pimUpdate(x_pim, filtered_grad, SUB);

    // UPDATE gradient_pim
    // SET value = value - filtered_grad_update.value
    // FROM filtered_grad_update
    // WHERE gradient_pim.index = filtered_grad_update.index
    pimUpdate(gradient_pim, filtered_grad_update, SUB);

    free(filtered_grad_update.values);
}
```

**Listing 1: MQL-HAL program for gradient descent with SQL comparison for each PIM function.**

a predicate to each element in a PIM memory region, outputting those that satisfy the condition in a sparse data structure. Common operations (e.g. greater than, less than, equals) are provided as built-in functions, but user-provided functions could be passed in using a library that serves as a specification for the computation to be performed in PIM. This filter operation takes in the following arguments:

(1) A target PIM memory region
(2) A sparse data structure that defines the output format, e.g. index-only, index and data, and compressed sparse row (CSR).
(3) A predicate in the form of a stateless function that takes in a single value from a memory address as input.

The update operation modifies values in PIM-enabled memory by applying a sparse data structure. The elements of the sparse input are matched to the corresponding dense PIM memory region by their index. In the sparse data structure, the index of each piece of data is passed in where as in the dense data structure the index is determined based on the position of the data in memory. This process is defined by:

(1) A target PIM memory region
(2) A sparse input data structure
(3) A binary update function that takes an element from the sparse data structure and a corresponding element from the dense PIM memory region to produce an output.

Like the filter operation, the update operation can use both built-in functions and user-provided functions to define the update function.

MQL-HAL is then further lowered to a PIM API which compiles to target certain PIM architectures. For user-provided functions, tools like LibraryX [13] can perform semantic capture and generate a dataflow graph of the computation. This representation can then be used by the SPIRAL [5] code generation system for analysis and optimization. For supporting new PIM architectures, this low-level API simply needs to be reimplemented using the new PIM target's ISA.

## 4 Co-Design of Algorithm and Hardware

To mitigate the communication bottleneck, various gradient compression techniques have been proposed. Among these, we primarily consider gradient sparsification methods [15], i.e. communicating only a subset of the gradient elements instead of the full, dense gradient vector. In particular, we experiment with magnitude-based filtering, where the importance of a gradient component is determined by its magnitude.

When implementing the algorithm, we can use the PIM-based filter and accumulate operations to (1) reduce the amount of data that needs to be moved from memory to CPU, and (2) take advantage of massively parallel computations of relatively simple arithmetic operations. The filter operation calculates an absolute value greater than or less than comparison of many values at once, which may make it amenable to offload to PIM architectures. This significantly decreases the amount of communicated data, as only the gradient components that meet a threshold are brought to the processor. The accumulate operation performs the update to parameters, which is a vector element-wise addition or subtraction. Again, this potential for parallel computation makes offloading the computation to PIM an attractive choice.

**Designing a PIM Accelerator**. To glean some understanding of how such an algorithm would perform on a PIM device, we have designed a bank-level PIM accelerator which can support filter and update operations. This architecture places a single SIMD processor with four 32-bit lanes at each bank interface, matching the 128-bit wide bank interface (GDL width, shown in Fig. 2)[1]. Each of these lanes has access to three registers (regA, regB, and regC), but future cost-benefit analysis may indicate that four registers would better suit this architecture, since this would allow us to reduce the number of reads per iteration of the Update Algorithm (shown below and elaborated upon in Algorithm 2). Bank-level PIM was selected because it has the most direct ability to perform limited random accesses within a bank, but subarray-level PIM architectures may yield better parallelism, provided they can efficiently support random accesses.

**Filter and Update Algorithms on PIM Hardware**. In this example, we aim to filter a vector to select elements that are greater than or equal to a given comparison value, however other filtering functions could also be implemented. The algorithm enumerated below should be executed on each SIMD lane on each bank. We distribute the values array evenly across all banks and store the index of the first element in each bank, which we have denoted as the "starting index" below in Algorithm 1.

