HStencil: Matrix-Vector Stencil Computation with Interleaved Outer Product and MLA

Han Huang
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
huangh367@mail2.sysu.edu.cn

Jiabin Xie
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
xiejb6@mail2.sysu.edu.cn

Guangnan Feng
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
fenggn7@mail.sysu.edu.cn

Xianwei Zhang
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
zhangxw79@mail.sysu.edu.cn

Dan Huang
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
huangd79@mail.sysu.edu.cn

Zhiguang Chen
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
chenzhg29@mail.sysu.edu.cn

Yutong Lu*
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, Guangdong, China
luyutong@mail.sysu.edu.cn

Abstract

Stencil computations are fundamental to various HPC and intelligent computing applications, often consuming significant execution time. The emergence of specialized matrix units presents new opportunities to accelerate stencil computations. While scalable matrix compute units provide substantial computing horsepower, prior efforts fail to fully utilize the computing capabilities for stencils due to suboptimal matrix-unit utilization, limited instruction-level parallelism, and low cache hit rates. This paper introduces *HStencil*, a novel stencil computing framework utilizing matrix and vector units. *HStencil* addresses these challenges through three contributions: 1) microkernels that jointly leverage matrix and vector units to enhance hardware utilization; 2) fine-grained instruction scheduling with interleaved execution to enhance instruction-level parallelism; and 3) spatial prefetch to sustain high performance when working sets exceed cache capacity. Evaluations on representative benchmarks demonstrate that *HStencil* achieves maximum speedups of 1.81x – 5.76x over auto-vectorization across different CPU platforms, delivers 31% - 91% higher performance versus state-of-the-art methods.

*Corresponding author.

CCS Concepts

• **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization → Parallel architectures**.

Keywords

Stencil Computation, Matrix and Vector Compute Units, Outer Product

ACM Reference Format:
Han Huang, Jiabin Xie, Guangnan Feng, Xianwei Zhang, Dan Huang, Zhiguang Chen, and Yutong Lu. 2025. HStencil: Matrix-Vector Stencil Computation with Interleaved Outer Product and MLA. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3712285.3759769

1 Introduction

Stencil computations are prevalent patterns in high-performance computing (HPC) and intelligent computing applications [6, 33]. Stencil operations can be characterized as iteratively updating each point in a multi-dimensional grid according to the neighboring values. As shown in Figure 1, stencil computations are broadly categorized into two spatial patterns: *star* and *box*. The *star* pattern updates points along coordinate axes, while the *box* pattern incorporates off-axis neighbors.

To accelerate stencil executions, prior researches have extensively explored leveraging the vector units in commercial processors. For example, the DLT[16] method formats the vector layout of the stencils to reduce redundant load accesses. Meanwhile, temporal vectorization[34] fuses stencil computations across iterations,

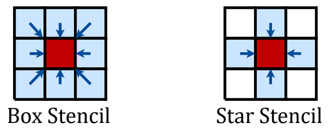
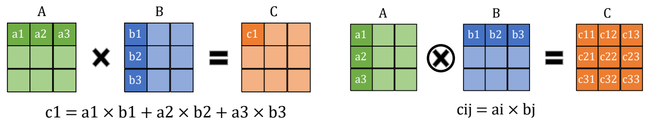


Figure 1: Box and Star Stencils.

enabling more effective vectorization. Motivated by the success of vector units, architecture designers proposed the more ambitious matrix units, which are to meet the rising demands of matrix operations in scientific computing and machine learning applications.

Modern architectures have introduced specialized matrix units, such as NVIDIA Tensor Cores[10], Google TPUs[14], scalable matrix compute units[32] and Intel AMXs[18]. These units put significant efforts into accelerating matrix operations through tailored matrix multiplication (MM) instructions. For instance, NVIDIA Tensor Cores feature inner product-based (Figure 2a) designs, whereas the emerging scalable matrix compute units[32] relies on the alternative outer product acceleration (Figure 2b). Compared to inner product, outer product operations can be decomposed into more memory-friendly operations on scalable matrix compute units. In this regard, scalable matrix compute units are highly promising to reduce memory overhead and provide more sufficient computing capabilities.



(a) Inner product. **(b) Outer product.**

Figure 2: Outer product and inner product operations.

Unlike dense matrix multiplications, stencil computations require a sophisticated design to exploit the matrix-unit-utilization. A plethora of works have been proposed to implement and optimize stencil computations on NVIDIA Tensor Cores (e.g., TCStencil[24], ConvStencil[9] and LoRAStencil[43]). For scalable matrix compute units, existing works (e.g. STOP [45]) fail to fully utilize the sufficient computing capabilities provided by the outer product operations. These works exhibit suboptimal utilization of matrix units in certain stencil patterns, and face performance degradation when processing out-of-cache stencil workloads. Moreover, the existing solutions suffer from performance bottlenecks on load/store instructions, and under-utilizes the instruction-level parallelism (ILP) for computation-memory access overlap. These gaps underscore the need for a holistic framework to unlock the full potential of outer product-based architectures for stencil workloads.

To overcome the challenges, this paper presents ***HStencil***, a hybrid and heterogeneous stencil computing framework designed to maximize performance on modern CPU architectures with scalable vector and matrix compute units. *HStencil* consists of three key components: 1) Enhanced micro kernels of stencil computation that leverage scalable vector and matrix compute units to ensure matrix unit utilization and continuous memory accesses. 2) Fine-grained instruction scheduling to exploit ILP by interleaving load, compute, and store operations across matrix and vector units. 3) Spatial prefetch strategy to efficiently manage out-of-cache stencils, improving cache hit rates and reducing off-chip memory traffic.

In summary, this paper makes the following contributions:

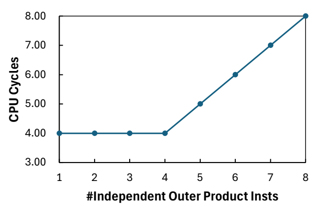
- We present *HStencil*, a novel stencil framework to efficiently leverage scalable matrix compute units. It combines scalable vector and matrix compute units to maximize computational throughput and memory efficiency on next-generation CPU architectures.
- We further incorporate enhanced micro kernels with in-place accumulation methods, instruction scheduling to exploit ILP across matrix and vector pipelines, spatial prefetch to minimize overhead for out-of-cache stencils.
- Experimental results on representative benchmarks demonstrate that *HStencil* significantly improves overall execution performance, far outperforming the state-of-the-art methods. Moreover, *HStencil* can be generalized to other heterogeneous architectures, and is the first efficient stencil implementation tailored to Apple M4 CPUs.

2 Background and Motivation

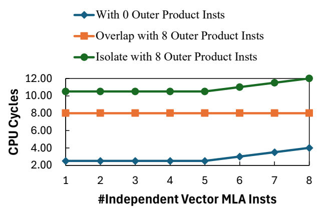
2.1 LX2 CPU Micro Architecture

The LX2 CPU is a next-generation high-performance CPU equipped with scalable vector and matrix compute units. The scalable matrix compute units feature outer product instructions that deliver substantial computing capability. Compared to the scalable vector Multiply-Accumulate (MLA) instruction, the outer product instruction reaches approximately **four times** the theoretical double-precision (FP64) performance of MLA. While this suggests that leveraging the outer product instructions can yield significant speedups compared to MLA instructions, the computing horsepower of vectors must not be overlooked. MLA instructions may outperform the outer product instructions, especially in cases where the utilization of the matrix unit is lower than 1/4.

Moreover, the matrix and vector instructions are dispatched to distinct pipelines, thereby enabling concurrent execution. To quantify this potential, we perform an ILP test with results being shown in Figure 3. The results confirm that matrix and vector instructions can be co-issued and executed concurrently. Moreover, overlapped execution can achieve up to 1.5x speedup over isolated execution, as shown in Figure 3b.



(a) Matrix Instruction ILP Test



(b) Vector Instruction ILP Test

Figure 3: Tests on Matrix and Vector ILP.

Matrix instructions operate on specialized two-dimensional matrix registers[5]. There are eight matrix registers for double-precision computation and can be assessed as **tiles**. Each tile can store up to 64 double-precision numbers, organized into 8 rows of 8 numbers, with each row known as a **slice**. The outer product instruction computes the outer product of two input vectors and accumulates the result in the matrix register. Data transfers between matrix and vector registers, as well as between matrix registers and memory, can be performed in both horizontal and vertical orientations.

Algorithm 1: Star-2D5P Stencil

```
  Input: matrix $A$, coefficient $c_1, c_2, c_3, c'_1, c'_3$
  Output: matrix $B$
1 for $point[i][j] \in A$ do
2     $B[i][j] = c_1 \times A[i][j-1] + c'_1 \times A[i-1][j] + c_2 \times A[i][j] + c'_3 \times A[i+1][j] + c_3 \times A[i][j+1]$
```

2.2 Outer Product-based Stencil Computation

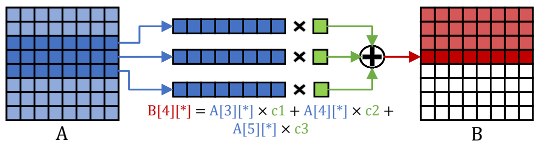
Stencil computations mostly involve summing the weighted values of spatially neighboring grid points. Algorithm 1 shows the computation of a star stencil. Additionally, the radius of the stencil is often characterized by $r = n$, where n is the distance from the center grid pointing to the outermost points. For simplicity, we use $r = 1$ 2D stencils to illustrate the computations.

