Published: 16 November 2025

Citation in BibTeX format

RACS '25: International Conference on Research in Adaptive and Convergent Systems
November 16 - 19, 2025
Ho Chi Minh, Vietnam

Conference Sponsors:
SIGAPP

RESEARCH-ARTICLE

Adaptive Block Size Selection for Translating Triton Kernels to RVV

LIU YUHAO, The Chinese University of Hong Kong, Shenzhen, Shenzhen, Guangdong, China

WILLIAM KEVIN, The Chinese University of Hong Kong, Shenzhen, Shenzhen, Guangdong, China

FEIGE ZHOU, The Chinese University of Hong Kong, Shenzhen, Shenzhen, Guangdong, China

YEH CHING CHUNG, The Chinese University of Hong Kong, Shenzhen, Shenzhen, Guangdong, China

WEICHUNG HSU, The Chinese University of Hong Kong, Shenzhen, Shenzhen, Guangdong, China

Adaptive Block Size Selection for Translating Triton Kernels to RVV

Yuhao LIU
Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong
224040365@link.cuhk.edu.cn

KEVIN WILLIAM*
Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong
122040033@link.cuhk.edu.cn

Feige Zhou*
Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong
122090815@link.cuhk.edu.cn

Yeh-Ching Chung
Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong
ychung@cuhk.edu.cn

WEI-CHUNG HSU
Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong
hsuweichung@cuhk.edu.cn

ABSTRACT

The increasing demand for efficient AI inference on edge devices has intensified the need for high-performance and portable programming models. Triton, a tile-based language designed for GPUs, lowers the threshold of writing efficient custom AI kernels. This paper addresses the performance portability of translating Triton kernels to edge devices such as RISC-V CPUs equipped with Vector Extension (RVV). We identify that a direct port of Triton kernels with GPU-tuned parameters, `BLOCK_SIZE`, results in significant performance degradation on RVV due to fundamental architectural differences. Through a detailed empirical evaluation on several benchmarks, we identify the performance impact of tiling parameters on vector register spilling and cache performance. Our analysis reveals a critical trade-off: some `BLOCK_SIZE` values improve cache and prefetching performance but increase vector register pressures and cause intense vector register spilling. Based on these insights, we develop some heuristics for selecting an adaptive block size to balance the performance trade-off between cache performance and vector register spilling, mapping Triton kernels more efficiently onto RVV-based systems. This enhances the performance portability of Triton kernels for heterogeneous computing in edge AI inference.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; **Heterogeneous (hybrid) systems**; • **Software and its engineering → Retargetable compilers**; **Dynamic compilers**; **Runtime environments**.

*Both authors contributed equally to this research.

KEYWORDS

AIPC, AI Inference, RISC-V, RVV, SIMD, Vector, Triton, GPU, Performance Portability, Cache

ACM Reference Format:
Yuhao LIU, KEVIN WILLIAM, Feige Zhou, Yeh-Ching Chung, and WEI-CHUNG HSU. 2025. Adaptive Block Size Selection for Translating Triton Kernels to RVV. In *International Conference on Research in Adaptive and Convergent Systems (RACS '25), November 16–19, 2025, Ho Chi Minh, Vietnam.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3769002.3769969

1 INTRODUCTION

In the contemporary high-performance computing (HPC) landscape, the demand for efficient Artificial Intelligence (AI) inference is escalating across a wide range of devices, from centralized cloud servers to edge and personal computing platforms [1, 2, 5]. While AI inference on heterogeneous system is becoming an important direction, Artificial Intelligence Personal Computer (AIPC) has also been increasingly attractive since AIPC often equipped with powerful SIMD/Vector computing capability. The reduced latency, improved data privacy and security make AIPC ideal for edge inference. Some work like T-MAC [9] and CoDL [4], have demonstrated the feasibility of deploying AI inference on AIPC.

To meet the demanding computing requirements of inference on CPUs, the ability to Single instruction with Multiple data (SIMD) is indispensable. While the well-known SIMD ISAs are X86 SSE/AVX and ARM Neon, RISC-V with Vector Extension (RVV) [6] is viewed as another young but powerful alternative. RVV is more scalable and flexible compared to other older alternatives. RVV has some unique features. One key feature is the use of the vector length register, *vl*, to specify the number of elements to be processed instead of bundling with the SIMD instruction. This enables performance portability when moving an application binary across various vector implementations. LMUL, the vector register grouping mechanism, is another key feature used to view multiple short vector registers as a longer one to improve vector execution efficiency. In general, RVV is open, modular, and extensible, and these features make it a good candidate for conducting edge inference.

Machine Learning (ML) workloads (including training and inference) have a high demand for memory bandwidth. This is why high-end NVIDIA Hopper architecture includes High Bandwidth Memory (HBM). Even with HBM, ML workloads still require sophisticated data management to make efficient use of shared memory and the L1/L2 data cache hierarchy in NVIDIA chips. Such complex data management is challenging, and is often conducted by CUDA programmers. For years, the community has tried to come up with some tile-based programming languages so that computation can be represented in blocks and sub-blocks. Once the algorithm is presented by blocks, the compiler can try to allocate such blocks in shared memory to alleviate the burden of programmers to manage the complex data movement. One of the most popular tile-based languages is Triton [7]. Triton is an open-source, Python-based language supported by OpenAI, designed to abstract GPU programming, which simplifies the creation of highly efficient custom kernels, and it has demonstrated performance comparable to that of carefully tuned CUDA code. However, Triton's application to CPU architectures is not yet fully explored. Given the wide adoption of heterogeneous computing for ML workloads, a custom kernel function should be able to be deployed on either GPU or CPU. If Triton becomes the choice of language for writing custom kernels on GPU, we would expect translating them efficiently to CPU would contribute significantly to future AI computing, especially on AIPC edge devices.