Below, in Algorithm 2, we enumerate the Update algorithm. This algorithm cannot be efficiently parallelized between SIMD lanes on our current bank-level architecture, since different lanes may need to read or store non-adjacent values to update them. In response to this, we propose a subset of instructions that do not permit lanes 1-3 to write to the row buffer. Similar to warp divergence in GPU programs, all lanes will perform the same operations in lock-step, but only lane 0 will write its results.

[1]The SIMD width of the bank unit should be adapted to the memory technology; e.g., 256-bit and 8-wide SIMD for GDDR

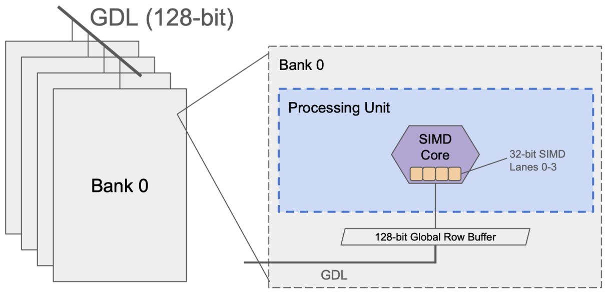


**Figure 2: The 128-bit global data line (GDL) connects to each bank, joining the SIMD core corresponding to each bank through a 128-bit global row buffer. The SIMD width must match the size of the global row buffer, which must match the size of the GDL.**

**Algorithm 1** Filter Algorithm

```
starting index ⟵ the first index on the current bank
values ⟵ a vector of values to be filtered
regA ⟵ comparison value
regB ⟵ index corresponding to SIMD lane ID (0-3)
while regB ≤ length(values) do
    if values[regB] ≥ regA then
        send regB + starting index to host
        send values[regB] to host
    end if
    regB += 4
end while
```

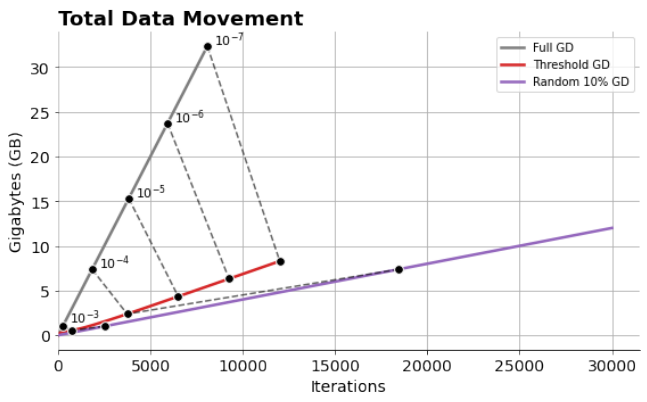
**Algorithm 2** Update Algorithm

```
starting index ⟵ the first index on the current bank
values ⟵ a segment of the values array
update data ⟵ interleaved key-value pairs to update values
regB ⟵ index value (starting at 0)
while regB ≤ length(update data) do
    regA ⟵ the key of update data[regB]
    regC ⟵ the value of update data[regB]
    regA -= starting index
    values[regA] -= regC
end while
```

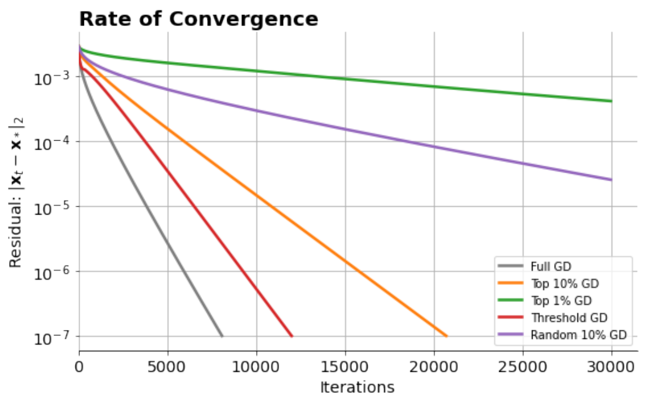
It may become apparent that, in the update algorithm, the line in which the starting index is subtracted from regA will result in one additional read per iteration of the while loop. Pending further cost-benefit analysis, this may legitimize the inclusion of one additional register per SIMD lanes.