Stencil computations are broadly categorized into two paradigms based on their data access patterns: gather and scatter [30]. In the gather approach, illustrated in (1) for a $r = 1$ stencil, elements from the input matrix A are actively aggregated to compute each output element in matrix B.

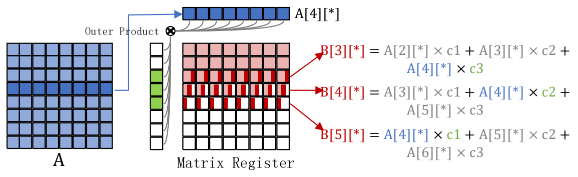
$$B_{(i,j)} \xleftarrow{\text{save}} c_1 * A_{(i-1,j)} + c_2 * A_{(i,j)} + c_3 * A_{(i+1,j)} \tag{1}$$

Conversely, the scatter approach, demonstrated in (2), operates passively: instead of accumulating contributions from A, each element in A contributes one-third to the final value of the target element of the matrix B per iteration.

$$\begin{pmatrix} B_{(i-1,j)} \\ B_{(i,j)} \\ B_{(i+1,j)} \end{pmatrix} \xleftarrow{\text{add}} \begin{pmatrix} c_3 \\ c_2 \\ c_1 \end{pmatrix} \otimes A_{(i,j)} \tag{2}$$



(a) Vector MLA stencil. (extended from gather form)


A[4][*]
Outer Product
A
Matrix Register
B[3][*] = A[2][*] × c1 + A[3][*] × c2 + A[4][*] × c3
B[4][*] = A[3][*] × c1 + A[4][*] × c2 + A[5][*] × c3
B[5][*] = A[4][*] × c1 + A[5][*] × c2 + A[6][*] × c3

(b) Outer product stencil. (extended from scatter form)

Figure 4: SIMD implementations of stencil computation.

To exploit SIMD (Single Instruction, Multiple Data) parallelism, the gather and scatter forms can be implemented by MLA instructions and outer product instructions, respectively. As shown in Figure 4a, the gather form leverages MLA instructions to vectorize scalar source elements. The scatter form employs outer product instructions (Figure 4b) to broadcast a source vector across a coefficient vector, thereby updating multiple columns of B in parallel.

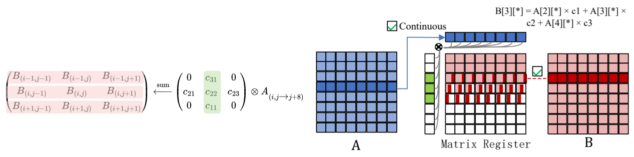
STOP[45] is the first and state-of-the-art work to leverage the outer product operations to compute stencils. Based on the outer product operations, the box stencils can be calculated in (3). The matrix C is the coefficient matrix, and the matrix B can be obtained by accumulating the outer product results.

$$\begin{pmatrix} B_{(i-1,j-1)} & B_{(i-1,j)} & B_{(i-1,j+1)} \\ B_{(i,j-1)} & B_{(i,j)} & B_{(i,j+1)} \\ B_{(i+1,j-1)} & B_{(i+1,j)} & B_{(i+1,j+1)} \end{pmatrix} \xleftarrow{\text{add}} \begin{pmatrix} c_{31} & c_{32} & c_{33} \\ c_{21} & c_{22} & c_{23} \\ c_{11} & c_{12} & c_{13} \end{pmatrix} \otimes A_{(i,j)} \tag{3}$$

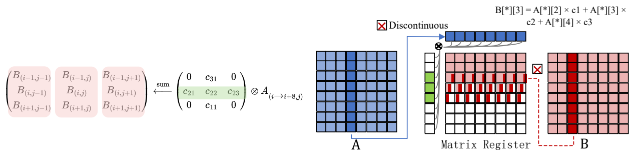
For star stencils, similar to the box ones, they can also be represented in (4). As star stencils need fewer coefficients, their coefficient matrices can be viewed as sparse forms of those of box stencils.

$$\begin{pmatrix} B_{(i-1,j-1)} & B_{(i-1,j)} & B_{(i-1,j+1)} \\ B_{(i,j-1)} & B_{(i,j)} & B_{(i,j+1)} \\ B_{(i+1,j-1)} & B_{(i+1,j)} & B_{(i+1,j+1)} \end{pmatrix} \xleftarrow{\text{add}} \begin{pmatrix} 0 & c_{31} & 0 \\ c_{21} & c_{22} & c_{23} \\ 0 & c_{11} & 0 \end{pmatrix} \otimes A_{(i,j)} \tag{4}$$

The above equations can be computed along the outer and inner axes. As shown in Figure 5, the outer-axis outer products take rows from the matrix A, and update rows of the matrix B. On the contrary, the inner-axis outer products take columns from the matrix A, and update columns of the matrix B. Therefore, outer-axis ways are more efficient than inner-axis ways for continuous memory access.


B[3][*] = A[2][*] × c1 + A[3][*] × c2 + A[4][*] × c3
Continuous
$\begin{pmatrix} B_{(i-1,j-1)} & B_{(i-1,j)} & B_{(i-1,j+1)} \\ B_{(i,j-1)} & B_{(i,j)} & B_{(i,j+1)} \\ B_{(i+1,j-1)} & B_{(i+1,j)} & B_{(i+1,j+1)} \end{pmatrix} \xleftarrow{\text{sum}} \begin{pmatrix} 0 & c_{31} & 0 \\ c_{21} & c_{22} & c_{23} \\ 0 & c_{11} & 0 \end{pmatrix} \otimes A_{(i,j\to j+8)}$
A
Matrix Register
B

(a) Outer-axis Outer Product


B[*][3] = A[*][2] × c1 + A[*][3] × c2 + A[*][4] × c3
Discontinuous
$\begin{pmatrix} B_{(i-1,j-1)} & B_{(i-1,j)} & B_{(i-1,j+1)} \\ B_{(i,j-1)} & B_{(i,j)} & B_{(i,j+1)} \\ B_{(i+1,j-1)} & B_{(i+1,j)} & B_{(i+1,j+1)} \end{pmatrix} \xleftarrow{\text{sum}} \begin{pmatrix} 0 & c_{31} & 0 \\ c_{21} & c_{22} & c_{23} \\ 0 & c_{11} & 0 \end{pmatrix} \otimes A_{(i\to i+8,j)}$
A
Matrix Register
B

(b) Inner-axis Outer Product

Figure 5: Outer products on the outer and inner axes.

2.3 Performance Bottlenecks

Although outperforming traditional compiler-based vectorization and temporal vectorization techniques, outer-product-based stencil computation (i.e. STOP) introduces unique challenges. It shifts the computational paradigm from vector-wise to matrix-wise processing, thereby bringing performance bottlenecks in matrix-unit utilization, instruction-level parallelism and cache hit rates.

Table 1: Matrix-unit utilization of different methods. (single-register)

Method	Matrix-unit Utilization
Outer-axis (Box)	41.7%
Outer-axis (Star)	18.3%
Outer&inner-axis (Star)	41.7%

2.3.1 *Low Matrix-unit Utilization.*

The utilization of scalable matrix compute units relies on two aspects: 1) utilization in a single matrix register and 2) utilization across multiple matrix registers.

The utilization in a single matrix register is demonstrated in Table 1. Box stencils achieve moderate matrix-unit utilization through outer-axis methods, using over 40% of the matrix units. However, star stencils present a unique optimization challenge due to sparsity in their coefficient matrices, which leads to suboptimal utilization of matrix units with outer-axis methods. As shown in Table 1, outer-axis methods achieve less than 20% matrix-unit utilization for star stencils, which is extremely low compared to box. While hybrid outer-inner outer products can preserve the utilization, they introduce performance bottlenecks through non-contiguous memory access patterns. This trade-off highlights the need for a redesigned computation strategy that simultaneously optimizes matrix-unit utilization and memory access for star stencils.

Moreover, single-register kernels fail to maximize scalable matrix compute unit throughput, as peak performance is only attained when executing four or more independent outer-product instructions concurrently (shown in Figure 3a). Therefore, implementations must employ multi-register kernels to enhance utilization across multiple matrix registers.

Table 2: Instructions per cycle (IPC) of different methods.

Method	Instructions Per Cycle
Vector-only	1.75
Matrix-only	1.46
Ideal	3.00

2.3.2 *Low Instruction Level Parallelism.*

While matrix instructions offer higher data throughput per instruction, their lower instruction throughput compared to vector instructions results in a notable ILP degradation. As shown in Table 2, the matrix-only approach achieves an IPC of only 1.46, significantly lower than the 1.75 of the vector-only method. However, analysis of ILP in Figure 3b reveals potential for optimization: overlapping eight outer product instructions with eight MLA operations demonstrates achievable ILP of 2.0. Furthermore, load/store instructions can be concurrently scheduled with computational instructions, potentially increasing the theoretical ILP upper bound beyond 2.0. These findings highlight the importance of efficient instruction scheduling to maximize pipeline utilization through interleaved execution of matrix instructions, vector instructions, and memory operations.

Table 3: L1 cache hit rates on out-of-cache stencils.

Matrix Size	Vector Method	Matrix Method
1024×1024	96.68%	66.16%
2048×2048	97.79%	66.26%
4096×4096	99.53%	34.58%
8192×8192	99.33%	32.51%

2.3.3 *Low Cache Hit Rates.*

While conventional caching policies work well for 1D vector-wise processing with contiguous memory access patterns, they exhibit significant inefficiencies when applied to 2D tensor-based computations. Traditional vector processing inherently exploits 1D spatial locality by sequentially updating contiguous memory regions. In contrast, Matrix-only implementations employ matrix-wise processing that operates on non-contiguous tiled matrix blocks, creating distinct 2D memory access patterns. Although commercial processors support strided prefetch[7], the complex memory access pattern of outer-product computation hinders the utilization of such hardware features. As shown in Table 3, the outer-product method has a lower L1-cache-hit rate of below 40% for large stencils, while vector-based solutions work well with the L1 cache. These limitations underscore the need for strategies to utilize cache for out-of-cache stencils.