While Triton excels on GPUs, Triton's model of tiling, vectorizing, and cache optimization has strong analogies to the CPU side, especially when CPUs support wide SIMD/Vector instructions (Table 1). The vector or SIMD registers enable parallel operations on multiple data elements, much like registers used across multiple threads in a GPU warp. The scratchpad memory, or local memory on the CPU, mirrors the shared memory of GPU architecture due to its high bandwidth, fast access, and programmable features. For the cache hierarchy, both GPU and CPU architectures use a private L1 cache for each core (CPU) or Streaming Multiprocessor (SM) (GPU), and a larger L2 cache shared by a cluster of CPU cores or across all SMs, respectively. This analogy makes translating Triton to the CPU a promising prospect.

As RVV becomes increasingly popular for edge computing, porting a high-level tile-based programming model like Triton to RVV will attract more attention. The main challenge of translating Triton to CPU lies in performance portability. The parameter `BLOCK_SIZE` is often chosen for GPU, which may work poorly on CPUs such as RVV. A simple method to select the best parameters for either GPU or CPU is auto-tuning. However, it would take a considerable amount of time (up to multiple days) to find the appropriate configuration over a wide range [3, 10]. In this work, we evaluate the single-core performance of an experimental Triton CPU backend on RVV, using a selection of AI inference and general computing benchmarks. To accelerate auto-tuning to find the appropriate configuration using some heuristics from experiments, we analyze the results to understand the impact of Triton's tiling on the RVV architecture. Because not every RVV processor is equipped with scratchpad memory, we have not included the use of scratchpad memory in our performance evaluation.

The remainder of this paper is organized as follows. Section 2 describes the motivation for this work. Section 3 presents the primary methodology. Section 4 details the experimental environments and results, then provides a corresponding analysis. Finally, Section 5 summarizes our findings and discusses future work.

Table 1: Analogy between CPUs and GPUs

Conceptions	GPUs	CPUs
Execution Unit	Warp	Vector instructions
Registers	Warp registers	Vector registers
Local Memory	Shared memory	Scratchpad memory
Cache	L1/L2 cache	L1/L2 cache

2 MOTIVATION

```
def matmul_kernel(a_ptr, b_ptr, c_ptr,
        M, N, K,
        stride_am, stride_ak,
        stride_bk, stride_bn,
        stride_cm, stride_cn,
        BLOCK_SIZE_M: tl.constexpr,
        BLOCK_SIZE_N: tl.constexpr,
        BLOCK_SIZE_K: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    pid_m = pid // num_pid_n
    pid_n = pid % num_pid_n
    block_offset_m = pid_m * BLOCK_SIZE_M
    block_offset_n = pid_n * BLOCK_SIZE_N
    a_tile_ptr = tl.make_block_ptr(
                    base=a_ptr, shape=(M, K),
                    strides=(stride_am, stride_ak),
                    offsets=(block_offset_m, 0),
                    block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_K), order=(1, 0))
    b_tile_ptr = tl.make_block_ptr(
                    base=b_ptr, shape=(K, N),
                    strides=(stride_bk, stride_bn),
                    offsets=(0, block_offset_n),
                    block_shape=(BLOCK_SIZE_K, BLOCK_SIZE_N), order=(1, 0))
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_tile_ptr)
        b = tl.load(b_tile_ptr)

        accumulator = tl.dot(a, b, accumulator, out_dtype=tl.float32)
        a_tile_ptr = tl.advance(a_tile_ptr, [0, BLOCK_SIZE_K])
        b_tile_ptr = tl.advance(b_tile_ptr, [BLOCK_SIZE_K, 0])
    c = accumulator
    c_tile_ptr = tl.make_block_ptr(
                    base=c_ptr, shape=(M, N),
                    strides=(stride_cm, stride_cn),
                    offsets=(block_offset_m, block_offset_n),
                    block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_N), order=(1, 0))
    tl.store(c_tile_ptr, c)
```

Listing 1: A simple matmul example written with Triton

This research is rooted in an analogy between Triton's established operational model on GPUs and its application to RVV in single core, shown in Table 1. Triton's effectiveness stems from its tiling capabilities, which partition large computational problems into smaller, manageable blocks. This strategy is fundamental to optimizing data locality and exposing parallelism. On a GPU, Triton maps a grid of program instances to thread blocks, and each thread processes a tile of data in warp registers. A similar concept is shown in a vector-enabled single core: the data parallelism execution by GPU warps is analogous to the data processing achieved by RVV's wide vector registers performing SIMD operations. The strategic use of GPU shared memory to keep data close to the compute units is similar to the goal of keeping data tiles close to cores within the scratchpad memory to minimize latency.

This architectural parallel suggests that Triton's high-level abstractions for data movement and parallel execution could make

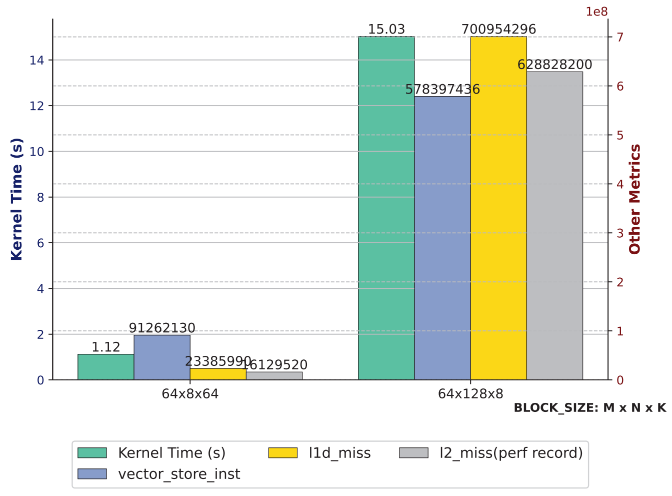


Figure 1: Performance comparison between different `BLOCK_SIZE`s

it an ideal programming model for heterogeneous systems that include vector-capable CPUs. However, directly porting GPU-tuned Triton code to CPUs causes performance issues due to architectural differences. GPUs, with their massive parallelism and large register files, benefit from large `BLOCK_SIZE`s. In contrast, CPUs have far fewer registers. Using a large `BLOCK_SIZE` on a CPU leads to performance degradation from register spilling and inefficient cache use.

