

# DARE: An Irregularity-Tolerant Matrix Processing Unit with a Densifying ISA and Filtered Runahead Execution

Xin Yang  
Fudan University  
Shanghai, China  
yangx23@m.fudan.edu.cn

Xin Fan  
HKUST  
Hong Kong, China  
xfanat@connect.ust.hk

Zengshi Wang  
Fudan University  
Shanghai, China  
wangzs23@m.fudan.edu.cn

Jun Han\*  
Fudan University  
Shanghai, China  
junhan@fudan.edu.cn

**Abstract**—Deep Neural Networks (DNNs) are widely applied across domains and have shown strong effectiveness. As DNN workloads increasingly run on CPUs, dedicated Matrix Processing Units (MPUs) and Matrix Instruction Set Architectures (ISAs) have been introduced. At the same time, sparsity techniques are widely adopted in algorithms to reduce computational cost.

Despite these advances, insufficient hardware–algorithm co-optimization leads to suboptimal performance. On the memory side, sparse DNNs incur irregular access patterns that cause high cache miss rates. While runahead execution is a promising prefetching technique, its direct application to MPUs is often ineffective due to significant prefetch redundancy. On the compute side, stride constraints in current Matrix ISAs prevent the densification of multiple logically related sparse operations, resulting in poor utilization of MPU processing elements.

To address these irregularities, we propose DARE, an irregularity-tolerant MPU with a Densifying ISA and filtered Runahead Execution. DARE extends the ISA to support densifying sparse operations and equips a lightweight runahead mechanism with filtering capability. Experimental results show that DARE improves performance by  $1.04\times$  to  $4.44\times$  and increases energy efficiency by  $1.00\times$  to  $22.8\times$  over the baseline, with  $3.91\times$  lower hardware overhead than NVR.

**Index Terms**—CPU, Matrix Processing Unit, Sparse Deep Neural Networks, Runahead Execution

## I. INTRODUCTION

Deep neural networks (DNNs) are widely applied across domains and have shown strong effectiveness. Among their various operations, general matrix multiplication (GEMM) is one of the most compute- and energy-intensive. To reduce the computational cost of GEMM, extensive efforts target both hardware and algorithm design. On the hardware side, as DNN workloads increasingly execute on CPUs [1]–[3], vendors such as Intel and Arm have introduced dedicated matrix Instruction Set Architectures (ISAs), including Advanced Matrix Extensions (AMX) [4] and Scalable Matrix Extensions (SME) [5], along with specialized matrix processing units (MPUs) [6]–[9]. On the algorithm side, sparsity has been widely explored, leading to GEMM variants such as sparse matrix–dense matrix multiplication (SpMM) and sampled dense matrix–dense matrix multiplication (SDDMM).

\* Corresponding author. This work was supported by the National Natural Science Foundation of China under Grant 61934002 and 62234008.

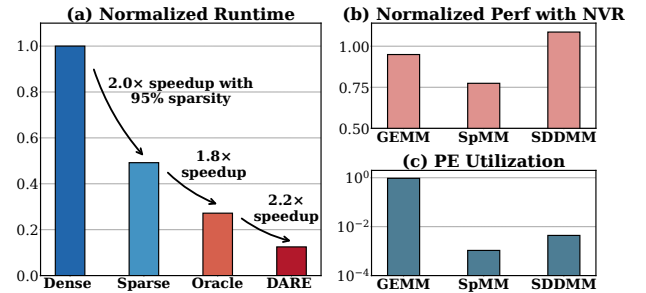


Fig. 1: (a) Runtime of sparse SDDMM normalized to that of dense GEMM on an AMX-like MPU. Oracle assumes a cache without misses. (b) Performance of an MPU with NVR [15], normalized to a baseline MPU without NVR. (c) Processing Element (PE) utilization in a systolic array under various workloads, defined as the ratio of active PEs to the total number of PEs during execution.

However, insufficient hardware–algorithm co-optimization often results in suboptimal performance. Fig. 1(a) shows that even with 95% sparsity, SDDMM achieves only a  $2.0\times$  speedup over dense GEMM on an AMX-like MPU. This gap arises from two issues: memory inefficiency and computation irregularity. On the memory side, sparse DNNs typically incur indirect accesses, leading to high cache miss rates and memory stalls. With an ideal zero-miss cache (Oracle in Fig. 1(a)), performance could improve by up to  $1.8\times$ . On the compute side, sparse DNNs often employ unstructured sparsity to preserve accuracy, while hardware engines such as systolic arrays, composed of mesh-connected processing elements (PEs), prefer regular and structured sparsity. This mismatch leads to poor hardware utilization and limited acceleration.

For memory inefficiency, runahead execution [10]–[14] is a well-established CPU prefetching technique that mitigates irregular memory accesses by pre-executing stalled instructions. NVR [15] first adopted this idea in Neural Processing Units (NPU) to improve sparse DNN performance. However, Fig. 1(b) shows that directly applying it to Matrix Processing Units (MPUs) may result in performance degradation for workloads such as GEMM and SpMM, compared to an MPU without NVR. In addition, NVR requires 9.72 KB of hardware state, which is a nontrivial overhead for MPUs.