## 5 Evaluation

To validate this approach, we investigate gradient descent and utilize various filters to reduce the amount of data movement required. For each of these filters, we observe the number of iterations required before convergence along with the amount of data transferred per iteration along with the total amount of data transferred before convergence. A random quadratic objective function of the



**(a) Cumulative data movement until convergence over iterations for different gradient descent methods with varying sparsification strategies. The points on each line mark the iteration where the residual falls below $10^{-j}$ for j from 3 to 7. The threshold-based filtering approach requires least data movement for each isoline with as much as 3.9x less when compared to full gradient descent. The random filtering approach requires about the same amount of data movement as Full GD but 10x more iterations.**



**(b) Convergence behavior of gradient descent under different gradient sparsification strategies on an ill-conditioned quadratic objective function. The residual computes the error between the value of $x_t$ in iteration t and the known minima $x_*$. $x_t$ is considered to have converged when the residual $\leq 10^{-7}$. The filter approaches both converge at a rate comparable to the full gradient descent while requiring less data movement.**

**Figure 3: The top k approaches utilize the k largest elements of the gradient while the threshold gradient descent approach uses all gradient values above a certain threshold and the random approach selects a random subset of the vector. While top k filters are able to converge in software, they are difficult to implement in PIM. A threshold filter, which can map to PIM hardware more easily, is able to converge in fewer iterations compared to the top k filter and reduces the amount of data transferred per iteration by 5.8x and in total by almost 4x.**

form: $f(x) = \frac{1}{2}\sum_{i=1}^{n} c_i x_i^2$ where $x \in \mathbb{R}^n$ is the optimization variable. $c$ is initialized using values distributed across a logarithmic scale to create an ill-conditioned problem. This helps simulate a common problem in optimization where some dimensions have a steeper curvature than others and thus produce larger gradients from small perturbations of $x$. This approach serves to highlight the effects of using a filter to sparsify the problem.

**Data Movement Reduction Analysis**. To quantify the benefits of employing a magnitude-based filtering and sparse updates executed in PIM, we compare the DRAM-CPU data traffic against a traditional dense CPU-based update scheme.

We define the following parameters for our data movement analysis: $D$ denotes the length of the parameter/gradient vector in DRAM, $\beta$ represents bytes per numeric value (e.g. 4 bytes for FP32), and $\alpha$ indicates bytes per index in sparse representation (4 bytes for int32). The magnitude filtering threshold is $\tau$, while $p_w$ and $p_g$ represent the pass ratios (as fractions of $D$) for the weight update vector and gradient returned to PIM, respectively. We use $x_t$ for the parameter vector at iteration $t$ and $g_t$ for the general gradient at iteration $t$, with learning rate $\eta$.

For conventional dense implementations, we consider two scenarios. In the *best case*, the gradient $g_t$ is produced and applied while cached in the processor, requiring only parameter reads and writes which gives: $B_{\text{dense(best)}} = 2\beta D$. In the *worst case*, the gradient spills to DRAM before the update, requiring additional gradient storage and retrieval which leads to: $B_{\text{dense(worst)}} = 4\beta D$.

Our PIM-based approach filters the dense gradient using the magnitude threshold $\tau$, with the CPU performing scaling operations on sparse values before PIM scatter-adds them back. We analyze two strategies for transferring data between the CPU and PIM. When both *indices and values* are transmitted bidirectionally, the data movement requirement is $B_{\text{PIM(idx+val)}} = 2(\alpha + \beta)(p_w + p_g)D$. Alternatively, for some applications it is possible to use an *index-only outbound* strategy, where only indices are sent from DRAM to CPU, which results in: $B_{\text{PIM(idx-only)}} = \alpha p_g D + \beta(p_g + p_w)D$.