Figure 1 shows the significant performance difference of kernel `matmul` between different block configurations on RVV (VLEN = 256 bits), and the listing 1 is the corresponding Triton source code. The large `BLOCK_SIZE` would lead to heavy vector register spilling because the length of the total element exceeds the capacity of vector registers.

Given this critical difference in architectural resources, adapting Triton for efficient CPU execution within a heterogeneous computing context requires a foundational understanding of how its parameter configurations behave on this new target. Therefore, this paper presents an empirical investigation into the performance impact of varying `BLOCK_SIZE` on RVV. Based on it, we can analyze the relationship between tile dimensions, cache utilization, and vector processing efficiency. And these experiences and heuristics can accelerate auto-tuning to find the appropriate parameters quickly.

3 METHODOLOGY

As there is no official version of the Triton CPU for the RISC-V backend, we build the Triton CPU on X86 (host) and use cross-compilation to obtain almost equivalent ELF files for RISC-V (remote) with RVV. To have a complete observation of the impact of `BLOCK_SIZE`, we tune each `BLOCK_SIZE` parameter in a wide range. Due to the limitation of cross-compilation, we modified the original auto-tuner of Triton to produce the corresponding LLVM IR for each kernel function block configuration. Each IR is then used for cross-compilation to RISC-V. The whole procedure includes the following steps:

Table 2: Kernel Specifications

Kernel Name	Dimension and Type
Matrix Multiplication	$M = 1024$, $K = 1024$, $N = 1024$ **Data type**: float32
Attention	batch_size = 4, context_length = 1024 #attention_head = 32 Head dimension = 64 **Data type**: float32
Jacobi-2D	$M = 1026$, $N = 1026$ **Data type**: float32

(1) Set the range for each `BLOCK_SIZE`, then run auto-tuner on host to get necessary IR files;
(2) Cross-compile each IR and launcher code written with C++, which would call kernels, and link them together to get a final ELF with RVV or other Extension (e.g. zicbop for prefetch) to support the same operations as the host fully;
(3) Run all ELFs with perf on remote to collect highly related performance metrics. And each kernel would be run 5 times.

4 EXPERIMENT

This section outlines the experimental benchmarks and the overall environment (i.e. software configurations and hardware specifications), which is used to evaluate the kernel performance, as well as performance illustration on benchmarks, along with the corresponding analysis.

4.1 Benchmarks

We selected several currently popular AI inference kernels[8] as our benchmark measurements. The arguments, as well as other basic information for each kernel, are shown in Table 2. Additionally, we incorporate Jacobi-2D to enhance the test set, and its input is wrapped with a halo of zeros. All kernels are public[1] and are completed with the Triton CPU compiler.

4.2 Experiment Environment

Clang 20.1.0 and LLVM 20.1.0 are used as the compilation infrastructure targeting x86_64-unknown-linux-gnu and riscv64-unknown-linux-gnu. Detailed information is shown in Table 3.

To support the necessary operations in Attention, the mathematical library SLEEF was built from source with native and RISC-V 64-bit architecture support, ensuring comprehensive compatibility across different target platforms.

The remote RISC-V platform information is shown in Table 4. In this RISC-V machine, each L1 cache is private, while four cores share the same L2 cache.

4.3 Experiment Results

We commence the performance illustration by exploring the defined tiling parameter searching space for each selected kernel, then

[1]Repository: https://github.com/liuyh-beep/Triton_RVV/tree/main/benchmark/src/triton

Table 3: Experiment Environment

Parameters	Value
Compiler	Clang 20.1.0
Operating System	Ubuntu 22.04.5 LTS (X86) Ubuntu 24.04.2 LTS (RISC-V)
Triton version	3.3.0
Compile options	- -target=riscv64-unknown-linux-gnu - -sysroot={RVV_GNU_DIR}/sysroot - -gcc-toolchain={RVV_GNU_DIR} -O2 -fPIC -fuse-ld=lld -fveclib=SLEEF -lm -L {SLEEF_RVV}/lib -march=rv64gcv_zvl256b_zicbop -mllvm -force-tail-folding-style= data-with-evl -mllvm -prefer-predicate-over- epilogue=predicate-dont-vectorize -mabi=lp64d

Table 4: RISC-V CPU information

Item	Info
CPU Model	Spacemit(R) X60 CPU max GHz: 1.6
VLEN	256 bits
Cache Line Size	64 B
L1 D/I Cache	32 KiB per instance direct-mapping
L2 Cache	512 KiB per instance 16-way associative

present a comprehensive performance analysis with highly related metrics.

4.3.1 Matrix Multiplication (matmul). In `matmul`, the configuration of each `BLOCK_SIZE` follows:

(1) `BLOCK_SIZE_M` $\in [4, 8, 16, 32, 64]$;
(2) `BLOCK_SIZE_N` $\in [8, 16, 32, 64]$;
(3) `BLOCK_SIZE_K` $\in [8, 16, 32, 64]$.