For computation irregularity, Fig. 1(c) shows that sparse workloads often cause low Processing Element (PE) utilization

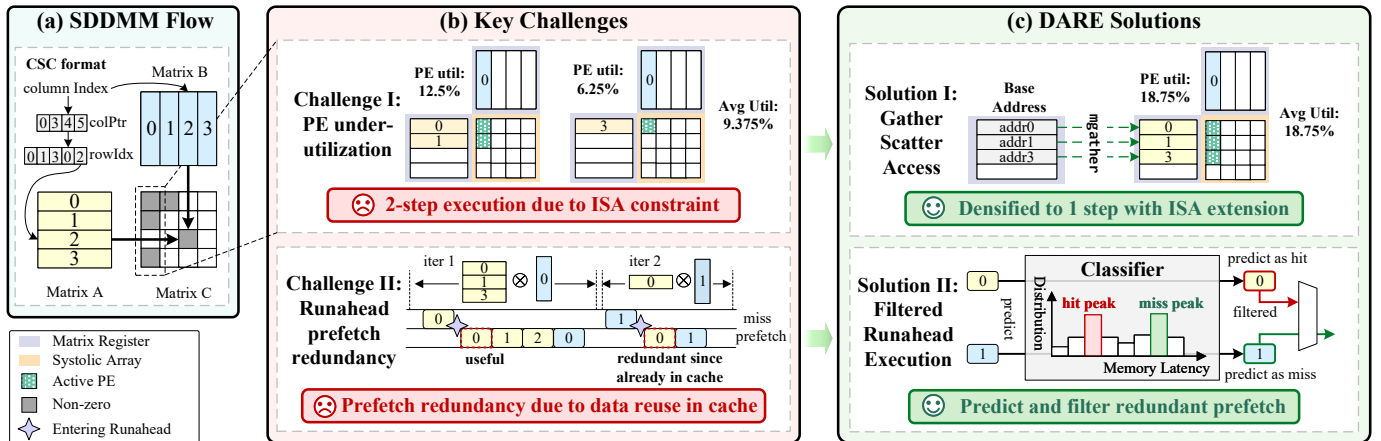


Fig. 2: (a) An example computation flow of SDDMM. Only the non-zero positions require computation. (b) The challenges encountered by MPUs with runahead technique on sparse DNN workloads: PE under-utilization (Section II-B) and runahead prefetch redundancy (Section II-C). (c) DARE’s solutions to the challenges: ISA extension to support non-strided access (Section III) and runahead execution with a runahead filter (Section IV).

on systolic arrays. Block-wise sparsity [16]–[18] can improve utilization but may introduce redundant computations. We observe that multiple sparse operations can be logically densified into a single dense operation to better exploit PE resources. However, limitations of current matrix ISAs make such densification infeasible, which further restricts PE utilization.

To address these inefficiencies, we propose DARE, an irregularity-tolerant MPU that integrates a Densifying ISA and filtered Runahead Execution. Our key contributions are as follows.

- We analyze the execution of sparse DNN layers on an MPU with NVR enabled and identify two issues (Fig. 2(b)): (i) PE under-utilization arising from ISA constraints and (ii) redundant NVR prefetches that ignore cache reuse.
- We propose a RISC-V matrix ISA that densifies multiple sparse operations into a single dense operation, thereby improving PE utilization (Fig. 2(c) upper).
- We design a lightweight runahead execution scheme for MPUs with an effective filter that improves area, performance, and energy efficiency (Fig. 2(c) lower).
- Experimental results show that DARE improves performance between  $1.04\times$  and  $4.44\times$  and energy efficiency between  $1.00\times$  and  $22.8\times$  over the baseline, while reducing hardware overhead by  $3.19\times$  compared to NVR.

## II. BACKGROUND AND MOTIVATION

### A. Matrix ISAs in CPUs

Recently, CPU vendors have integrated matrix ISA extensions, beyond traditional vector ISAs and processors [19]–[22]. We use Intel’s AMX [4] as a reference. It introduces eight 1 KB matrix registers, each with 16 rows and 64 bytes per row, and provides instructions for memory access and Matrix-Multiply-Accumulate (MMA), all executed at the matrix-register granularity. The memory access instructions require a uniform address stride across rows, which simplifies the ISA design.

### B. Irregularity in DNNs and ISA Constraints

Sparsity is widely adopted in DNNs to reduce computational cost [23], resulting in computation kernels with irregular memory access patterns and computation flows, exemplified by SpMMA and SDDMM [24], [25]. An example computation flow of SDDMM<sup>1</sup> is shown in Fig. 2(a), where computation is required only at the non-zero positions of matrix C. The load of matrix A involves two levels of indirection imposed by the Compressed Sparse Column (CSC) format<sup>2</sup>, which is difficult to predict. In summary, sparsity poses significant challenges to PE utilization in systolic arrays and to memory access efficiency.

**Current Matrix ISAs would benefit from non-strided memory access instructions to implement densification.** Block-wise sparsity [16]–[18] can improve utilization but may introduce redundant computation. We observe that multiple sparse MMA operations can be densified into a single MMA instruction to achieve higher PE utilization. For instance, the computation of the first column of matrix C in Fig. 2(a) can be densified as shown in the upper part of Fig. 2(c). However, unequal address strides between rows 0, 1, and 3 of matrix A prevent densification under current ISA stride constraints, resulting in a two-step execution shown in the upper part of Fig. 2(b). This motivates the need for ISA extensions that support non-strided memory access to improve PE utilization.