The data movement reduction that can be achieved for *indices and values is* and *index-only outbound* scenario are:

$$\text{Data Transfer Reduction}_{\text{PIM(idx+val)}} = \frac{B_{\text{dense}}}{2(\alpha + \beta)(p_w + p_g)D}$$

$$\text{Data Transfer Reduction}_{\text{PIM(idx-only)}} = \frac{B_{\text{dense}}}{\alpha p_g D + \beta(p_g + p_w)D}$$

assuming $\alpha = \beta$ (32-bit indices and values) and setting $p_w = p_g = p$ for simplicity. The analysis demonstrates that with 90% sparsity ($p = 0.1$), our PIM pipeline reduces external traffic by 5–10× for the indices + values approach and up to 13× for the index-only approach. These gains increase significantly as sparsity deepens, making the approach particularly attractive for applications with highly sparse gradient updates.

**Filtering Approaches**. We explore three filtering strategies to investigate the tradeoffs between effectiveness and hardware feasibility. The top 10% filter selects all components of the gradient that correspond to the top 10% of gradient magnitudes. Although this filter cannot be easily implemented in PIM, it serves to highlight the potential effectiveness of always selectively updating the most significant components. The hard threshold filter, which can be easily implemented in PIM, selects all values of the gradient that are above a certain threshold. This hard threshold is determined by investigating the initial values of the gradient vector and setting a threshold that would select about 10% of the data. The hard threshold is then lowered by 1% after each iteration where no gradient values are above the threshold. The random 10% filter serves as a baseline by selecting a random 10% subset of indices in each iteration, independent of gradient magnitude. The results for each of these filters can be seen in Figure 3.

**Table 1: Convergence and Data Movement for Different Filtering Techniques**

| Method | Iterations to Converge | Updates Per Iteration | Total Updates (GB) |
|---|---|---|---|
| Full | 8,083 | 1,000,000 | 32.3 |
| Top 10% | 20,720 | 100,000 | 8.28 |
| Hard Threshold | 12,007 | 172,624 | 8.29 |
| Random 10% | N/A | 100,000 | N/A |

**Potential Performance Gain**. The hard threshold filtered gradient descent approach converges to the same tolerance as full gradient descent while transferring significantly less data. As seen in Table 1, in order to converge to $10^{-7}$, machine epsilon for 32-bit floats, the full gradient descent approach requires about 8,000 iterations and transfers 32 GB of data across the memory bus. For a hard threshold gradient descent approach, only 8 GB of data is transferred while the number of iterations increases by 50%. Using a random filter which avoids using PIM, the residual never converges. For the lower tolerances where the random filter converges, the same amount of data as the full gradient descent is transferred while requiring 10x more iterations.

For these applications with sufficient parallelism and low arithmetic intensity, data movement time can be used as a first-order model to approximate performance. In this model, performance is estimated as the sum of the CPU compute time, data movement time, and PIM compute time if applicable. Here we assume the problem is large enough to saturate both the DRAM bandwidth and PIM internal bandwidth. From this we get the following equation: $T = I(\frac{mF}{P} + \frac{m}{B} + \frac{D}{B_{PIM}})$ where $I$ is the number of iterations until convergence, $m$ represents the total bytes transferred per iteration, $B$ is the bandwidth (bytes/sec) of the system, $F$ is the number of floating-point operations required on the CPU for each byte for a given objective function, $P$ is the number of floating point operations per second the CPU is capable of, $D$ is the size of the vector, and $B_{PIM}$ is the internal bandwidth for PIM inside DRAM. This model assumes that for the filter operation the PIM computation on each bank can overlap with the data movement time of the filtered result that is processed on the host side. This is due to the fact that the PIM compute unit in each bank processes an independent set of data, allowing each unit to work in parallel. For update operations, since the updated values are written to DRAM, the PIM compute unit of each bank is able to independently fetch and write back the updated values. The internal bandwidth for PIM can be approximated with $B_{PIM} = BXG$ where B is the external DRAM bandwidth, X is the bank-level parallelism, and G is the global data line.