`matmul` is performed using Triton's built-in operation `tl.dot`, which is an operation of Triton's dialect and optimized according to host CPU features. Under our host environment, this operation would be fed to `triton-cpu-convert-dot-to-fma`, and this pass would convert `tl.dot` with the algorithm 1.

Figure 2 shows the correlation between execution time and the number of vector memory instructions, as well as cache miss count. In each subfigure, the bars are clustered by `BLOCK_SIZE` specified with the x-axis label, and within each bar cluster, another `BLOCK_SIZE` parameter is increasing. In Figure 2a, the data is sorted in ascending order of L2 miss count. Obviously, the performance is dominated by L1 and L2 cache efficiency. To make the rules more straightforward, Figure 2b and Figure 2c respectively illustrate the changes in vector load and store instruction counts and L1/L2 cache misses over different `BLOCK_SIZE`. There is an apparent negative correlation between cache miss and `BLOCK_SIZE_N`, while the relationship between vector register spill and `BLOCK_SIZE_M` or `BLOCK_SIZE_N` is complicated.

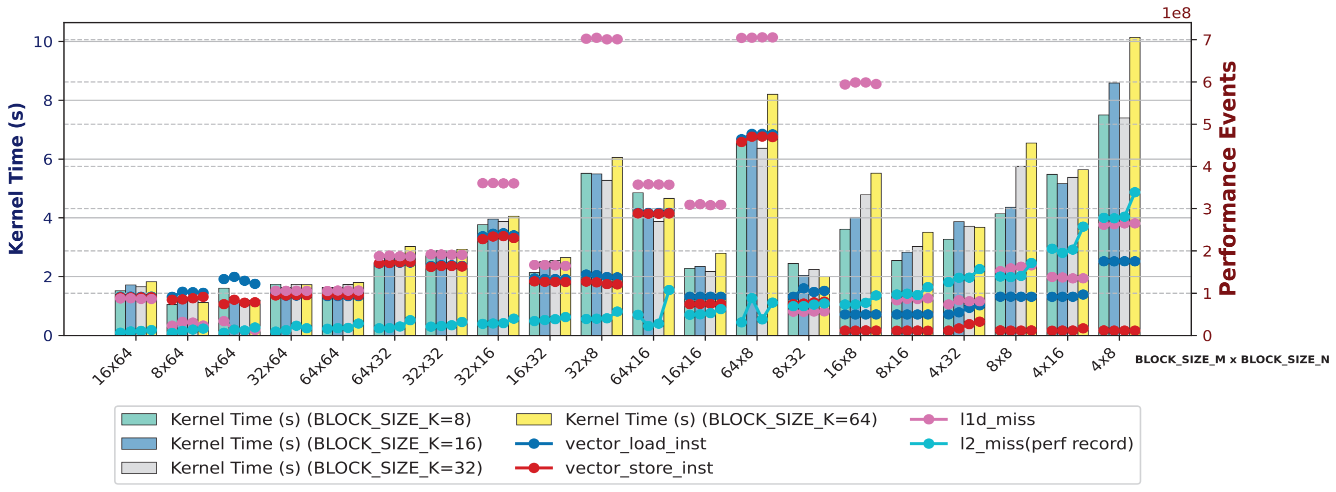
Algorithm 1: `triton-cpu-convert-dot-to-fma` pass

```
Data: $lhsBuf$ is a block of input matrix A, with size of
      BLOCK_SIZE_M × BLOCK_SIZE_K, while $rhsBuf$ is a
      block of input matrix B, with size of
      BLOCK_SIZE_K × BLOCK_SIZE_N. $IsTransposed$ is
      true if matrix A needs to be transposed.
Result: $accVecs$ would store the result of dot operation.
1  $nextRhsVec \leftarrow \text{loadRow}(rhsBuf, 0)$;
2  for $k = 0$ to BLOCK_SIZE_K − 1 do
3      $rhsVec \leftarrow nextRhsVec$;
4      if $k \neq$ BLOCK_SIZE_K − 1 then
5          $nextRhsVec \leftarrow \text{loadRow}(rhsBuf, k + 1)$;
6      end
7      prefetch the (k + BLOCK_SIZE_K)-th row of $rhsBuf$;
8      $nextLhsBroadcasted \leftarrow \text{broadcastElem}(lhsBuf[0][k])$;
9      for $m = 0$ to BLOCK_SIZE_M − 1 do
10         $lhsBroadcasted \leftarrow nextLhsBroadcasted$;
11         if $m \neq$ BLOCK_SIZE_M − 1 then
12             $nextLhsBroadcasted \leftarrow$
                 $\text{broadcastElem}(lhsBuf[m + 1][k])$;
13         end
14         if $(IsTransposed \wedge m \bmod 8 =$
             $0) \vee (\neg IsTransposed \wedge k \bmod 8 = 0)$ then
15             prefetch($lhsBuf[m][k +$ BLOCK_SIZE_K$]$);
16         end
17         $accVecs[m] \leftarrow$
             $\text{FMA}(rhsVec, lhsBroadcasted, accVecs[m])$;
18     end
19 end
```