### C. Runahead Execution and Limitations

Runahead execution [10]–[14] is a prefetching technique for irregular memory accesses. It allows the processor to speculatively execute future instructions when stalled by long-latency memory accesses. The memory requests generated during this runahead phase provide accurate prefetches since they follow the actual control flow. NVR [15] extends this idea to DNN workloads by applying runahead on an NPU. However, porting such techniques to an MPU presents significant challenges.

<sup>1</sup>Computation flow of SpMMA can be found in [26].

<sup>2</sup>The compressed sparse row format also imposes indirection [27]. We here take CSC as an example.

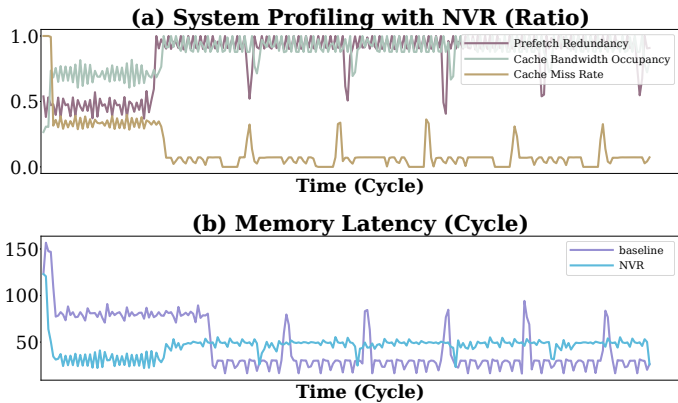


Fig. 3: (a) The cache miss rate, prefetch redundancy and the cache bandwidth occupancy in NVR on SDDMM. (b) The average memory access latency in baseline and NVR.

**Inefficiency from Prefetch Redundancy.** A redundant prefetch occurs when a prefetched block is already present in the cache. Fig. 3(a) shows that runahead produces excessive redundant prefetches when the cache miss rate is low. They contend for cache bandwidth like normal requests and can eventually saturate it. As shown in Fig. 3(b), this redundancy substantially increases memory latency and degrades performance. The root cause lies in data reuse within DNN workloads. The lower part of Fig. 2(b) illustrates an example: although only column 1 of matrix B misses in iteration 2, NVR prefetches both row 0 of matrix A and column 1 of matrix B. Row 0 of matrix A was already prefetched in iteration 1 and remained in the cache, making the prefetch redundant. Processing units with smaller local memories (registers or scratchpads) suffer more redundancy because they rely on caches for reuse. Thus, redundancy becomes more severe when moving from a large NPU with a 256 KB scratchpad such as Gemmini [28] to a smaller MPU with 8 KB registers such as AMX. Because data reuse is common in DNN workloads, redundancy broadly degrades both performance and energy efficiency.

**High Implementation Overhead.** Most prior runahead techniques rely on checkpointing because they speculatively pre-execute future instructions and require context restoration. Checkpoint register files can incur up to 8 KB overhead in an AMX-like design. Some approaches, such as NVR, avoid checkpointing but incur a higher overhead of 9.72 KB. Thus, existing runahead implementations remain too costly for MPUs.

The inefficiency in area, performance, and energy consumption of deploying runahead on MPUs calls for a more lightweight design with a runahead filter to fully exploit runahead execution.

### III. DARE INSTRUCTION SET ARCHITECTURE

In this section, we first introduce the basic DARE ISA. Then we extend it to support non-strided memory access.

#### A. Basic Instruction Set Architecture

The DARE ISA is a RISC-V matrix ISA inspired by Intel AMX. It provides eight 1 KB matrix registers ( $m0-m7$ ), each

organized as 16 rows by 64 bytes, and three Control and Status Registers (CSRs),  $matrixM$ ,  $matrixK$ , and  $matrixN$ , which define the logical shape of the matrix registers.

To support load, store, and MMA operations, DARE introduces three core instructions: `mld`, `mst`, and `mma`. `mld` loads a tile from memory into a matrix register, and `mst` stores a tile from a matrix register to memory. Each tile consists of  $matrixM$  rows and  $matrixK$  bytes per row. Both `mld` and `mst` take two source operands from general-purpose registers: one holds the base address of the first row and the other holds the stride between consecutive rows. `mma` performs an MMA across three matrix registers. The shape of two source registers is  $matrixM \times matrixK$  and  $matrixN \times matrixK$ , respectively. In addition, a configuration instruction, `mcfg`, is provided to configure the CSRs.

#### B. Supporting Non-Strided Memory Access

We extend the basic ISA to support non-strided memory access by treating the first element of each matrix register row as a base address vector. We refer to this extension as Gather Scatter Access (GSA) and define the corresponding non-strided load and store instructions as `mgather` and `mscatter`, respectively. `mgather` and `mscatter` use the elements of this base address vector as per-row base addresses for memory load and store operations. The base address vector itself can be loaded from memory using `mld`. The address generation overhead can be alleviated by decoupling address generation from MPU execution into two threads, while the hardware overhead requires only minor modifications to the decoder's handling of source operands compared to `mld` and `mst`. The complete DARE ISA is summarized in Table I.