We estimate the performance for a system with an Intel Xeon E5-2640v4 with DDR4 memory using the results from the full gradient descent and filtered gradient descent. These results show hard threshold gradient descent could lead to a 3.9x speedup over the full gradient descent. In this system, the data movement time dominates the total runtime of the traditional gradient descent approach. For in memory computation, the PIM latency does not significantly change application runtime since the internal PIM bandwidth that can be utilized to perform filtering and update operations on the entire dataset is 128 times faster than the external DRAM bandwidth.

For filtering approaches like Top 10% which are unable to use PIM hardware, the whole dataset must be transferred each iteration while the amount of computation is reduced. In this case, the reduced amount of computation leads to a slow down since this approach will require more iterations. For memory-bound iterative methods which can be cast into filter/update computational patterns, the sparsification of each iteration must be balanced with the increased number of iterations that will be required in order to achieve good performance.

## 6 Conclusion

In this work, we proposed a hardware-software co-design approach to accelerate iterative methods by leveraging the filter and update patterns commonly found in numerical algorithms, which are memory-bound. By offloading these memory intensive operations to PIM hardware, the amount of data transferred across the memory bus is significantly reduced while maintaining similar convergence behavior. To support this model, we introduce a PIM compute stack to provide a domain-specific abstraction that allows developers to express compute in memory in a declarative fashion that is portable across PIM platforms. Through analytical modeling and experimentation, we have demonstrated for gradient descent, simple magnitude-based gradient filters only increase the number of iterations until convergence by 1.5x, while reducing the total data transferred by 3.9x. As future work, we plan to explore mapping larger machine learning applications and numerical methods to this compute model and demonstrate this compute model for a wider range of algorithms. From the hardware approach, we plan on performing more detailed performance and power modeling of the portion of the algorithm performed in a range of PIM architectures.

## Acknowledgments

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and the U.S. Department of Energy under award number DE-AC05-00OR22725. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy and DARPA.

## References

[1] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Sarit Khirirat, Nikola Konstantinov, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 5977–5987.

[2] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. doi:10.1145/1654059.1654078

[3] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or not for emerging general purpose processing in DDR memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture.* 231–244.

[4] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning. 2017. In-memory intelligence. *IEEE Micro* 37, 4 (2017), 30–38.

[5] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. doi:10.1109/JPROC.2018.2873289

[6] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), 3:1–3:19. doi:10.1147/JRD.2019.2934048

[7] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez Luna, and Onur Mutlu. 2021. SIMDRAM: An end-to-end framework for bit-serial SIMD computing in DRAM. *arXiv preprint arXiv:2105.12839* (2021).

[8] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R Stan, and Kevin Skadron. 2020. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 556–569.

[9] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. 2017. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 288–301.

[10] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 273–282. doi:10.1145/2464996.2465013

[11] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. *CoRR* abs/2012.03112 (2020). arXiv:2012.03112 https://arxiv.org/abs/2012.03112

[12] Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization* (2nd ed.). Springer, New York, NY.

[13] Sanil Rao. 2025. LibraryX: A Framework for Cross-Library-Call Optimization. (5 2025). doi:10.1184/R1/29089637.v1

[14] Youcef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[15] Atal Sahu, Aritra Dutta, Ahmed M. Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. 2021. Rethinking gradient sparsification as total error minimization. In *Advances in Neural Information Processing Systems,* M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 8133–8146.

[16] Atal Narayan Sahu, Aritra Dutta, Ahmed M. Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. 2021. Rethinking gradient sparsification as total error minimization. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 8133–8146.

[17] Farzana Ahmed Siddique, Deyuan Guo, Zhenxing Fan, Mohammadhosein Gholamrezaei, Morteza Baradaran, Alif Ahmed, Hugo Abbot, Kyle Durrer, Kumaresh Nandagopal, Ethan Ermovick, et al. 2024. Architectural Modeling and Benchmarking for Digital DRAM PIM. In *2024 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 247–261.

[18] H. S. Stone. 1970. A logic-in-memory computer. *IEEE Transactions on Computers (TC)* 100, 1 (1970), 73–78.