Triton kernel performance on RISC-V CPU

Contents

Introduction to Triton

Triton is an open-source, Python-based Domain-Specific Language (DSL) developed by OpenAI to simplify the writing of high-performance GPGPU code. It was first introduced in 2019 by Tillett et al. in a paper titled "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations" at Harvard University. The core design concepts of Triton include block-level parallel programming, LLVM-based intermediate representation, and multi-level optimization processes. Since 2020, OpenAI has taken over the development of Triton, releasing version 3.0 after years of iterations. Several open-source projects, including PyTorch, Unsloth, and FlagGems, have adopted Triton to develop part or all of their kernels.

Initially, Triton only supported NVIDIA consumer-grade GPUs, but it gradually expanded to industrial-grade GPUs like A100 and H100. Other AI chip vendors, including both domestic and international GPGPU and DSA accelerator hardware platforms, have also begun supporting Triton. At Triton Conference [1][2] 2024 in Silicon Valley in September, chip manufacturers like Intel, AMD, Qualcomm, Nvidia, together with solution providers like Microsoft and AWS shared their progress and performance results with Triton, showcasing its broad application across various hardware platforms. Furthermore, Triton has also begun exploring CPU adaptation to further enhance hardware compatibility.

At the hardware level, Triton targets Cooperative Thread Arrays (CTA), while at the software level, it focuses on block-level parallel programming. Compared to higher-level frameworks like PyTorch, Triton is more focused on the detailed implementation of computational operations. Developers can enjoy more freedom in manipulating tile-level data read/write operations, execute computational primitives, and define thread block partitioning methods. Triton hides the scheduling details at and below thread block level, with the compiler automatically managing shared memory, thread parallelism, memory coalescing and tensor layout, thereby reducing the difficulty of parallel programming and improving development efficiency. Developers only need to understand basic parallel principles and focus on algorithms and their implementations, which in turn help to write high-performance kernels efficiently.

Although Triton compromises some programming flexibility, it still achieves comparable performance to CUDA through multi-level compilation and optimization. For example, in the official tutorial, the Triton implementation of matrix multiplication achieves performance on par with cuBLAS under specific testing conditions, fully utilizing the hardware's computational capacity and showcasing outstanding performance [3].