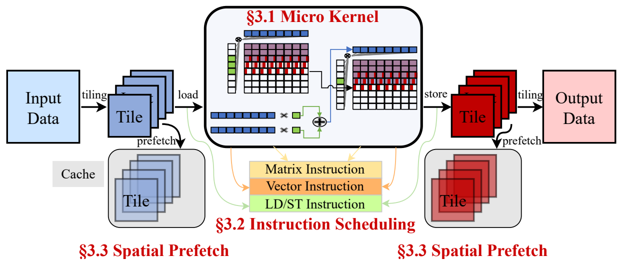


Figure 6: Overview of the *HStencil* framework.

3 Design

In this section, we present *HStencil*, a framework for accelerating stencil computations on modern CPUs. As shown in Figure 6, *HStencil* comprises three core components:

- Hybrid matrix-vector micro kernel for stencil computation, which optimizes matrix unit utilization and memory accesses by leveraging the strengths of both outer product and Multiply-accumulate operations.
- Fine-grained matrix-vector instruction scheduling method that maximizes instruction-level parallelism for matrix, vector, and load/store instructions.

- Spatial prefetch method that leverages the prefetch operations to utilize cache for matrix processing in stencil computation.

3.1 Micro Kernel

Towards efficient stencil computation on modern architectures, high-performance micro kernels must be designed to maximize matrix unit utilization and ensure contiguous memory access patterns. In this work, we first introduce optimized micro kernel designs, and then extend the single-register micro kernels to multi-register ones by integrating data tiling and data reuse strategies, enabling efficient utilization of matrix units.

Table 4: Notations of commonly-used symbols.

Notation	Value
T_{matrix}	Matrix Outer Product Computation Time
T_{vector}	Vector MLA Computation Time
C_{L1LD}	Load Cycles of L1 Cache
C_{L1ST}	Store Cycles of L1 Cache

3.1.1 Single-register Kernel. Performing outer product on the outer axis works well on dense stencils (e.g. box stencils). For sparse stencils like star, the outer-axis outer product approach suffers from low matrix register utilization, and the combined outer- and inner-product strategy introduces discontinuous memory accesses. To address this trade-off, we propose a novel hybrid methodology that balances register efficiency and memory access, leveraging scalable vector and matrix compute units to optimize kernel performance.

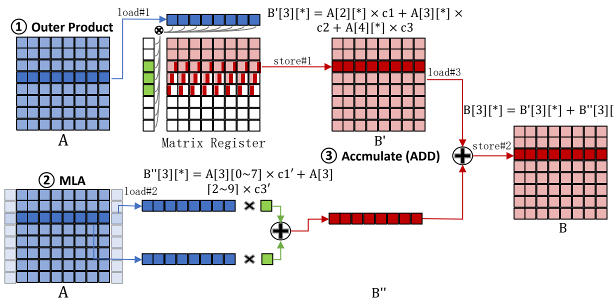

① Outer Product
load#1
B'[3][*] = A[2][*] × c1 + A[3][*] × c2 + A[4][*] × c3
store#1
load#3
A
Matrix Register
B'
B[3][*] = B'[3][*] + B''[3][*]
③ Accmulate (ADD)
store#2
② MLA
B''[3][*] = A[3][0~7] × c1' + A[3][2~9] × c3'
load#2
A
B''
B

Figure 7: Naive matrix-vector method.

Leveraging the MLA instructions, we treat them as alternatives in stencil computation where outer product instructions do not work well. As illustrated in Figure 7, our **naive implementation** integrates outer product and MLA instructions. The first step employs outer products for outer-axis computation, and store the intermediate results back to the memory. The second step utilizes MLA instructions for inner-axis computation. After that, intermediate results are reloaded from the memory, accumulated, and finally stored back to memory. Although this method successfully assigns appropriate tasks to matrix and vector units, computing matrix and vector results independently requires redundant load/store operations and harms overall performance.

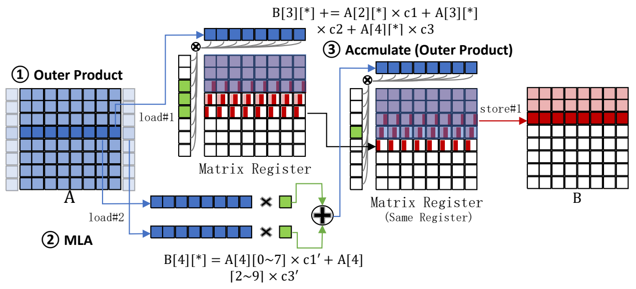

B[3][*] += A[2][*] × c1 + A[3][*] × c2 + A[4][*] × c3
③ Accmulate (Outer Product)
① Outer Product
load#1
Matrix Register
store#1
A
Matrix Register
(Same Register)
B
load#2
② MLA
B[4][*] = A[4][0~7] × c1' + A[4][2~9] × c3'

Figure 8: In-place accumulation matrix-vector method.

To avoid the overhead of an extra accumulation process, we propose a more efficient method enhanced by the **in-place accumulation**. As shown in Figure 8, the method continues to utilize matrix and vector units to compute the outer-axis and inner-axis components, and Algorithm 2 is the kernel implementation for this method. On the i_0-th iteration, the method computes the outer-axis part (line 8), and updates one portion from the $(i_0 - r)$-th to $(i_0 + r)$-th rows. Meanwhile, the vector computes the inner axis of the i_0-th row. Therefore, the $(i_0 - r)$-th row will be available after the i_0-th iteration, and it leverages an outer product instruction to accumulate the results from vector registers to matrix ones.

Algorithm 2: In-place accumulation matrix-vector kernel.

```
   Input: $A$
   Output: $B$
1  partition Xstart, Ystart, Xend, Yend
2  for $i \in \{\text{Ystart}, \ldots, \text{Yend}\}$ do
3      $i \leftarrow i + 1 \times \text{SVL}$
4      for $j \in \{\text{Xstart}, \ldots, \text{Xend}\}$ do
           /* Can extend to multiple registers      */
5          $j \leftarrow j + (\text{register\_count}) \times \text{SVL}$
           /* Micro Kernel Starts                   */
6          for $i_0 \in \{i - r, \ldots, i + \text{SVL} + r\}$ do
7              load $A_{i_0,j}$
               /* Outer product compute             */
8              $\text{ZAReg}{+} = cof1_{(i_0 - i + r)} \otimes A_{i_0,j}$
9              if $0 \le i_0 \le \text{SVL} - 1$ then
10                 for $j_0 \in \{-r, \ldots, r\}$ do
                       /* MLA Compute               */
11                     $\text{Reg1}{+} = A_{i_0, j+j_0} \times cof2_{j_0+r}$
                   /* Accmulate to matrix           */
12                 $\text{ZAReg}_{i_0}{+} = cofadd \otimes \text{Reg1}$
13             if $i_0 \ge r$ then
14                 store $B_{i_0-r,j} \leftarrow \text{ZAReg}_{i_0-r,j}$
```

Computation-overhead Saving. In the naive implementation, since the outer product operations and the MLA operations are performed independently, their execution cannot be overlapped, and the accumulation process requires an extra add operation as the overhead.

$$\begin{aligned} T_{\text{naive\_compute}} &= T_{\text{vector}} + T_{\text{matrix}} + T_{\text{naive\_overhead}} \\ T_{\text{naive\_overhead}} &= T_{\text{add}} \end{aligned} \tag{5}$$

As for the in-place accumulation method, we introduce a trick in the accumulation process. Traditionally, accumulating the vector and matrix results requires a multi-stage workflow: iterate over matrix register slices, perform slice-to-vector transfers, aggregate partial sums, and finally write results back to the memory. This process introduces overhead quantifiable as $T_{\text{m2v}} + T_{\text{add}}$, where T_{m2v} (matrix-to-vector transfer) dominates latency, requiring two times more cycles than outer product instructions.

Outer product instructions compute the outer product of two vectors, and perform a store-and-accumulate operation to matrix registers. By selectively enabling target matrix slices through coefficient vector adjustments, outer product instructions inherently consolidate store and addition operations into one single instruction. Consequently, overhead gets reduced as depicted by Equation 6. Also, since the matrix and vector instructions are interleaved, their execution time can be overlapped.

$$\begin{aligned} T_{\text{inplace\_compute}} &= \max(T_{\text{vector}}, T_{\text{matrix}}) + T_{\text{inplace\_overhead}} \\ T_{\text{inplace\_overhead}} &= T_{\text{outer\_product}} \end{aligned} \tag{6}$$

Memory-overhead Saving. In the naive method, the accumulation process also brings redundant loads/stores between registers and memory. As illustrated in Figure 7, it requires two loads for input data, one store for storing intermediate results, and extra one load and one store for the accumulation process. Overall, the total number of memory accesses is five, and the overhead is shown in Equation 7.

$$T_{\text{naive\_memory}} = 3 \times C_{\text{L1LD}} + 2 \times C_{\text{L1ST}} \tag{7}$$

The in-place method integrates the vector computation and matrix computation, thereby saving the memory overhead. First, it only needs one store, since it accumulates the vector results to matrix registers via outer product instructions. Second, it does not need an extra accumulation process, and saves the redundant loads/stores. Collectively, these optimizations reduce the total memory operations to two loads and one store per row of stencil computations. Totally, the number of memory accesses is three, and the memory overhead is shown in Equation 8.

$$T_{\text{inplace\_memory}} = 2 \times C_{\text{L1LD}} + 1 \times C_{\text{L1ST}} \tag{8}$$

3.1.2 Multi-register Kernel. Besides single-register kernels, achieving full utilization of scalable matrix compute units necessitates multi-register implementations. As scalable matrix compute units require concurrent execution of at least four independent outer product instructions to attain peak theoretical FLOPS, high-performance kernels should therefore be designed to utilize four or more matrix registers simultaneously. Effective utilization of multiple matrix registers requires loop unrolling to enable concurrent occupation of architectural resources and facilitate instruction-level parallelism. As illustrated in Figure 9, we implement loop unrolling on the j axis to enhance data locality by processing more tiles of data per iteration.