TABLE I: DARE Instructions List

Assembly Format	Description
<code>mcfg, rs1, rs2</code>	Write the value in <code>rs2</code> to the CSR indexed by <code>rs1</code>
<code>mld, md, (rs1), rs2</code>	Load a tile from address <code>rs1</code> with <code>rs2</code> stride to <code>md</code>
<code>mst, ms3, (rs1), rs2</code>	Store a tile to address <code>rs1</code> with <code>rs2</code> stride from <code>ms3</code>
<code>mma, md, ms1, ms2</code>	Multiply <code>ms1</code> and <code>ms2</code> and accumulate to <code>md</code>
<code>mgather, md, (ms1)</code>	Load a tile addressed by <code>ms1</code> to <code>md</code>
<code>mscatter, ms2, (ms1)</code>	Store a tile addressed by <code>ms1</code> from <code>ms2</code>

## IV. DARE MICROARCHITECTURE

#### A. DARE Overview

An overview of the DARE microarchitecture is shown in Fig. 4(a). Instructions are dispatched non-speculatively from the host CPU to the MPU. DARE is an out-of-order superscalar processor. It connects directly to the Last Level Cache (LLC) through a Load Store Unit (LSU), which includes a Load Queue (LQ) and a Store Queue (SQ). All instructions are decomposed into micro-operations (uops) to simplify hardware implementation. Memory access instructions are further decomposed at the granularity of matrix register rows.

We introduce a circular queue named the **Runahead Issue Queue (RIQ)** that holds stalled instructions and serves as a candidate pool for prefetch uops. To enable runahead execution

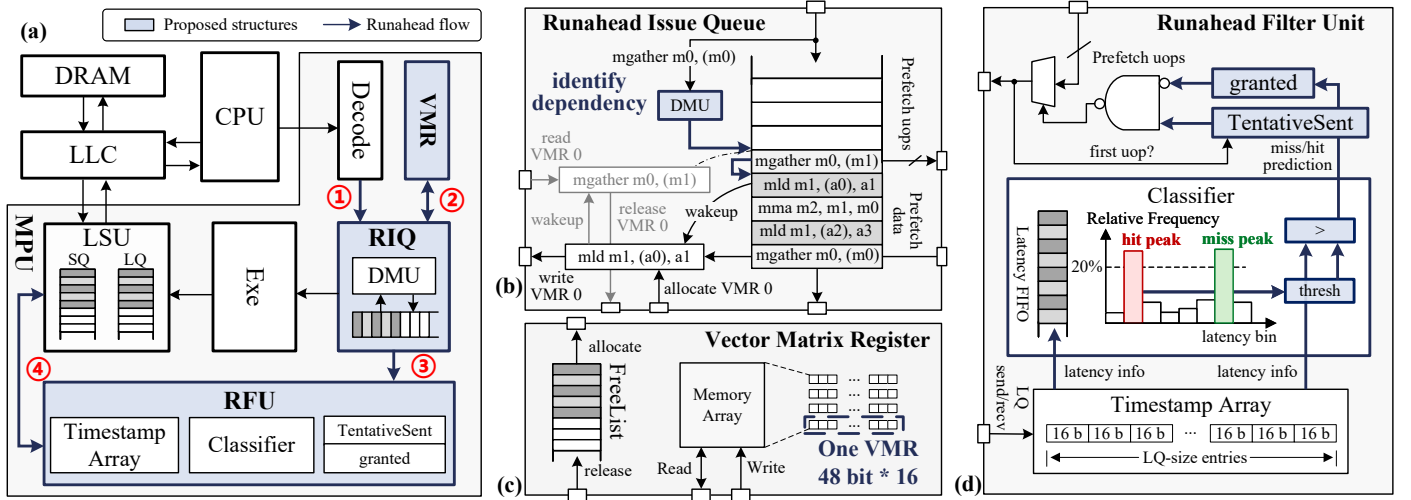


Fig. 4: (a) Overview of the DARE architecture. Blue blocks indicate components proposed by DARE. (b) Runahead Issue Queue (RIQ) with a sub-module named the Dependency Management Unit (DMU). (c) Vector Matrix Register (VMR) as an auxiliary register file to store base vector addresses. (d) Runahead Filter Unit (RFU) with a threshold-based classifier.

for `mgather`, we add a reduced matrix register file called the **Vector Matrix Register (VMR)**, which provides temporary storage for base address vectors. To further mitigate performance and energy inefficiencies caused by prefetch redundancy, we extend DARE with a **Runahead Filter Unit (RFU)**.

## B. Overall Workflow

The CPU dispatches DARE instructions non-speculatively to the MPU, where each instruction is first decoded and then inserted into the RIQ ①. Each memory access instruction in the RIQ is decomposed into prefetch uops, which are then arbitrated and filtered by the RFU ③. Only uops that pass arbitration are issued to the LSU ④. The head instruction of the RIQ is issued when it has no Read-After-Write (RAW), Write-After-Write (WAW), or Write-After-Read (WAR) conflicts with older instructions in the MPU. For `mgather` instructions in the RIQ, the RIQ wakes up the producer instruction that generates the base address vector. Each instruction in the dependency chain allocates a VMR entry ② to hold its loaded data.