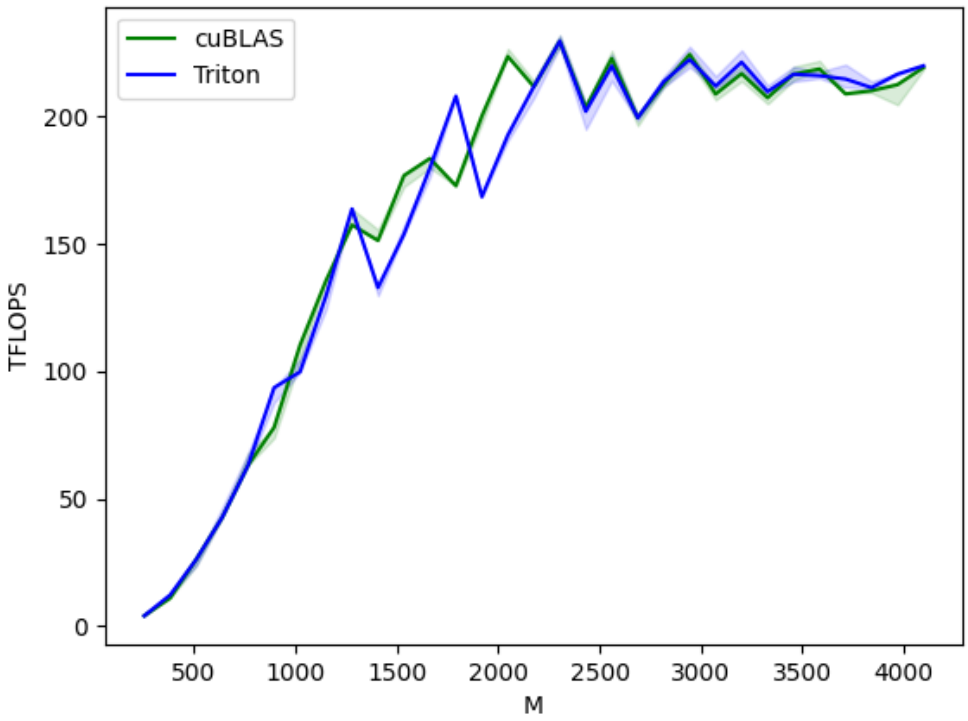

Fig1: Triton vs cuBLAS on Matmul Kernel

Triton on CPU

Background

There are several Triton for CPU projects including the official Triton-CPU [5] from OpenAI and Microsoft's Triton-Shared [7], along with other efforts such as Cambricon's Triton-Linalg[8]. However, Triton-Linalg has only completed Triton language frontend integration, with backend CPU support still under development. Triton-Shared integrates MLIR's linalg dialect into its Triton's Intermediate Representation (IR) before lowering it to LLVM IR dialect. This pipeline lacks target dependent optimizations and remains experimental with issues on performance. Triton-Shared is primarily used to explore the possibility of expanding Triton support on CPU platform.

In contrast, Triton-CPU integrates MLIR's vector dialect into Triton's high-level IR to effectively vectorize the code. Additionally, multithreading is supported with auto generated OpenMP C code from a Python script, leading to a significant boost in parallel computation performance on CPU. In Triton community meetup[4] in August 2024, Triton-CPU demonstrated performance that was comparable to or even better than Torch on CPU, highlighting its potential in efficiently utilizing multi-core resources and instruction-level parallelism.

RISC-V, an open Instruction Set Architecture (ISA), has shown immense potential in the AI chip domain. Its flexibility, scalability, and support for customization make it an ideal choice for designing efficient AI processors. The scalable vector extension instruction set (RVV) has shown powerful vector processing capabilities, enabling efficient execution of parallel computing tasks, which is a key requirement for AI algorithms such as matrix multiplication and vector operations. Additionally, RISC-V allows developers to add custom instructions to accelerate specific AI computational tasks. For instance, specialized instructions can be designed for common neural network operations like convolution and activation functions to improve execution efficiency.

The Python language features of Triton and its higher-level programming model abstractions can significantly reduce the costs for chip manufacturers in developing and maintaining kernel libraries. Leveraging the multi-level intermediate representation (IR) mechanism provided by the MLIR software stack further enhances the adaptability of Triton's kernel libraries to the RISC-V architecture, improving compatibility among different RISC-V AI chips. Additionally, the highly customizable nature of RISC-V AI chips allows manufacturers to design specialized instructions tailored to specific kernels, thereby fully unleashing the potential of the kernel libraries. Therefore, in-depth research on Triton kernel performance on existing RISC-V AI chips, and optimizing for major performance bottlenecks, is critical for advancing the development of Triton, MLIR, and the RISC-V ecosystem. This not only promotes the collaborative development of various RISC-V AI chips but also contributes to the prosperous development of the entire ecosystem.

Goals

The Triton-CPU Meetup only showcased the performance of some kernels on the x86 platform, while the performance of Triton kernels on the RISC-V platform has yet to be fully explored. This study seeks to compare the performance of kernels written in OpenAI's Triton language and C language with the same algorithm, focusing solely on the impact of compiler optimizations. By deeply analyzing the differences in the assembly codes, this research aims to identify further optimization opportunities for the Triton-CPU compiler, providing guidance for subsequent performance improvements.

Development Environment

We use RISC-V cross compilers on x86 platform to generate RISC-V executables. Specifically, we generate LLVM IR using Triton-CPU, then cross-compile it into assembly code for RISC-V target. The final RISC-V binary runs on SpacemiT's K1 development board, which includes 8 dual-issue RISC-V CPUs with 256bit RVV 1.0 support. This setup is ideal for testing kernels' performance with the hardware's parallel computing capabilities.

For the compiler, the C code was compiled using RISC-V **GCC 15.0.0** and our **ZCC 3.2.4** (an in-house optimized LLVM-based compiler). The LLVM IR for Triton kernels was generated using the Triton-CPU project's built-in compiler and then compiled

into RISC-V assembly code using ZCC. The compilation was consistently done with `-O3` optimization level to ensure maximum performance.

Kernel Implementation

We selected commonly used kernels in large language models, such as rope, matmul, softmax, layernorm, as well as image processing kernels like resize, warp, and correlation, for benchmarking. The kernel implementation can be found in Terapines's AI-Benchmark repository [9].