While loop unrolling enhances ILP, it introduces memory pressure, thereby necessitating complementary data reuse strategies. Stencil computations inherently exhibit interleaved data access patterns, providing opportunities for data reuse. Data reuse methods amplify these opportunities by leveraging vector EXT instructions to concatenate the adjacent vectors together to form the needed input vectors, as shown in Figure 9. By utilizing the data reuse method, the concatenated vectors can be used for both matrix and vector computation.

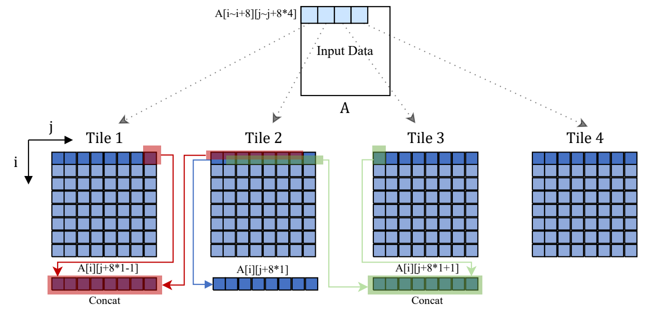


Figure 9: Data tiling and reuse. Loaded vectors are concatenated for $j + 8 * 1 - 1$ and $j + 8 * 1 + 1$ vectors.

3.2 Fine-grained Matrix-Vector Instructions Replacement and Scheduling

Efficient stencil computation not only needs optimized micro kernels to maximize computational unit utilization but also fine-grained instruction scheduling to overlap computation with memory operations. To fully utilize the ILP, we propose two novel techniques, including: vector instruction replacement to reduce pipeline stalls and matrix-vector instruction scheduling to orchestrate concurrent execution across computation, memory, and vector/matrix operation pipelines.

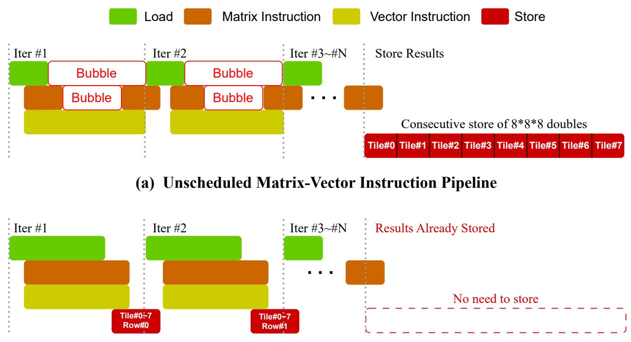
3.2.1 Vector Instruction Replacement. The integration of vector instructions into outer product-based stencil computation offers significant advantages. However, vector operations can become a performance bottleneck if utilizing a large amount of vector instructions. To address this, we analyze the execution cycles of vector and matrix instructions for box and star stencils in Table 5. Our results reveal that vector instructions dominate execution cycles in certain stencils, causing a potential performance limitation. To mitigate this, we propose strategies to reduce vector instructions while preserving computational efficiency.

Table 5: Matrix / vector instruction ratio.

Method	Cycles (Matrix / Vector)
Matrix Star & Box	40 / 0
Matrix-Vector Star	16 / 48
Matrix-Vector Box	40 / 32

MLA Instructions Replacement. Replacing outer products with vector MLA in stencil computation aims to use light-weighted vector instructions to complete operations where outer products are suboptimal. However, aggressive substitution creates an imbalance: after replacement, vector instructions dominate the execution time, leaving the scalable matrix compute units underutilized. To address this, we propose a partial rollback strategy, selectively reverting vector MLA instructions to outer product instructions. Outer product instructions leverage the idle matrix units, and we can trade latency increases for improved overall throughput via concurrent execution of matrix and vector instructions.

EXT Instructions Replacement. To minimize redundant memory access, we employ EXT instructions to concatenate interleaved input vectors. While this approach benefits memory access, it introduces pipeline contention, where MLA instructions compete with EXT for pipeline resources, creating performance bottlenecks. As stencil computation needs interleaved data and the memory access will hit the L1 cache, extra memory access is affordable. Thus, we alter some of the EXT instructions back to load instructions, thereby balancing more of the pipeline.



(a) Unscheduled Matrix-Vector Instruction Pipeline

(b) Scheduled Matrix-Vector Instruction Pipeline

Figure 10: Instruction scheduling of matrix instructions, vector instructions, load and store instructions.

3.2.2 Matrix-Vector Instruction Scheduling. After the replacement, the execution cycles of vector instructions, matrix instructions and load instructions are aligned, enabling their overlap to hide latency. We propose a matrix-vector scheduling method to orchestrate concurrent execution across the matrix, vector, and load/store pipelines. As shown in Figure 10, we load the input vectors at first. Once vectors are loaded, outer product instructions are dispatched to compute outer products along the outer axis, and EXT instructions are issued to carry out vector concatenation. For box stencils, the only used vector instructions are the EXT instructions for concatenating vectors to save interleaved data loading. Thus, we schedule to parallelize the vector EXT and load instructions, making sure the input data is always ready for matrix instructions. For star stencils, vector instructions are used for both concatenating vectors and computing. Similarly, we parallelize the EXT instructions and load instructions are carried out to prepare the data for computing. After that, the vector MLA instructions and matrix outer product instructions start to process the data and compute, maximizing the instruction throughput.

Beyond optimizing computation and load instructions, the store operations in stencil computations can be further optimized. As each step of the stencil computation will update multiple rows in the matrix register, it is completely safe to store only after the completion of all computations. This results in the consecutive storage of up to 512 64-bit floating points at a time, as shown in Figure 10. It imposes significant stress on the cache and memory, potentially leading to bandwidth bottlenecks. In our computation framework, the data in the stencil computations will be available before the entire computation is completed, offering opportunities to overlap store operations. For outer product-based stencil implementations, store operations need not be delayed until all computations are completed. The availability of data depends on the stencil radius r. Each iteration updates a portion of $2 \times r + 1$ rows. Therefore, the ith row of the ZA register is available after $2 \times r + i$ iterations, as shown in line 14. By scattering store operations throughout the iterations, we can mitigate the store bottleneck, thereby improving overall stencil computation efficiency.

3.3 Spatial Prefetch for Matrix Units

For in-cache stencils, performance improvements can be achieved through kernel- and instruction-level optimizations. In contrast, out-of-cache stencils necessitate memory-centric optimizations to maximize cache utilization. Caching policies in commercial CPUs are primarily optimized for vector-wise data access patterns, resulting in suboptimal performance for matrix-oriented computations that inherently require multidimensional data reuse. Therefore, we need to manually guide the cache to store the desired data.

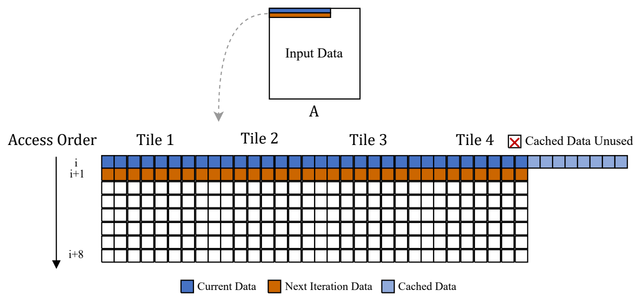


Figure 11: Data access pattern of *HStencil* computation.

On NVIDIA GPUs, where software-controlled caches (e.g., shared memory) enable explicit data management, programmers can manually cache critical stencil data to optimize tensor-wise computation. Likewise, on non-programmable caches, such as those in commercial CPUs, we leverage prefetch instructions as alternatives to manually guide the default caching policies to mitigate access inefficiencies.

Effective use of prefetch instructions in stencil computations requires a detailed analysis of memory access patterns. As illustrated in Figure 11, the *HStencil* computation for input matrix A exhibits a row-wise access pattern. During each iteration, the algorithm reads a contiguous block of data (up to 64 elements per row), which triggers hardware caching policies to prefetch adjacent memory addresses. However, subsequent iterations require data from the next row of the matrix, resulting in a non-contiguous access pattern

Algorithm 3: Spatial Prefetch in Stencil Computation

```
Input: $A$
Output: $B$
1 for $i \in \{\text{Ystart}, \ldots, \text{Yend}\}$ do
2     for $j \in \{\text{Xstart}, \ldots, \text{Xend}\}$ do
3         for $i_0 \in \{i - r, \ldots, i + \text{SVL} + r\}$ do
              /* Prefetch A[i0+1][*] and Load
                 A[i0][*]                         */
4             prefetch $A_{i_0+1,j}$, load $A_{i_0,j}$
5             (Matrix+Vector Computation)
              /* Prefetch B[i0-r][*]              */
6             prefetch $B_{i_0-r,j}$
7             (Matrix+Vector Computation)
8             if $i_0 \geq r$ then
                  /* Store to B[i0-r][*]          */
9                 store $B_{i_0-r,j} \longleftarrow \text{ZAReg}_{i_0-r,j}$
```

that renders default hardware prefetching ineffective. A similar inefficiency arises when storing results in matrix B, where updates are performed row-wise in contiguous blocks.