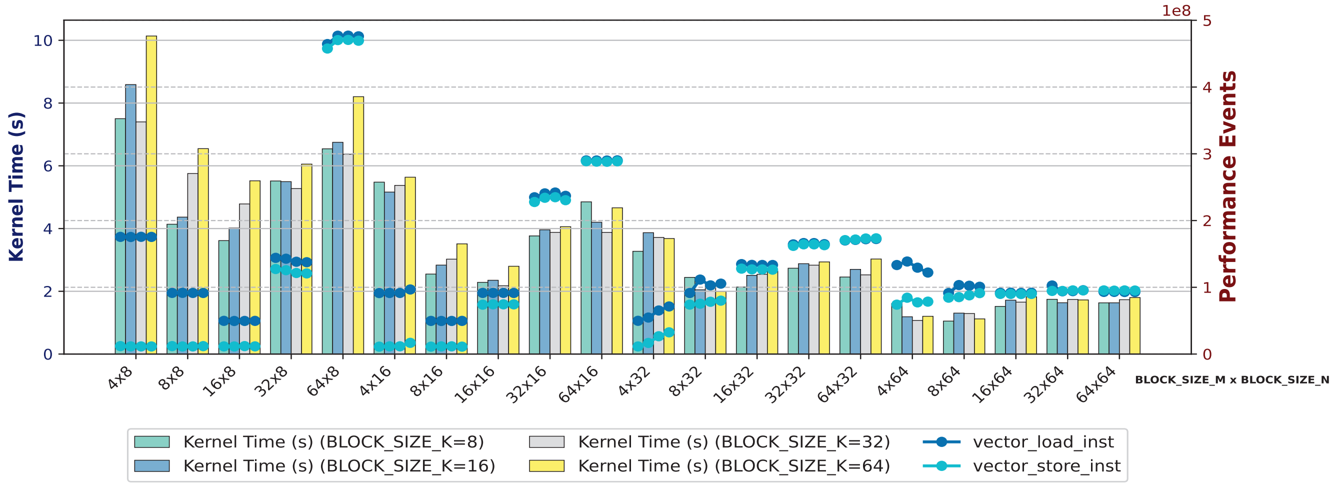
To quantify the impact of `BLOCK_SIZE` on vector load and store instruction counts and cache usage, we attempt to derive some equations to approximate it. From the process of `triton-cpu-convert-dot-to-fma`, we can find that, ideally, within accessing a column of *lhsBuf*, the vector registers would be occupied by two rows of *rhsBuf* plus all rows in *accVecs*. However, the actual number of active vector registers may be larger than this because the code scheduling shown in different configurations is not always identical, which could cause extra register spills. Therefore, we built the following formulas to present the number of vector store instructions generated from spilling:

$$SPILL_{col} = \texttt{BLOCK\_SIZE\_M} - \frac{32 \cdot VLEN}{\texttt{BLOCK\_SIZE\_N} \cdot SEW} + \mu \quad (1)$$

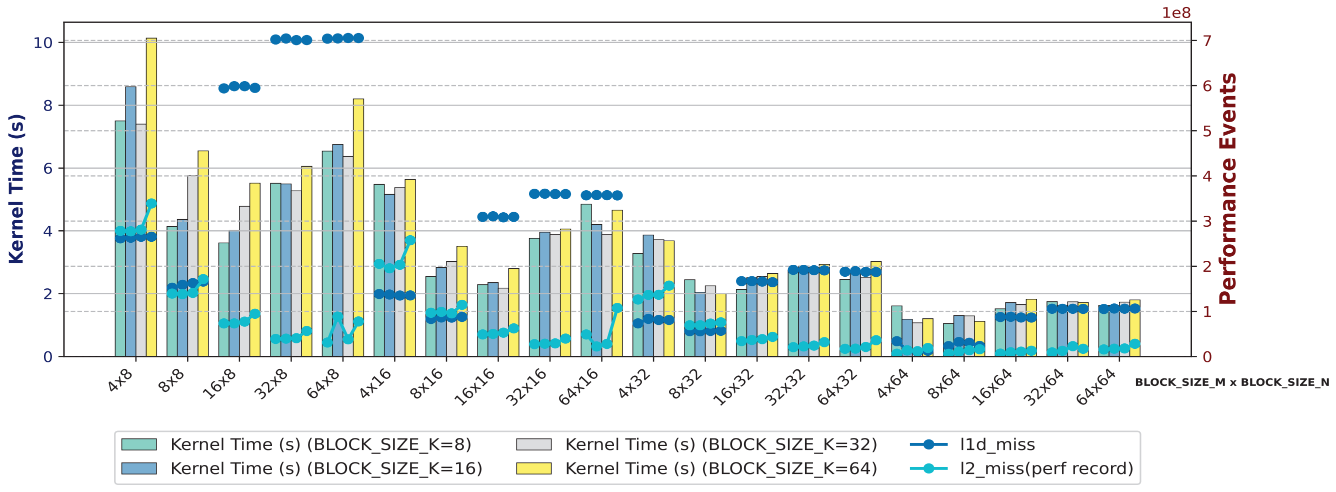
$$PB = \frac{M \cdot N}{\texttt{BLOCK\_SIZE\_M} \cdot \texttt{BLOCK\_SIZE\_N}} \quad (2)$$



(a) Performance overview with crucial metrics



(b) Vector load/store instruction counts changing



(c) L1 and L2 miss changing

Figure 2: Correlation between kernel execution time and related metrics (matmul)

$$SPILL = SPILL_{col} \cdot K \cdot PB \tag{3}$$

where $SPILL_{col}$ is the count of vector spilling instructions incurred during accessing a column of *lhsBuf*, and PB is the total number of program blocks, $VLEN$ is the bit length of a single RISC-V physical vector register, and SEW represents the width of the currently selected element within the vector register. Then $SPILL$ presents the total counts of vector spill instructions during the whole runtime, while $\mu \in [3, 5]$ is an artifact term to represent the extra vector pressure, and its range is derived from heuristics. We may conclude that, when $SPILL_{col}$ is larger than 0, there would be extra vector register spilling.