The implementation of these kernels uses standard algorithm. Although not every kernel is tuned to it's best algorithm implementation, efforts were made to ensure consistency between the C language and Triton language implementations. The vectorization and multithreading in the C language are primarily achieved through `#pragma` directives, without using any RISC-V intrinsics, relying entirely on the compiler's automatic optimization capabilities.

We strive to ensure a fair comparison of the compiler performances across different programming language implementations of the kernels in our design. This will also help us gain a solid baseline for subsequent optimizations.

Performance Analysis

On the same hardware platform with the same `-O3` optimization level, we tested the performance of both Triton and C kernels. For each kernel, the running time in single, four, and eight threads (T1, T4 and T8) was measured with different input shapes. To make comparison and presentation easier, the performance data for different input/output shapes were normalized relative to the kernel’s time complexity, with the final performance results displayed in GB/s (the higher the value, the better the performance).

RoPE

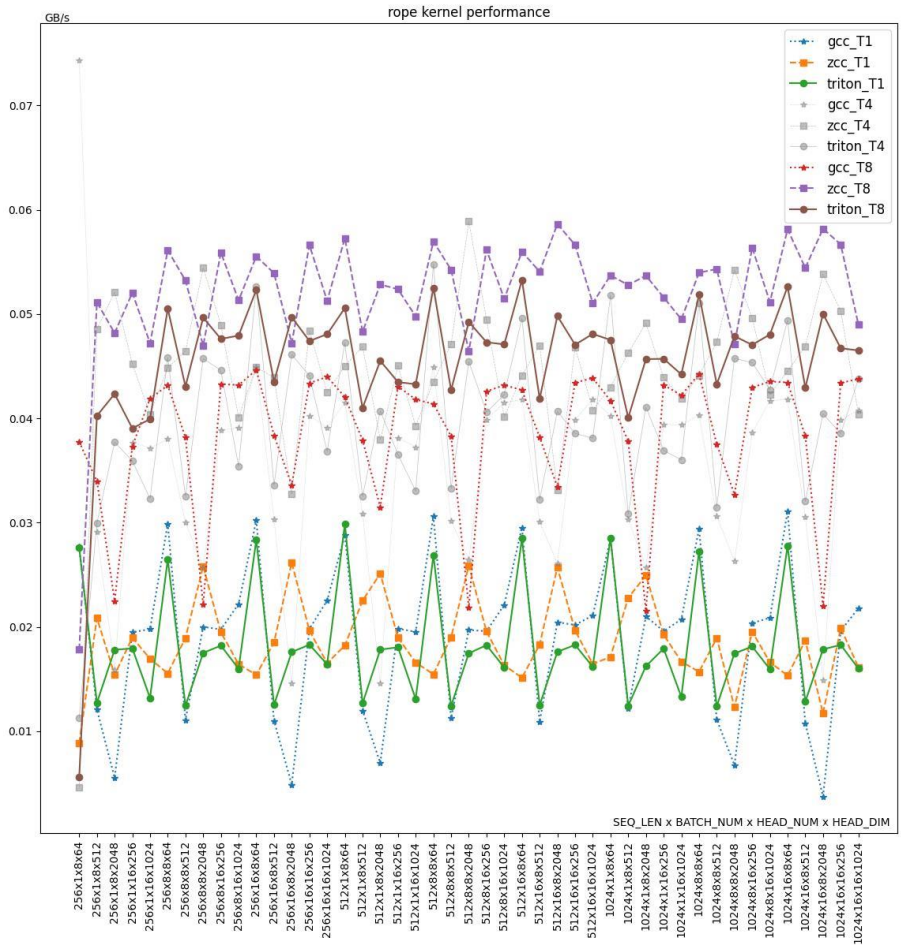

Fig2: rope kernel performance on different shape

Average performance on different compiler							
Average (GB/s)	GCC	ZCC	Triton	Triton/ZCC	Triton/GCC	(Triton/ZCC)/(TN/T1)	(Triton/GCC)/(TN/T1)
T1	0.01874	0.01871	0.01825	97.55%	97.40%		
T4	0.03489	0.04454	0.04022	90.31%	115.27%	103.91%	100.90%
T8	0.038	0.0529	0.0459	86.84%	118.65	100.95	100.62

	74	3	6		%	%	%
T4/T1	200.71%	263.82%	229.29%	86.91%	114.24%		
T8/T1	227.35%	311.65%	268.09%	86.02%	117.92%		

> Note (applicable to all following performance analysis tables):
>
> 1. The speedup of threads is calculated as the average of the ratios based on the original data (i.e., T4/T1 and T8/T1).
> 2. (Triton/ZCC)/(TN/T1) reflects the change in the performance gap between multi-threaded compilers relative to single-threaded ones, after removing the effect of average thread speedup.