## C. Runahead Issue Queue

The RIQ is a 32-entry circular queue, as determined in Section V-F, that accommodates stalled instructions together with a Dependency Management Unit (DMU), as shown in Fig. 4(b). Each RIQ entry stores the full instruction information and a decompose counter. Memory instructions in the RIQ are decomposed into micro-operations (uops), which are sent to the RFU for arbitration.

When processing an `mgather` instruction, the DMU traverses the RIQ backward to identify the corresponding dependency chain, which terminates at an `mld`. It then wakes and executes the oldest instruction in the chain. Each completed instruction in turn wakes its consumer. Every woken instruction allocates a VMR entry and treats it as its destination register. A VMR entry is released once its consumer finishes reading it.

## D. Vector Matrix Register

The Vector Matrix Register (VMR) is a reduced matrix register file designed to enable accurate prefetching for `mgather` as it introduces matrix registers into the address dependency chain. Because the DARE architecture lacks additional physical registers, an auxiliary file is required to temporarily store base address vectors. Since only the first 48 bits of each row are needed under the Sv48 virtualization format with 48-bit virtual addresses, the VMR reduces the matrix register capacity per row to 48 bits, as shown in Fig. 4(c). Each VMR entry is a 16-element vector corresponding to the 16 rows of a matrix register, with each element holding 48 bits. The VMR contains 16 entries in total, as determined in Section V-F. These entries are dynamically managed by a free list implemented as a circular queue that tracks the currently available VMR entries.

## E. Runahead Filter Unit

The RFU, illustrated in Fig. 4(d), filters redundant prefetch uops. It implements a **Tentative Uop Mechanism**: for each memory access instruction in the RIQ, only the first uop is issued initially, and subsequent uops are issued only if this tentative uop misses in the LLC or the instruction requires to write a VMR entry. Each RIQ entry is augmented with two fields: *granted* and *TentativeSent*. Uops are suppressed if the condition  $!granted \ \&\& \ TentativeSent$  holds. The *granted* flag is set when the tentative uop is classified as a LLC miss or when the instruction allocates a VMR entry, while the *TentativeSent* flag is set when a corresponding uop is issued. The central challenge is **how to determine the miss/hit status of a uop, given that DARE cannot directly probe LLC status and memory conditions may vary at runtime**.

DARE addresses this challenge using a threshold-based, unsupervised binary classifier with uop latency as its only input. The key observation is that the memory access latency distribution typically exhibits a **bimodal shape**: one peak corresponds to LLC hits (red in Fig. 4(d)) and the other to

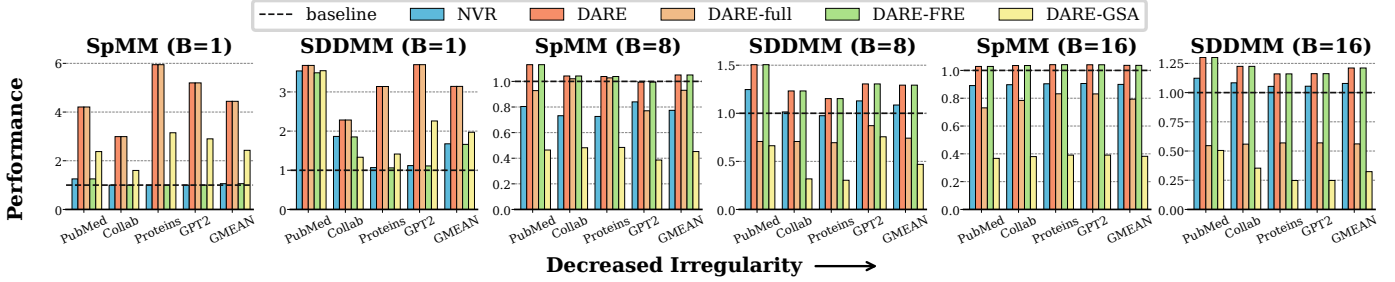


Fig. 5: Performance normalized to baseline. DARE achieves 1.04 $\times$  to 4.44 $\times$  performance improvement on average compared to the baseline across various benchmarks. DARE is reported as the better between DARE-FRE and DARE-full.

TABLE II: System configuration

Name	Detailed Configuration
Frequency	2.0 GHz
Host CPU	RV64GC + proposed DARE ISA
MPU	48-entry LQ/SQ, 16 $\times$ 16 systolic array with 32-bit-datapath PEs, 2-way-issue out-of-order
LLC	2MB, 16-way set associative, 16 banks, 1 read/1 write port, 20-cycle hit latency
Main Memory	45 ns latency, 50 GiB/s bandwidth

LLC misses (green in Fig. 4(d)). The classifier dynamically updates a threshold such that any uop whose latency exceeds the threshold is classified as a cache miss. The threshold is updated in three steps:

- 1) The classifier builds a histogram of recently observed latencies (32 in this paper) using bins of size 8 cycles.
- 2) A bin whose relative frequency exceeds 20% is identified as a peak. Only the smallest and largest peaks are retained.
- 3) When the distance between the two peaks exceeds a margin (4 bins in this paper), the threshold is set to the latency of the minimum bin between them plus a fixed slack of 32 cycles. The slack prevents misclassifying a miss uop as a hit when hit latency fluctuates slightly.