For vector reload, the instruction count can be expressed using similar equations because the number of reloads and spills is similar. However, if we want to present the total counts of vector load instructions, additional vector load instructions would be involved, as they are used to load input data into *rhsBuf*. We have

$$VLd_{rhsBuf} = K \cdot \mathrm{PB} \tag{4}$$

$$VLd = VLd_{rhsBuf} + SPILL \tag{5}$$

where VLd_{rhsBuf} is the count of necessary vector loads that occurred during the whole runtime, and VLd is the total number of vector load instructions, including reload instructions. Obviously, the BLOCK_SIZE_M would decide the number of active vectors, while LMUL is controlled by BLOCK_SIZE_N, which in turn changes *vl*.

On the other hand, we use the following model to estimate cache miss count:

$$MISS_{rhsBuf} = K \tag{6}$$

$$MISS_{lhsBuf} = \texttt{BLOCK\_SIZE\_M} \cdot \frac{K}{ELE_{CL}} \cdot f(\texttt{BLOCK\_SIZE\_M}) \tag{7}$$

$$MISS = (MISS_{lhsBuf} + MISS_{rhsBuf}) \cdot PB \tag{8}$$

where $\frac{K}{ELE_{CL}}$ shows the number of cache lines occupied by a single row of the matrix. And f(BLOCK_SIZE_M) is a function about BLOCK_SIZE_M, which is derived from possible cache conflict miss when accessing across many rows, as it is easy to incur conflict miss given our L1 cache is directly mapped. $MISS_{rhsBuf}$ represents the cache miss that happened in accessing all rows of *rhsBuf* within a program block. Although a larger BLOCK_SIZE_N would lead to accessing across cache lines, a longer vector means more contiguous data, and the hardware prefetcher would automatically detect the refill signal to prefetch the data for subsequent use.

Collectively, larger BLOCK_SIZE_N is beneficial for cache performance, while it also would increase vector register spilling due to longer vectors. And a smaller BLOCK_SIZE_M is better to avoid possible conflict misses. Our model demonstrates a reasonably good fit to the experimental data. The final performance is the comprehensive consideration of vector load instruction counts and cache misses.

4.3.2 Attention. For Attention, we use the same BLOCK_SIZE configurations as matmul.

According to the formulas of standard attention[8], we write three kernels, two matrix multiplication plus a softmax, then combine them to perform the complete function. The reason for using two different matmuls is that we can use different symbols in perf for sample analysis. Another difference from a single matmul is that we use BLOCK_SIZE_N and BLOCK_SIZE_K to control the vector length respectively. For softmax, the BLOCK_SIZE is the same as the context length for all configurations.

Figure 3 presents the key performance metrics. The primary source of overhead is identified as the two sequential matrix multiplications. The optimal configuration is found to be BLOCK_SIZE_M = 8, BLOCK_SIZE_K = 16, and BLOCK_SIZE_N = 16. We hypothesize that this configuration strikes a balance that mitigates the inefficiencies of hardware prefetching and minimizes the count of vector load instructions. The reason is, shown as Figure 3b and Figure 3a, this best performing configuration exhibits only a 50%–60% difference in cache performance compared to the configuration with the lowest cache miss counts (e.g. BLOCK_SIZE_M = 8, BLOCK_SIZE_K = 64, BLOCK_SIZE_N = 64) and an approximately 60% degradation in vector load instruction count. Compared to optimal configurations in a single matmul kernel, the chosen configuration shows a 12× and 3× difference in L1 and L2 cache performance, respectively. At the same time, the degradation in vector load instruction counts is significantly smaller (40%–50%).

As discussed in Section 4.3.1, the final performance is co-determined by vector register spilling and cache performance. While larger BLOCK_SIZE parameters are generally advantageous for hardware cache prefetching, our results suggest that when these dimensions approach the size of the original matrix, the prefetcher may prematurely move substantial useless data. This inefficiency can negate the potential benefits of cache prefetching.

4.3.3 Jacobi-2D. The BLOCK_SIZE configuration list of Jacobi-2D is shown as:

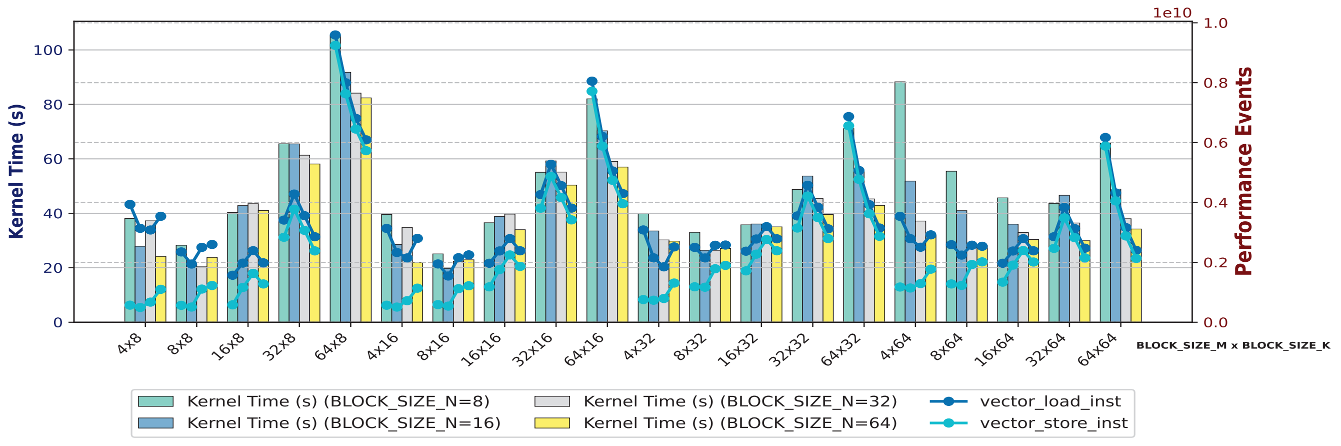
(1) BLOCK_SIZE_M $\in [4, 8, 16, 32, 64]$;
(2) BLOCK_SIZE_N $\in [4, 8, 16, 32, 64]$.