To address this, we strategically insert prefetch instructions to guide cache behaviors as shown in algorithm 3, transforming the hardware's default 1D spatial prefetching into an approximation of 2D data locality. For the input matrix A and result matrix B, we insert two critical prefetch operations during each iteration. The first is to prefetch the next row of matrix A required for the subsequent iteration into the L1 cache, as shown in line 4. The second is to prefetch the destination row in matrix B for the current iteration's results (line 6). For efficient prefetching, the prefetch instructions for A are scheduled alongside corresponding load instructions, priming the cache for future accesses. Matrix B's prefetch is embedded within the stencil computation phase, overlapping memory hints with arithmetic operations.

4 Architecture Generalization

Rather than being restricted to a single architecture, *HStencil* is a cross-architecture framework, ensuring performance portability across diverse hardware platforms by leveraging outer product units. In this work, we evaluate its performance portability through a case study on the Apple M4 CPU[2].

4.1 Micro Kernel Portability

Apple M4 CPU also consists of the scalable matrix compute units. However, it lacks vector MLA units and supports matrix compute units MLA[4] (M-MLA) as alternatives. The M-MLA instructions perform matrix multiply-accumulate operations on vector groups within the matrix registers. For stencil computations, we leverage the M-MLA instructions as alternatives, and perform stencil computation with MLA and outer products.

While M-MLA instructions also provide strong computing power, the cooperation between M-MLA and matrix outer products has been limited in exploration. The M-MLA instructions update fragmented parts (i.e. 0, 2, 4, 6 rows) of the matrix register, which makes the data layout different from the original form. Thus, the in-place accumulation is architecturally infeasible on the M4 CPU, necessitating reversion to the naive method.

4.2 Instruction Scheduling Portability

Stencil computation on the Apple M4 CPU incorporates scalable matrix compute units, load/store, and vector EXT instructions. While the M4 lacks vector MLA units, vector EXT instructions remain viable and can be used to optimize interleaved data loading. Also, on the Apple M4 CPU, independent vector EXT instructions can be effectively overlapped with the scalable matrix compute units instructions. By leveraging these instructions, we are able to employ fine-grained instruction scheduling to balance pipeline utilization across scalable matrix compute units, vector EXT, and LD/ST instructions, thereby maximizing ILP efficiency.

4.3 Prefetch Optimization Portability

Like mainstream commercial processors, the Apple M4 CPU lacks native hardware support for 2D tensor caching policies on matrix units. To address this, we apply spatial prefetch strategies by inserting explicit prefetch instructions for matrices, thereby leveraging cache utilization to mitigate memory access latency.

5 Evaluation

In this section, we first present the in-cache results of *HStencil* across various stencil benchmarks, and examine the details of the *HStencil* design contributing to the performance. Subsequently, we conduct out-of-cache experiments and portability studies, showing the effect of prefetch optimizations and the performance portability. To this end, we also conduct performance portability experiments on the Apple M4 CPU.

5.1 Experiment Platform

We conduct the in-cache and out-of-cache experiments on the newly released LX2 high-performance CPU. It utilizes a 512-bit vector length, processing 8 double-precision floating-point numbers at a time, and employs 8×8 matrix registers. The memory hierarchy of the LX2 CPU consists of an L1 data cache, an L1 instruction cache, and an L2 cache. For compilation, we used Clang 17.0.1[1] on the LX2 CPU.

Table 6: Different methods and their abbreviations.

Abbr.	Method
Auto	Auto-vectorization by Clang Compiler
Vector-only	Expert-optimized vector-based solution
Matrix-only	SOTA matrix-based solution (STOP[45])
HStencil	Our implementation

5.2 In-cache Experiments

In-cache experiments consist of three parts: 1) performance improvement; 2) performance breakdown; 3) Matrix-Vector cooperation.

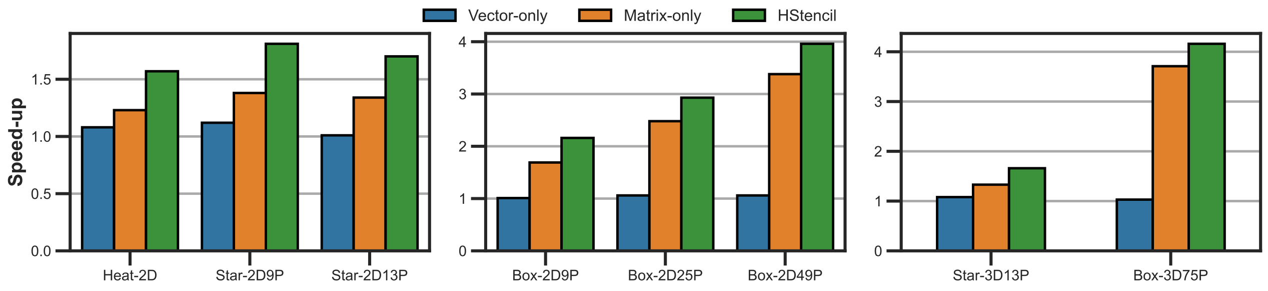


Figure 12: In-cache performance of *HStencil* versus matrix/vector-based methods in micro kernels of 128×128 size. Speedups are normalized to auto-vectorization.

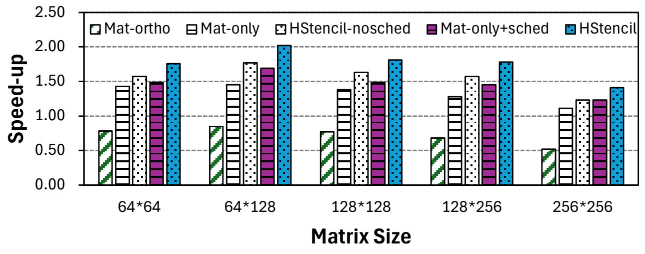
5.2.1 *Performance Improvement.* Firstly, we conduct a performance comparison of our approach against the Vector-only and Matrix-only methods. The experiments are conducted on in-cache 128×128 stencils, and the results are shown in Figure 12. The data is normalized to speedups compared to the auto-vectorization performed by the Clang compiler.

As illustrated in Figure 12, *HStencil* framework demonstrates superior performance compared to the Matrix-only and Vector-only methods. For star stencils, we employ loop tiling along the X-axis with a tiling factor of 8. *HStencil* achieves an average speedup of 1.69x, compared to 1.32x for the Matrix-only method. This performance advantage is maintained with different types of stencils. For box stencils, *HStencil* surpasses auto-vectorization by an average factor of 3.02x, while matrix-only achieves an average of 2.52x.

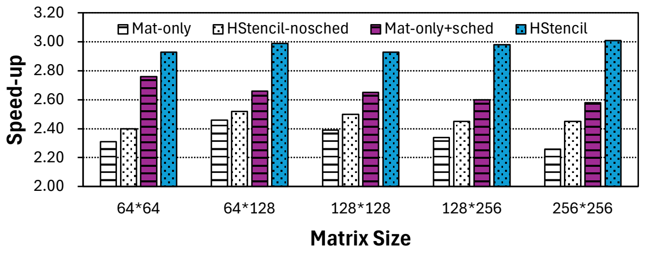
The 3D stencil computation can be viewed as a 2D stencil computation with different weights. *HStencil* accumulate the results over different 2D planes, and thus can be generalized to complete 3D computation. For star stencils, *HStencil* achieves a speedup of 1.66x, compared to 1.33x for the Matrix-only method. In box stencils, *HStencil* maintains its superior performance, achieving a speedup of 4.16x, compared to 3.71x for matrix-only.

5.2.2 *Performance Breakdown.* Aiming to demonstrate the performance enhancements achieved through our optimization techniques, we present a performance breakdown analysis as shown in Figure 13. The Mat-ortho employs both inner-axis and outer-axis outer products, and Matrix-only uses outer-axis outer products. In star stencils (Figure 13a), Mat-ortho is slower than auto-vectorization, and Mat-only achieves an average of 1.33x speedup over auto-vectorization. *HStencil* effectively leverages both matrix and vector unit capabilities, and achieves an average of 1.55x speedup without instruction scheduling. With instruction scheduling, *HStencil* attains 1.76x speedup. In box stencils (Figure 13b), Mat-only achieves an average of 2.34x speedup over auto-vectorization. *HStencil* achieves an average of 2.46x speedup without instruction scheduling and 2.96x speedup with instruction scheduling.

5.2.3 *Matrix-Vector Cooperation.* In the cooperation study, we use Instruction Per Cycle (IPC) as the primary metric to compare the performance of *HStencil* with the matrix-only and vector-only methods. As illustrated in Figure 14, the matrix-only method exhibits IPC lower than 1.60 because of the increased overhead associated with matrix instructions. Although the vector-only method achieves a moderate IPC of 1.825 on average, there remains room for improvement. *HStencil* harnesses the strengths of both the vector and matrix methods, achieving an IPC of at most 2.30 in stencil computation. It demonstrates superior performance, at most 1.31x and 1.59x faster than the vector and the matrix method in IPC, respectively.