This dynamic classifier enables the RFU to adapt to runtime conditions, effectively reducing prefetch redundancy.

## V. EVALUATION

### A. Experimental Setup

1) *Methodology*: We implement DARE in Register Transfer Level (RTL) and construct a cycle-accurate DARE model in gem5 [29] to collect performance data. We synthesize DARE with Synopsys Design Compiler in a TSMC 28 nm process at 2 GHz to obtain power and area results. Cache timing and energy data are obtained from CACTI 7 [30]. The detailed system configuration is listed in Table II. We emulate NVR by equipping the MPU with infinite RIQ and VMR capacity, thereby preserving NVR’s distant-prefetch capability, since baseline MPUs lack a specialized sparse unit. The baseline MPU excludes the RIQ, RFU, and VMR.

For the ablation study, we define the following variants:

- DARE-FRE: Only FRE is enabled.
- DARE-GSA: Only GSA is enabled.

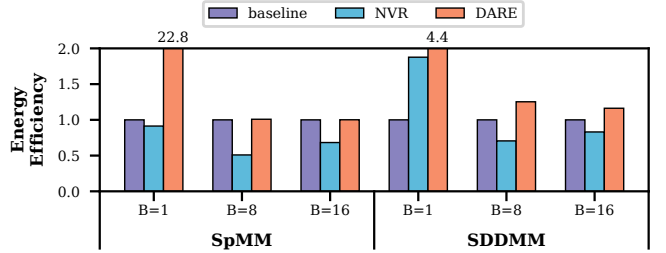


Fig. 6: Energy efficiency normalized to the baseline.

- DARE-full: Both GSA and FRE are enabled.
- DARE: Better in DARE-FRE or DARE-full, as GSA can be disabled via an offline profiling.

2) *Benchmarks and Datasets*: We select two kernel-level benchmarks: SpMM and SDDMM. We choose graphs from PubMed [31], OGBL-collab, and OGBN-proteins [32], and the attention map of GPT-2 [33] on Wikitext2 [34] pruned to 90% sparsity as the dataset. For the selected graphs, we take a subgraph from each to reduce simulation time. We further blockify the original datasets, with the notation  $B = N$  indicating the block shape used to blockify is  $N \times N$ .

### B. Hardware Overhead

The overall storage overhead is 3.05 KB, representing a 3.19 $\times$  reduction compared to NVR. The total area overhead is 9.2% relative to a baseline MPU, with 3.8%, 4.1%, and 1.3% attributed to the VMR, RIQ, and RFU, respectively.

### C. Performance

1) *Overall Performance*: Performance results are shown in Fig. 5. DARE achieves a geometric mean performance improvement of 1.04 $\times$  to 4.44 $\times$  across benchmarks compared to the baseline. DARE consistently outperforms both NVR and the baseline across all benchmarks. For specific benchmarks (e.g., SpMM with  $B = 8$ ), NVR degrades performance (0.77 $\times$ ) while DARE yields an improvement (1.05 $\times$ ), underscoring DARE’s effectiveness.

2) *Ablation Study*: The ablation study results in Fig. 5 indicate that DARE-full performs best with highly irregular workloads, while DARE-FRE is more effective with lower irregularity. In highly irregular benchmarks ( $B = 1$ ), both DARE-FRE and DARE-GSA contribute to performance enhancement, and DARE-GSA outperforms DARE-FRE in such scenarios. Moreover, the performance enhancement of DARE-full (4.44 $\times$ ) exceeds the product of DARE-FRE (1.06 $\times$ ) and

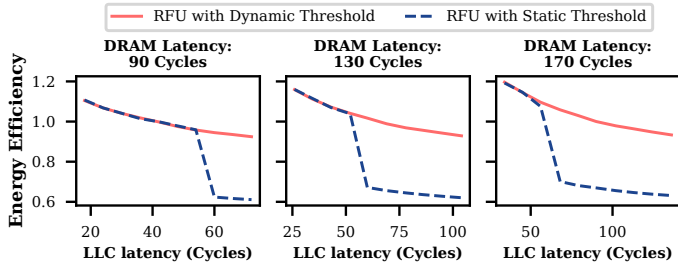


Fig. 7: The robustness in energy efficiency of DARE across memory environments.

DARE-GSA ( $2.43\times$ ) in SpMM, indicating a synergistic effect between them. When irregularity decreases to  $B \geq 8$ , the additional loading of base address vectors by DARE-GSA results in performance degradation, and DARE-FRE emerges as the dominant optimization. The ablation study results suggest that GSA should be selectively disabled based on workload characteristics, as will be discussed in Section V-G.

#### D. Energy Efficiency

Fig. 6 depicts the energy efficiency results, showing that DARE achieves an average energy efficiency ranging from  $1.00\times$  to  $22.8\times$  compared to the baseline. In specific workloads such as SDDMM with  $B = 8$ , NVR improves performance ( $1.09\times$ ) at the cost of lower energy efficiency ( $0.71\times$ ) due to its significant redundant prefetches. In contrast, DARE enhances both performance ( $1.29\times$ ) and energy efficiency ( $1.25\times$ ) by reducing this redundancy. Furthermore, DARE significantly enhances energy efficiency in SDDMM ( $4.37\times$ ) and SpMM ( $22.8\times$ ) with  $B = 1$  by reducing runtime and improving data reuse within the MPU. In benchmarks with low irregularity (SpMM with  $B \geq 8$ ), DARE yields modest energy efficiency benefits due to the low cache miss rates in such scenarios.