Algorithm 2: Jacobi-2D Triton Method

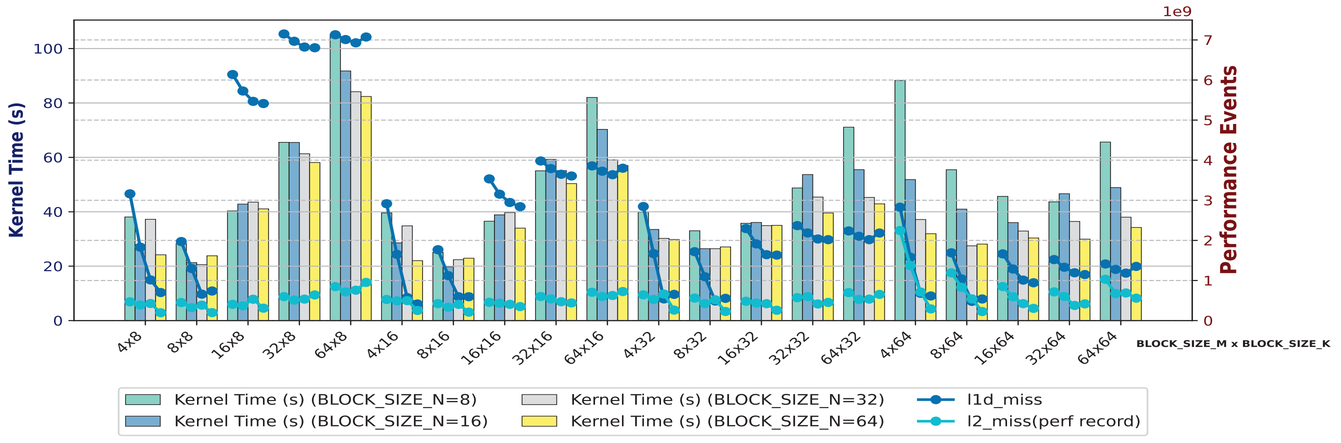
```
Data: The pointer to original input matrix $A$ of size $M \times N$.
Result: A updated block of result matrix $B$.
1 BLOCK_SIZE = [BLOCK_SIZE_M, BLOCK_SIZE_N];
2 offset$_{up}$ = $[0, 1]$;
3 $A_{up}$ ← tl.make_block_ptr($A$, BLOCK_SIZE, offset$_{up}$)
4 $A_{mid}$ ← tl.advance($A_{up}$, $[1, 0]$);
5 $A_{down}$ ← tl.advance($A_{mid}$, $[1, 0]$);
6 $A_{left}$ ← tl.advance($A_{mid}$, $[0, -1]$);
7 $A_{right}$ ← tl.advance($A_{mid}$, $[0, 1]$);
8 $B \leftarrow 0.2 \times (A_{up} + A_{mid} + A_{down} + A_{left} + A_{right})$;
```

We implement the Jacobi-2D kernel using Algorithm 2 at the block level. The total number of program blocks follows Equation (2). Since the input data is zero-padded, offset$_{up}$ defines the pointer offsets along row and column dimensions. Positive offsets prevent redundant loading of data that overlap between adjacent blocks. Currently, Triton CPU lacks native element-shift operations, requiring extra loads for left/right shifts.

Figure 4 shows that cache misses dominate performance. Similar to matmul, larger BLOCK_SIZE_N and smaller BLOCK_SIZE_M



(a) Vector load/store instruction counts changing



(b) L1 and L2 miss changing

Figure 3: Correlation between kernel execution time and related metrics (Attention)

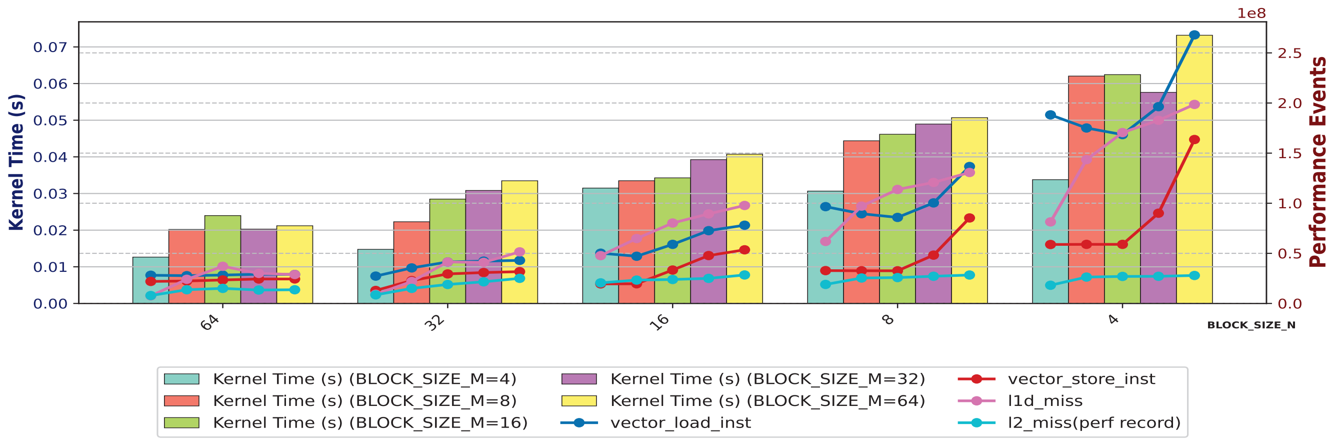
improve cache efficiency by leveraging longer vectors, which reduce the number of program blocks and better utilize hardware prefetching. However, longer vectors also increase register spilling and may raise L1 conflict misses.