(a) Breakdown for star stencils.



(b) Breakdown for box stencils.

Figure 13: Performance breakdown of *HStencil* in $r = 2$ 2D stencils.

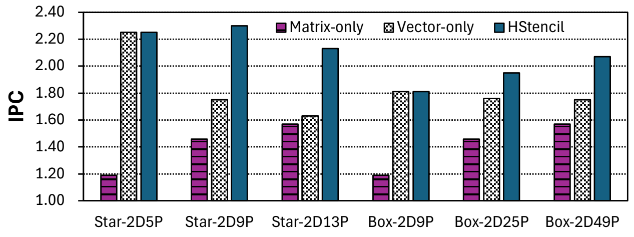


Figure 14: IPC comparison of *HStencil* and other stencil computation methods in 2D stencils of size 128×128.

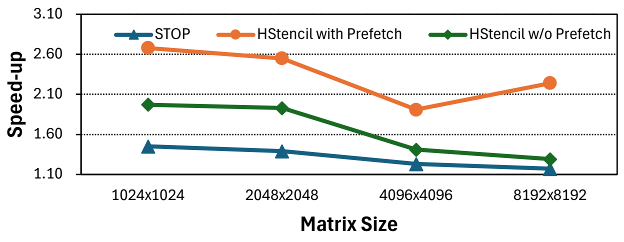


Figure 15: Performance speedups over auto-vectorization on out-of-cache matrix sizes.

5.3 Out-of-cache Experiments

Besides in-cache performance, we extend our evaluation to out-of-cache stencils on both single-core and multi-core.

5.3.1 Single-core Performance. Firstly, we examine the out-of-cache performance on a single core. As depicted in Figure 15, when executed without spatial prefetch optimization, the speedup of *HStencil* decreases as the matrix size increases. In contrast, spatial prefetch prevents this degradation, achieving an average speedup of 2.35x and delivering performance that is 42% faster than without such optimizations. Also, *HStencil* outperforms the STOP method by at most 91%, since STOP does not have instruction scheduling optimizations or prefetch optimizations.

In addition to the overall speedup, we analyze the L1 cache hit rates before and after applying the spatial prefetch optimization, as shown in Table 7. The results indicate that *HStencil* not only boosts the L1 hit rate (from approximately 30% to around 60%) but also increases the total L1 cache hit count by an average of 2.98x. These improvements suggest that better L1 cache utilization contributes significantly to the observed performance gains.

Table 7: L1 cache metrics of $r = 2$ box stencils.

Method	w/o Prefetch		with Prefetch	
L1 Cache Metrics	Hit Rate	Hit Times	Hit Rate	Hit Times
1024×1024	66.16%	2.5×10^5	71.72%	5.8×10^5
2048×2048	66.26%	1.1×10^6	73.66%	2.6×10^6
4096×4096	34.58%	4.4×10^6	60.21%	1.6×10^7
8192×8192	33.51%	1.7×10^7	59.74%	6.1×10^7

5.3.2 Multi-core Performance. In addition to single-core evaluations, we examine the scalability of stencil computations from one core up to 32 cores. As illustrated in Figure 16 for an Box-2D9P stencil with the size of $8192x8192$, *HStencil* achieves a strong scaling performance, reaching 12.91 GStencil/s on 32 cores. It substantially outperforms the matrix-only (7.76 GStencil/s) and vector-only (7.14 GStencil/s) approaches.

5.4 Performance Portability on Apple CPUs

The performance portability study is conducted on the Apple M4 Pro CPU[2]. It also has 8 matrix registers and can process up to $8 \times 8 \times 8$ double precision numbers in parallel. The Apple M4 Pro

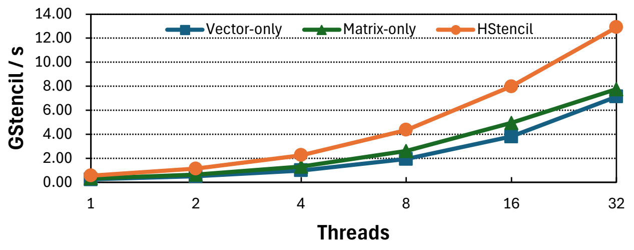


Figure 16: Scaling *HStencil* from 1 core to 32 cores on Box-2D9P stencil with the size of 8192×8192.

CPU features a 128KB L1 data cache, a 192KB L1 instruction cache, and a 4MB shared L2 cache. We utilize the Apple Clang 16.0.0 (clang-1600.0.26.6) [3] as the compiler.

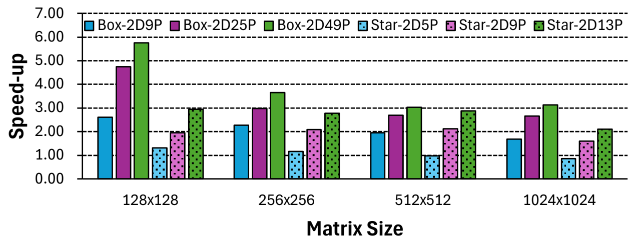


Figure 17: Speedups of *HStencil* over auto-vectorization in 2D stencils on Apple M4 Pro CPU.

On the Apple M4 Pro CPU, we evaluate the performance of *HStencil* on 2D stencils, as shown in Figure 17. Notably, with a 1024KB L1 cache, we can conduct experiments on stencils up to 1024×1024 in size. *HStencil* achieves an average of 3.07x speedup over auto-vectorization for all box stencil sizes and demonstrates an average speedup of 1.90x for star stencils. Consistent with the performance observed on the LX2 CPU, these results highlight the performance portability of *HStencil* across different modern CPUs.

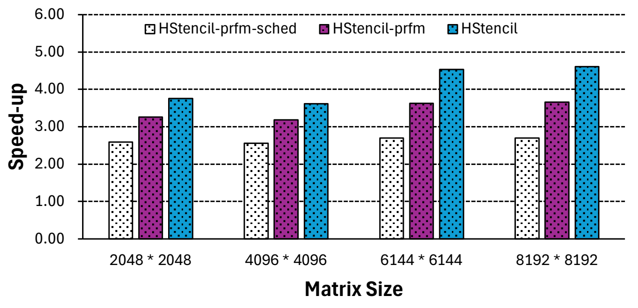


Figure 18: Speedups over auto-vectorization on out-of-cache matrix sizes on the Apple M4 Pro CPU.

Besides experiments on small matrix sizes, we also conduct out-of-cache experiments on $r = 2$ box stencils, as shown in Figure 18. The results show the effect of instruction scheduling and spatial prefetch on the Apple M4 CPU. Without such optimizations, *HStencil* can only obtain an average speedup of 2.63x. Aiding by instruction scheduling and spatial prefetch, *HStencil* can gain 30% and 20% more performance on average, respectively.

6 Related Work

6.1 Stencil Optimization on Matrix Units

Recent research has explored matrix-based approaches to stencil computations using NVIDIA GPU Tensor Cores, aiming to further enhance performance. TCStencil [24] aims to leverage Tensor Cores for handling complex stencil patterns. [19] enhances TCStencil by introducing temporal blocking, which results in performance speed-ups. ConvStencil [9] is inspired by the analogy between convolution and stencil computations, proposing a Stencil2row method akin to the im2row technique used in convolutional neural networks (CNNs). LoRAStencil [43], motivated by Low-Rank Adaptation (LoRA) techniques in model training, leverages matrix chain multiplication to reduce redundant memory accesses.

6.2 Matrix Units for HPC

Besides stencil computations, matrix units from various vendors have been extensively investigated for many HPC applications. Research on Tensor Cores has explored a wide range of domains, including MM-like operations such as matrix factorization [20, 38, 40], GEMM with extended precision [13, 23], and sparse matrix multiplication [12, 21, 22, 31, 35, 36]. Additionally, for other linear algebra operations, studies have investigated reduction and scan operations [11], FFT [28] and other linear algebra aspects [39, 41]. Intel AMX units have been optimized for GEMM operations [27] and AI-accelerated CFD simulations [15]. For CPUs, studies have included efficient FFT methods using outer product units [37]. Additionally, preliminary work on the Apple M4 CPU has explored GEMM operations on outer product units [29]. Among these works leveraging matrix units, *HStencil* is the first to conduct evaluations on both LX2 and Apple M4 CPUs.

6.3 Cooperation of Matrix and Vector Units

Vector and matrix units, optimized for distinct computational patterns, can provide performance benefits when used together. On NVIDIA GPUs, concurrent execution of Tensor Core kernels and CUDA Core kernels has been demonstrated in [8, 42]. Research has explored their collaborative potential, including scheduling strategies to overlap their execution [44], integration to enhance GEMM performance and GPU throughput [17], and leveraging both for accelerating sparse matrix-vector multiplication [25, 26]. To the best of our knowledge, no existing studies have investigated the cooperation of matrix and vector units on modern CPUs. *HStencil* is the first work to explore the cooperation between scalable vector and matrix compute units.

7 Conclusion

Stencil computation is fundamental in scientific computing, typically requiring efficient processing of grid-based data. In this study, we introduce *HStencil*, the first practical framework that utilizes both scalable vector and matrix compute units to efficiently compute stencils on new generation CPUs. Additionally, our method is the first to target both LX2 and Apple M4 CPUs, and realizes performance improvement on both. Besides the main contributions, our work provides several takeaways:

- Matrix and vector instructions are executed on different units, and can thus be issued simultaneously on modern CPUs like LX2 CPU and M4 CPUs.
- To accumulate vector results to matrix registers, the fastest way is to use the outer product operations.
- Scalable vector compute units performance portability was not initially guaranteed between different CPUs. One needs to perform a transformation from vector MLA instructions to alternative MLA instructions on Apple M4 CPUs.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by Guangdong S&T Program under Grant No. 2024B0101040005, NSFC Grants (62461146204, 62472462). Yutong Lu is the corresponding author of this paper.