#### E. Robustness across Memory Environments

Fig. 7 illustrates the energy efficiency under various memory environments for SDDMM with  $B = 8$ , normalized to the baseline. We implemented a baseline RFU with a static threshold of 64 cycles. In general, energy efficiency decreases with increased LLC latency, as the performance gain from FRE diminishes. However, the static-threshold RFU suffers an abrupt energy efficiency drop when LLC latency exceeds a value near its threshold, as it cannot distinguish between LLC hits and misses when LLC latency surpasses the static threshold, causing it to grant every instruction. In contrast, the dynamic-threshold RFU proposed in DARE demonstrates high robustness across various memory environments.

#### F. Sensitivity to VMR Size and RIQ Size

Fig. 8 illustrates the sensitivity of DARE’s performance to VMR and RIQ size, with performance normalized to the range  $[0,1]$  based on the maximum and minimum values for each case. The results suggest that the resource requirements for RIQ and VMR vary with the block size. When  $B = 1$ , increasing the VMR size improves performance, while a larger RIQ may lead to performance degradation. This is because a larger RIQ increases the number of `ml_d` instructions targeting base address

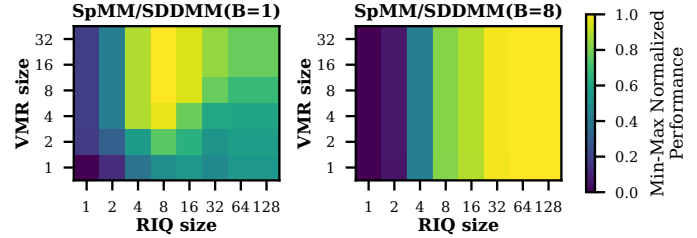


Fig. 8: The sensitivity of performance to VMR size and RIQ size. Performance is normalized to  $[0,1]$  with the maximum and minimum.

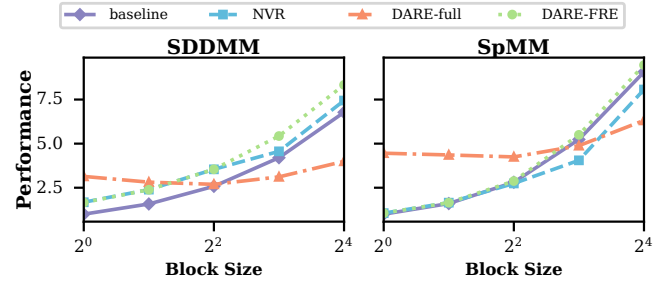


Fig. 9: The sensitivity of performance to the block size. All results are normalized to baseline where  $B = 1$ .

vectors, which are forced to be granted by the RFU. When  $B = 8$ , performance is monotonic with respect to RIQ size, while VMR size has a negligible effect. This is because DARE-FRE dominates in this scenario, with no requirement for the VMR but a necessity for a large RIQ to hide the prediction latency of the RFU. To balance these two scenarios, we chose a 32-entry RIQ and a 16-entry VMR for our design.

#### G. Sensitivity to Block Size

Fig. 9 shows the performance trends under different block sizes. The performance of DARE-full relative to DARE-FRE is monotonic with respect to block size, showing benefits at small  $B$  and performance degradation at large  $B$ . Therefore, an offline profiling step can be used to decide when to disable GSA according to the block size: disabled at  $B \geq 4$  for SDDMM and at  $B \geq 8$  for SpMM. The results also show that using block-wise sparsity can effectively improve performance across workloads, as larger block sizes fit the systolic array better. DARE-FRE benefits from larger block sizes ( $B \geq 8$ ) to a greater extent than both the baseline and NVR.

## VI. CONCLUSION

This paper presents DARE, an irregularity-tolerant MPU that integrates a densifying ISA (GSA) and a filtered run-head mechanism (FRE) to improve PE utilization and reduce prefetch redundancy. Experimental results demonstrate that GSA and FRE act synergistically for highly irregular workloads ( $B = 1$ ), while FRE dominates when irregularity decreases ( $B \geq 8$ ). Furthermore, DARE exhibits high robustness across various memory environments. Across SpMM and SDDMM benchmarks, DARE achieves up to a  $4.44\times$  performance improvement and a  $22.8\times$  energy efficiency enhancement over the baseline, with a  $3.19\times$  hardware overhead reduction compared to NVR. These results render DARE a practical solution for handling irregularity in DNN workloads.