5 CONCLUSIONS

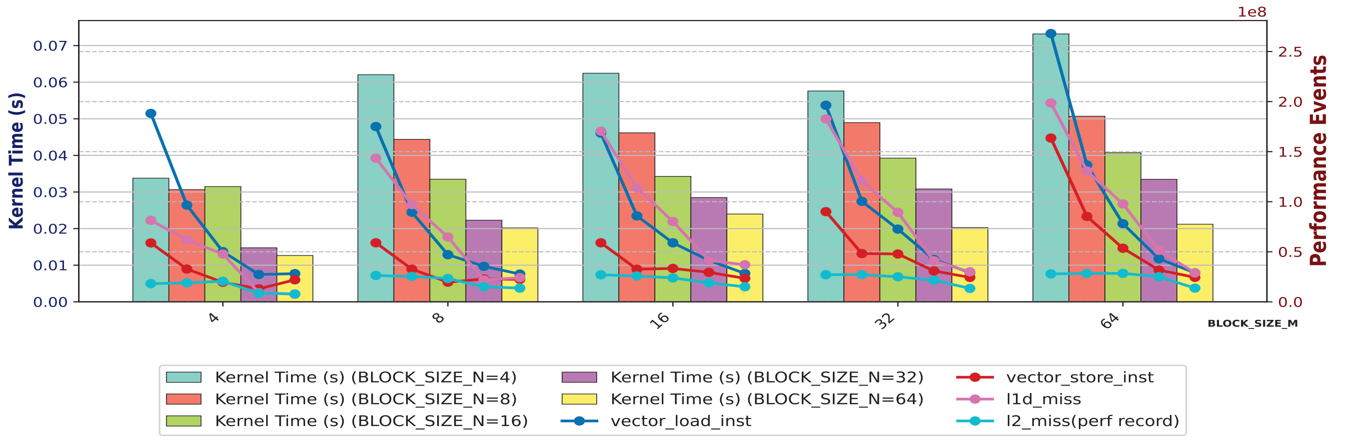
This work investigated the issue of performance portability of Triton, an efficient GPU-centric tile-based programming model. The tile-based programming enables the Triton compiler to allocate blocks more effectively in the GPU shared memory and manage warp execution of thread blocks. Such tile-based programming is also perceived as ideal for CPUs with SIMD/Vector capability. Tile-based computing is known for being cache-friendly, and also suitable for vectorization and vector register allocation. While the architectural analogies between GPUs and CPUs seem to suggest that Triton is a promising programming model for AIPC edges, a naive translation using GPU-centric parameters fails to achieve high performance on the CPU side. The core challenge lies in the adaptive selection of the tiling parameters, which has a profound and contrasting impact on vector register utilization and cache efficiency in RVV architectures.

Our empirical analysis of key computational kernels led to several key findings. The main performance is co-determined by cache misses and vector register spilling; thus, a balance must be struck between these two factors. For contiguous data access (stride-1 references), a longer vector is beneficial for cache performance due to better spatial locality and effective hardware cache prefetching. It also enables the compiler to exploit the LMUL to group a few shorter vector registers into a longer register. However, an extremely long vector, which is close to the scale of the data, is detrimental to hardware prefetching. In general, long vectors would incur more vector register spilling. Additionally, conflict misses may also occur due to multiple rows of data mapping to the same cache set when the vector loads are lengthy. Based on these heuristics, we can accelerate the auto-tuning processes, thereby avoiding multi-day compilation due to fruitless searches.

For future work, we will expand our evaluation to include the use of scratchpad memory, exploring the performance potential and



(a) Performance overview with crucial metrics



(b) Vector load/store instruction counts and L1/L2 cache miss changing

Figure 4: Correlation between kernel execution time and related metrics (Jacobi-2D)

issues of explicitly managed on-chip memory. Then, develop the Triton CPU to support the RVV backend, enhancing the utilization of both scratchpad memory and vector registers.

ACKNOWLEDGMENTS

This work was supported by Pengcheng Peacock Plan Grant 2025TC-0017, Grant 2023TA0033, and Huawei Technologies Co., Ltd, China under contract No., TC20220913039.

REFERENCES

[1] Jiasi Chen and Xukan Ran. 2019. Deep Learning With Edge Computing: A Review. *Proc. IEEE* 107, 8 (2019), 1655–1674. https://doi.org/10.1109/JPROC.2019.2921977
[2] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. 2020. Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence. *IEEE Internet of Things Journal* 7, 8 (2020), 7457–7469. https://doi.org/10.1109/JIOT.2020.2984887
[3] Thomas Falch and Anne Elster. 2016. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency and Computation: Practice and Experience* 29 (12 2016). https://doi.org/10.1002/cpe.4029
[4] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices. In *Proceedings of the Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 209–221. https://doi.org/10.1145/3498361.3538932
[5] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. 2022. Bringing AI to edge: From deep learning's perspective. *Neurocomputing* 485 (2022), 297–320. https://doi.org/10.1016/j.neucom.2021.04.141
[6] RISC-V International. 2021. RISC-V Vector Extension Version 1.0 (RVV). (2021). https://github.com/riscvarchive/riscv-v-spec
[7] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973
[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
[9] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. 2025. T-MAC: CPU Renaissance via Table Lookup for Low-Bit LLM Deployment on Edge. In *Proceedings of the European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 278–292. https://doi.org/10.1145/3689031.3696099
[10] Lianmin Zheng and Chris Hoge. 2020. AutoTVM. (2020). https://tvm.apache.org