References

[1] LLVM Project 2023. *Clang Compiler User's Manual.* LLVM Project. https://releases.llvm.org/17.0.1/tools/clang/docs/index.html Version 17.0.1.
[2] Apple. 2024. Apple introduces M4 Pro and M4 Max. https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/ Apple Newsroom.
[3] Apple. 2024. Command Line Tools for Xcode 16.2 Beta 2. https://download.developer.apple.com/Developer_Tools/Command_Line_Tools_for_Xcode_16.2_beta_2/Command_Line_Tools_for_Xcode_16.2_beta_2.dmg Apple Developer Tools.
[4] ARM. 2023. *SME and SME2 Overview.* https://developer.arm.com/documentation/109246/0100/SME-Overview/SME-and-SME2
[5] ARM. 2023. Streaming SVE mode and ZA storage. https://developer.arm.com/documentation/109246/0100/Introduction/The-Scalable-Matrix-Extensions/Streaming-SVE-mode-and-ZA-storage. Accessed: 2023-10-24.
[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67. doi:10.1145/1562764.1562783
[7] Jean-Loup Baer and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91).* Association for Computing Machinery, New York, NY, USA, 176–186. doi:10.1145/125826.125932
[8] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16).* Association for Computing Machinery, New York, NY, USA, 681–696. doi:10.1145/2872362.2872368
[9] Yuetao Chen, Kun Li, Yuhao Wang, Donglin Bai, Lei Wang, Lingxiao Ma, Liang Yuan, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. ConvStencil: Transform Stencil Computation to Matrix Multiplication on Tensor Cores. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24).* Association for Computing Machinery, New York, NY, USA, 333–347. doi:10.1145/3627535.3638476
[10] Nvidia Corporation. 2021. Nvidia A100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper
[11] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19).* Association for Computing Machinery, New York, NY, USA, 46–57. doi:10.1145/3330345.3331057
[12] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems, Volume 3 (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 253–267. doi:10.1145/3620666.3651378

[13] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 278–291. doi:10.1145/3437801.3441599

[14] Google Cloud. 2021. *Cloud TPU*. https://cloud.google.com/tpu

[15] Kamil Halbiniak, Krzysztof Rojek, Sergio Iserte, and Roman Wyrzykowski. 2024. Unleashing the Potential of Mixed Precision in AI-Accelerated CFD Simulation on Intel CPU/GPU Architectures. In *Computational Science – ICCS 2024*, Leonardo Franco, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer Nature Switzerland, Cham, 203–217.

[16] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *Compiler Construction*, Jens Knoop (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 225–245.

[17] Khoa Ho, Hui Zhao, Adwait Jog, and Saraju Mohanty. 2022. Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 223–228. doi:10.1109/ISVLSI54635.2022.00051

[18] Intel Corporation. 2021. *Advanced Matrix Extensions Overview*. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html

[19] Futa Kambe and Toshio Endo. 2024. Accelerating Stencil Computations on a GPU by Combining Using Tensor Cores and Temporal Blocking. In *Proceedings of the 16th Workshop on General Purpose Processing Using GPU* (Edinburgh, United Kingdom) *(GPGPU '24)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3649411.3649412

[20] Vinay Kukutla, Ramachandra Achar, and Wai Kong Lee. 2024. TC-GVF: Tensor Core GPU based Vector Fitting via Accelerated Tall-Skinny QR Solvers. *IEEE Transactions on Components, Packaging and Manufacturing Technology* (2024), 1–1. doi:10.1109/TCPMT.2024.3410298

[21] Eunji Lee, Yoonsang Han, and Gordon Euhyun Moon. 2024. Accelerated Block-Sparsity-Aware Matrix Reordering for Leveraging Tensor Cores in Sparse Matrix-Multivector Multiplication. In *Euro-Par 2024: Parallel Processing*, Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber (Eds.). Springer Nature Switzerland, Cham, 3–16.

[22] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient quantized sparse matrix operations on tensor cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 37, 15 pages.

[23] Zejia Lin, Aoyuan Sun, Xianwei Zhang, and Yutong Lu. 2024. MixPert: Optimizing Mixed-Precision Floating-Point Emulation on GPU Integer Tensor Cores. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Copenhagen, Denmark) *(LCTES 2024)*. Association for Computing Machinery, New York, NY, USA, 34–45. doi:10.1145/3652032.3657567

[24] Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. 2022. Toward accelerated stencil computation by adapting tensor core unit on GPU. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 28, 12 pages. doi:10.1145/3524059.3532392

[25] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 73, 14 pages. doi:10.1145/3581784.3607051

[26] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. 2024. AmgT: Algebraic Multigrid Solver on Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 52, 16 pages. doi:10.1109/SC41406.2024.00058

[27] Chandra Sekhar Mummidi, Victor C. Ferreira, Sudarshan Srinivasan, and Sandip Kundu. 2024. Highly Efficient Self-checking Matrix Multiplication on Tiled AMX Accelerators. *ACM Trans. Archit. Code Optim.* 21, 2, Article 21 (Feb. 2024), 22 pages. doi:10.1145/3633332

[28] Louis Pisha and Łukasz Ligowski. 2021. Accelerating non-power-of-2 size Fourier transforms with GPU Tensor Cores. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 507–516. doi:10.1109/IPDPS49936.2021.00059

[29] Stefan Remke and Alexander Breuer. 2024. Hello SME! Generating Fast Matrix Multiplication Kernels Using the Scalable Matrix Extension. arXiv:2409.18779 [cs.DC] https://arxiv.org/abs/2409.18779

[30] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. *SIGPLAN Not.* 49, 6 (June 2014), 65–76. doi:10.1145/2666356.2594342

[31] Haotian Wang, Wangdong Yang, Rong Hu, Renqiu Ouyang, Kenli Li, and Keqin Li. 2023. A Novel Parallel Algorithm for Sparse Tensor Matrix Chain Multiplication via TCU-Acceleration. *IEEE Transactions on Parallel and Distributed Systems* 34, 8 (2023), 2419–2432. doi:10.1109/TPDS.2023.3288520

[32] Finn Wilkinson and Simon McIntosh-Smith. 2022. An Initial Evaluation of Arm's Scalable Matrix Extension. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 135–140. doi:10.1109/PMBS56514.2022.00018

[33] Jiabin Xie, Guangnan Feng, Han Huang, Junxuan Feng, Zhiguang Chen, and Yutong Lu. 2024. Extreme-scale Direct Numerical Simulation of Incompressible Turbulence on the Heterogeneous Many-core System. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 120–132. doi:10.1145/3627535.3638479

[34] Liang Yuan, Hang Cao, Yunquan Zhang, Kun Li, Pengqi Lu, and Yue Yue. 2021. Temporal vectorization for stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. doi:10.1145/3458817.3476149

[35] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. *CoRR* abs/2009.14600 (2020). arXiv:2009.14600 https://arxiv.org/abs/2009.14600

[36] Kaige Zhang, Xiaoyan Liu, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. Jigsaw: Accelerating SpMM with Vector Sparsity on Sparse Tensor Core. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 1124–1134. doi:10.1145/3673038.3673108

[37] Ruge Zhang, Haipeng Jia, Yunquan Zhang, Baicheng Yan, Penghao Ma, Long Wang, and Wenxuan Zhao. 2024. OpenFFT-SME: An Efficient Outer Product Pattern FFT Library on ARM SME CPUs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 938–949. doi:10.1109/IPDPS57955.2024.00088

[38] Shaoshuai Zhang, Elaheh Baharlouei, and Panruo Wu. 2020. High Accuracy Matrix Computations on Neural Engines: A Study of QR Factorization and its Applications. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 17–28. doi:10.1145/3369583.3392685

[39] Shaoshuai Zhang, Vivek Karihaloo, and Panruo Wu. 2020. Basic Linear Algebra Operations on TensorCore GPU. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 44–52. doi:10.1109/ScalA51936.2020.00011

[40] Shaoshuai Zhang, Ruchi Shah, Hiroyuki Ootomo, Rio Yokota, and Panruo Wu. 2023. Fast Symmetric Eigenvalue Decomposition via WY Representation on Tensor Core. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 301–312. doi:10.1145/3572848.3577516

[41] Shaoshuai Zhang and Panruo Wu. 2021. Recursion Brings Speedup to Out-of-Core TensorCore-based Linear Algebra Algorithms: A Case Study of Classic Gram-Schmidt QR Factorization. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 80, 11 pages. doi:10.1145/3472456.3473522

[42] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 58–68. doi:10.1145/3330345.3330351

[43] Yiwei Zhang, Kun Li, Liang Yuan, Jiawen Cheng, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. LoRAStencil: Low-Rank Adaptation of Stencil Computation on Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 53, 17 pages. doi:10.1109/SC41406.2024.00059

[44] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. 2021. Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 290–298. doi:10.1109/ICCD53106.2021.00054

[45] Wenxuan Zhao, Liang Yuan, Baicheng Yan, Penghao Ma, Yunquan Zhang, Long Wang, and Zhe Wang. 2024. Stencil Computation with Vector Outer Product. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 247–258. doi:10.1145/3650200.3656611

Appendix: Artifact Description

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

This paper presents ***HStencil***, a hybrid and heterogeneous stencil computing framework designed to maximize performance on modern CPU architectures with scalable vector and matrix compute units. The main contributions of *HStencil* are listed as follows:

C_1 We present *HStencil*, a novel stencil framework to efficiently leverage outer product-based matrix units. It combines scalable vector and matrix compute units to maximize computational throughput and memory efficiency on next-generation CPU architectures.