## REFERENCES

- [1] G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, "Spade: A flexible and scalable accelerator for spmm and sddmm," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, Conference Proceedings, p. 1–15.
- [2] H. Kim, G. Ye, N. Wang, A. Yazdanbakhsh, and N. S. Kim, "Exploiting Intel Advanced Matrix Extensions (AMX) for Large Language Model Inference," *IEEE Computer Architecture Letters*, vol. 23, no. 1, p. 117–120, 2024.
- [3] H. Kim, N. Wang, Q. Xia, J. Huang, A. Yazdanbakhsh, and N. S. Kim, "LIA: A Single-GPU LLM Inference Acceleration with Cooperative AMX-Enabled CPU-GPU Computation and CXL Offloading," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, Conference Proceedings, p. 544–558.
- [4] Intel, "Intel® Architecture Instruction Set Extensions Programming Reference," 2022. [Online]. Available: <https://www.intel.com/content/www/us/en/content-details/671368/intel-architecture-instruction-set-extensions-programming-reference.html>
- [5] Arm, "Arm Scalable Matrix Extension (SME) Introduction," 2024. [Online]. Available: <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-scalable-matrix-extension-introduction>
- [6] G. Jeong, E. Qin, A. Samajdar, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna, "Rasa: Efficient register-aware systolic array matrix engine for cpu," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, Conference Proceedings, p. 253–258.
- [7] G. Jeong, S. Damani, A. R. Bambhaniya, E. Qin, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna, "VEGETA: Vertically-Integrated Extensions for Sparse/Dense GEMM Tile Acceleration on CPUs," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Conference Proceedings, p. 259–272.
- [8] T. Ta, J. Randall, and C. Batten, "SparseZipper: Enhancing Matrix Extensions to Accelerate SpGEMM on CPUs," *arXiv preprint arXiv:2502.11353*, 2025.
- [9] X. Yang, X. Kong, K. Wang, X. Fan, Z. Zhou, Z. Yang, Z. Wang, and J. Han, "VLSUMaP: A High-Performance Matrix Processor with Virtually Expanded LSU Boosting HBM Bandwidth Utilization," in *Proceedings of the Great Lakes Symposium on VLSI 2025*, 2025, pp. 450–456.
- [10] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, Conference Proceedings, p. 129–140.
- [11] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Conference Proceedings, p. 397–410.
- [12] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector Runahead," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Conference Proceedings.
- [13] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled Vector Runahead," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Conference Proceedings, p. 17–31.
- [14] J. Roelandts, A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Scalar Vector Runahead," 2024.
- [15] H. Wang, Z. Zhao, J. Wang, Y. Du, Y. Cheng, B. Guo, H. Xiao, C. Ma, X. Han, and D. You, "NVR: Vector Runahead on NPUs for Sparse Memory Access," *arXiv preprint arXiv:2502.13873*, 2025.
- [16] S. Narang, E. Undersander, and G. Diamos, "Block-sparse recurrent neural networks," *arXiv preprint arXiv:1711.02782*, 2017.
- [17] J. Qiu, H. Ma, O. Levy, S. W.-t. Yih, S. Wang, and J. Tang, "Block-wise self-attention for long document understanding," *arXiv preprint arXiv:1911.02972*, 2019.
- [18] J. Yuan, H. Gao, D. Dai, J. Luo, L. Zhao, Z. Zhang, Z. Xie, Y. Wei, L. Wang, Z. Xiao *et al.*, "Native sparse attention: Hardware-aligned and natively trainable sparse attention," *arXiv preprint arXiv:2502.11089*, 2025.
- [19] J. Dong, X. Chen, and M. Gao, "A Unified Vector Processing Unit for Fully Homomorphic Encryption," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–7.
- [20] N. K. Purayil, M. Perotti, T. Fischer, and L. Benini, "AraXL: A Physically Scalable, Ultra-Wide RISC-V Vector Processor Design for Fast and Efficient Computation on Long Vectors," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–7.
- [21] X. Yu, Y. Sun, Y. Zhao, H. Kuang, and J. Han, "RVCE-FAL: A RISC-V Scalar-Vector Custom Extension for Faster FALCON Digital Signature," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.
- [22] V. Titopoulos, K. Alexandridis, C. Peltekis, C. Nicopoulos, and G. Dimitrakopoulos, "IndexMAC: A Custom RISC-V Vector Instruction to Accelerate Structured-Sparse Matrix Multiplications," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.
- [23] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, p. 370–403, 2021.
- [24] M. K. Rahman, M. H. Sujon, and A. Azad, "Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 256–266.
- [25] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, "Sampled dense matrix multiplication for high-performance machine learning," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 32–41.
- [26] G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, "Spade: A flexible and scalable accelerator for spmm and sddmm," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, Conference Proceedings, p. 1–15.
- [27] C. Zhang, P. Scheffler, T. Benz, M. Perotti, and L. Benini, "Near-Memory Parallel Indexing and Coalescing: Enabling Highly Efficient Indirect Access for SpMV," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.
- [28] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, and H. Mao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, Conference Proceedings, p. 769–774.
- [29] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, p. 1–7, 2011.
- [30] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, p. 1–25, 2017, oA status: bronze. [Online]. Available: [http://dl.acm.org/ft\\_gateway.cfm?id=3085572&type=pdf](http://dl.acm.org/ft_gateway.cfm?id=3085572&type=pdf)
- [31] J. White, "PubMed 2.0," *Medical reference services quarterly*, vol. 39, no. 4, p. 382–387, 2020.
- [32] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, p. 22118–22133, 2020.
- [33] B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and S. Agarwal, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, vol. 1, 2020.
- [34] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.