C_2 We further incorporate enhanced micro kernels with in-place accumulation methods, instruction scheduling to exploit ILP across matrix/vector pipelines, spatial prefetch to minimize overhead for out-of-cache stencils.

C_3 Experimental results on representative benchmarks demonstrate that *HStencil* significantly improves overall execution performance, far outperforming the state-of-the-art methods. Moreover, *HStencil* optimizations can be generalized to other heterogeneous architectures, and is the first efficient stencil algorithm tailored to Apple M4 CPUs.

A.2 Computational Artifacts

The computational artifacts are listed in the DOI below:

A_1 https://doi.org/10.5281/zenodo.15273411

The table below shows the detailed information about the artifact and its related paper elements.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Figure 12-18

B Artifact Identification

In the following subsections, we outline the swift installation and deployment of the *HStencil* framework:

(1) **Dependency installation:**
  - Install all required dependencies on the LX2 and Apple M4 CPU, including: a) Vendor-specific compiler, b) CMake, and c) Performance-analysis tools. (e.g., `perf`)

(2) **Experimental evaluation:**
  - **In-cache evaluaion:** a) Microkernel speedup comparision, b) performance breakdown, and c) ILP analysis.
  - **Out-of-cache evaluaion:** a) Cache behavior study on out-of-cache kernels, and b) run experiments across multiple cores within a single NUMA node.
  - **Performance portability evaluaion:** Executing *HStencil* on the Apple M4 CPUs, and obtain speedups versus auto-vectorization on M4 CPUs.

(3) **Figure generation:**
  - Get all the results and fill them in the excel or python script in the (`HStencil/figures/`) . Thus, users can get the figures as in the paper.

B.1 Computational Artifact A_1

Relation To Contributions

The artifact consists of multiple parts of the results.

(1) The first part is the in-cache results. The results are related to the contributions C_1, C_2.
(2) The second part is the out-of-cache results. The results are related to the contributions C_2.
(3) The third part is the performance portability analysis on the Apple M4 CPUs. It evaluates C_3 on the different architecture.

Expected Results

In-cache evaluation: *HStencil* should gain up to 30% speedup to the matrix-only (e.g. STOP algorithm) solutions in microkernels. Besides the performance, the Instruction Per Cycle (IPC) of *HStencil* should be higher than both the vector-only method and the matrix-only method.

Out-of-cache evaluation: *HStencil* should gain up to 90% speedup to the matrix-only solutions on out-of-cache stencils, and should obtain higher cache hit times and cache hit rates. On the multiple core experiments, *HStencil* should show a good scalability within the 32 cores in a single NUMA node of LX2 CPU.

Performance portability evaluation: *HStencil* should show performance speedups on the M4 CPU.

Expected Reproduction Time (in Minutes)

For the artifact setup, the installation of the dependencies should take more than 60 min to download and about 30 min to install with `make -j`. After that, the compilation of *HStencil* for multiple workloads takes about 5 minutes to finish.

The *HStencil*-optimized stencils should be tested on both the LX2 CPU and the Apple M4 Pro CPUs. The total evaluation time for all the benchmarks should take about 20 min.

Artifact Setup (incl. Inputs)

Hardware. *HStencil* currently supports mainstream CPUs with scalable vector and matrix compute units. Two of the recommended CPUs are the LX2 CPU and the Apple M4 CPU. (e.g. Macbook pro, Mac mini) The in-cache and out-of-cache experiments are conducted on LX2 CPUs, and the performance portability experiments are conducted on M4 CPUs. The Apple M4 CPU consists of a 128KB L1 data cache, a 192KB L1 instruction cache, and a 4MB shared L2 cache. The memory hierarchy of the LX2 CPU consists of a L1 data cache, a L1 instruction cache, and a L2 cache. Both of the two CPUs use DDR as the memory.

Software. The evaluation targets two platforms and their respective software configurations:

LX2 (LX2-specific Ubuntu) The LX2 experiments run under the LX2-specific Ubuntu operating system. System software and profilers must be preinstalled. Required dependencies:
  - **Clang (LX2-specific)**: available from software package in the `dependencies.sh`.
  - **CMake v3.31.0**: (https://cmake.org/download/)
  - **perf v5.10.0**: Linux performance analysis tool

- **numactl (LX2-specific)**: for NUMA control and affinity

Apple M4 (macOS) The portability experiments run on macOS with Apple M4 CPUs. Required dependencies:

- **Homebrew v4.5.7**: package manager, (https://brew.sh)
- **Apple Clang v16.2**: included in CommandLine Tools for Xcode (https://download.developer.apple.com/Developer_Tools/Command_Line_Tools_for_Xcode_16.2/Command_Line_Tools_for_Xcode_16.2.dmg)
- **CMake v3.31.0**: (https://cmake.org/download/)
- **Xcode v16.4** (https://developer.apple.com/services-account/download?path=/Developer_Tools/Xcode_16.4/Xcode_16.4.xip)

Datasets / Inputs. The datasets employs a suite of representative stencil benchmarks across varying dimensions and sizes. The stencil types consist of:

- 2D and 3D `Star` stencils
- 2D and 3D `Box` stencils
- `Heat-2D` stencil

And the problem sizes consist of:

- **In-cache**: problem sizes that fit within the cache (from 64×64 to 256×256)
- **Out-of-cache**: sizes that exceed the cache (from 1024×1024 to 8192×8192)

These datasets are generated automatically, and users can modify the weights of the stencil matrix.

Installation and Deployment. Before configuring the *HStencil*, we need to install the dependencies, and the installation processes on the two CPUs are different.

On the LX2 CPU, `perf` tool should be first installed. Also, it should be ensured that the user has sufficient privileges to capture hardware metrics. Then users can execute the script in the path `HStencil/dependencies/LX2/dependencies.sh`, which downloads and install the Clang component of the software package. The script will also download and install the `CMake`.

On the Apple M4 CPU, users should first install `homebrew`, which is the package manager of the macOS. After that, users can execute the `HStencil/dependencies/M4/dependencies.sh` script, installing `CMake` and the `CommandLine Tools` for Xcode. Although macOS includes a default Clang, this artifact relies on the exact version of Apple-provided toolchain.

After the pre-requirements are installed, we can compile *HStencil* to generate distinct kernels for the multiple stencils workloads. Our kernels are written in hybrid intrinsics and assembly code, so we need to set the target ISA extensions at compile time. On the LX2 CPU, we use the `-mcpu=lx2` to activate both scalable vector and matrix compute units. On the M4 CPU, we use the isa specific flags to guarantee the ISA support. We recommend `-O3` for maximum optimization and auto-vectorization. The `-rtlib=compiler-rt` is passed as the *LD_FLAG* to link the runtime library for Clang. The `compile.sh` will set all the options and users can just execute this script.

Artifact Execution

The artifact comprises three independent evaluation tasks: 1) in-cache, 2) out-of-cache, and 3) performance portability. The three tasks have no dependencies, and for simplicity we can execute these tasks in sequence, with scripts provided in the `HStencil/scripts`. Before execution, source the provided `env.sh` script to configure environment variables (e.g., software paths and toolchain settings). On LX2 CPUs, each execution must also bind cores and memory using numactl (for example, numactl –all -C $core -m $mem_node). For example, for the out-of-cache task, users should first set up the environment by `env.sh`, and run the script in `outcache.sh`, which will run all the tasks, and store the corresponding results in the `results` folder.

In-cache evaluation: It comprises three sequential analyses, corresponding to Figure 12-14. The first part is comparing the performance of *HStencil* versus the matrix-only implementation and a vector-only implementation (Figure 12). In this part, users run 3 different versions of kernels, and compare their performance. Next, we provide an ablation study of *HStencil* (Figure 13), comparing the the effect of the multiple optimizations. Finally, we evaluate the effectiveness of *HStencil* on the ILP. We use `perf stat -e instructions,cycles` to collect the IPC of the *HStencil*, vector-only version and matrix-only version, to see how *HStencil* improve the IPC.

Out-of-cache evaluation: This evaluation comprises two experiments, corresponding to Figure 15-16 and Table 7. We first conduct an experiment on the effect of the prefetch instructions. We collect the overall time, as well as collecting the cache metrics using `perf stat -e L1-dcache-loads,L1-dcache-load-misses`. Secondly, by using OpenMP, we scale *HStencil* to multiple cores (maximum 32 cores) in a numa node, and see the scalability of *HStencil*. For efficient scaling, we should bind all the cores to a single NUMA node, (e.g. `numactl -all -C 0-31 -m 0`), and set the `OMP_PROC_BIND` to be `close`.

Performance portability evaluation: It corresponds to the result of Figure 17-18. We conduct the microkernel experiments on the macbook M4 Pro. The experiments are compared to the auto-vectorization baseline on the M4, which is done by the Apple clang using the neon instruction set.

Artifact Analysis (incl. Outputs)

The experimental results will be printed on the log. Users can collect the results, and then simply open the provided Excel template (`HStencil/figures/data.xlsx`) and replace its placeholder values with new results. The built-in charts will automatically update to match the paper's figures. Note that Figure 12 is produced by the Python script `plot.py`, users can update the data section of plot.py, and then run `python plot.py`.