In the case of single-thread (T1) runs, the performances of Triton, ZCC, and GCC are similar. However, in multi-threaded cases with four and eight threads (T4 and T8), ZCC shows a performance advantage of 10% to 13% over Triton, while Triton outperforms GCC by approximately 15%. We believe the primary reasons may include:

- Impact of multi-threading: The multi-threaded speedup ratio by different compilers varies greatly. We need further investigation to analyze the issues that affect kernel performance on multi-threaded environments.
- Triton has to generate more instructions inside the main loop in order to handle explicit maskload operations. This may result in slightly lower single-threaded performance compared to ZCC and GCC.

Example of Triton code reading data from the input array using a masked load in RoPE:

```
Python
x1 = tl.load(input_ptr + x1_offset, mask=mask, other=0.0)
x2 = tl.load(input_ptr + x2_offset, mask=mask, other=0.0)
```

Core Loop Assembly Code:

```
Assembly language
# GCC
.L8:
  vsetvli a5,a2,e32,m1,ta,ma
  vle32.v v3,0(a1)
  vle32.v v2,0(s9)
```

```
  vle32.v v4,0(a0)
  vle32.v v1,0(s8)
  slli  a4,a5,2
  sub a2,a2,a5
  add a0,a0,a4
  vfmul.vv  v2,v2,v3
  add a1,a1,a4
  add s8,s8,a4
  add s9,s9,a4
  vfmsub.vv v1,v4,v2
  vse32.v v1,0(a3)
  add a3,a3,a4
  vle32.v v2,0(s10)
  vle32.v v1,0(s11)
  add s10,s10,a4
  add s11,s11,a4
  vfmul.vv  v2,v2,v4
  vfmadd.vv v1,v3,v2
  vse32.v v1,0(s7)
  add s7,s7,a4
  bne a2,zero,.L8
```

```
Assembly language
# ZCC
.LBB1_14:
  vsetvli a5, a1, e32, m8, ta, ma
  vle32.v v8, (s1)
  vle32.v v16, (a4)
  vle32.v v0, (a2)
  vle32.v v24, (s0)
  slli  a0, a5, 2
  sub a1, a1, a5
  add s1, s1, a0
  add a4, a4, a0
  vfneg.v v0, v0
  vfmul.vv  v0, v16, v0
  vfmacc.vv v0, v8, v24
  vse32.v v0, (a3)
  add a3, a3, a0
  vle32.v v24, (a2)
  vle32.v v0, (s0)
  add a2, a2, a0
  add s0, s0, a0
```

```
  vfmul.vv  v8, v8, v24
  vfmadd.vv v0, v16, v8
  vse32.v v0, (s11)
  add s11, s11, a0
  bnez  a1, .LBB1_14
```

```
Assembly language
# Triton
.LBB0_2:
  vsetvli zero, zero, e32, m4, ta, mu
  vor.vx  v28, v8, t0
  addw  a1, a6, t0
  vmv4r.v v16, v12
  addw  t4, a7, t0
  vmv4r.v v20, v12
  addw  a2, t3, t0
  vmv4r.v v24, v12
  vmslt.vx  v0, v28, t5
  slli  a1, a1, 2
  add a1, a1, a4
  vle32.v v16, (a1), v0.t
  addw  a1, t2, t0
  vmv4r.v v28, v12
  slli  t4, t4, 2
  slli  t6, a2, 2
  slli  a1, a1, 2
  add t4, t4, a5
  vle32.v v20, (t4), v0.t
  add a2, a0, t6
  vle32.v v24, (a2), v0.t
  add a2, a0, a1
  vle32.v v28, (a2), v0.t
  addi  t1, t1, 32
  add t6, t6, a3
  add a1, a1, a3
  vfmul.vv  v4, v20, v28
  vfmul.vv  v28, v16, v28
  vfrsub.vf v4, v4, fa5
  vfmacc.vv v28, v20, v24
  vfmacc.vv v4, v24, v16
  vse32.v v4, (t6), v0.t
  vse32.v v28, (a1), v0.t
  addi  t0, t0, 32
```

```
  bltu  t1, t5, .LBB0_2
```

Matmul

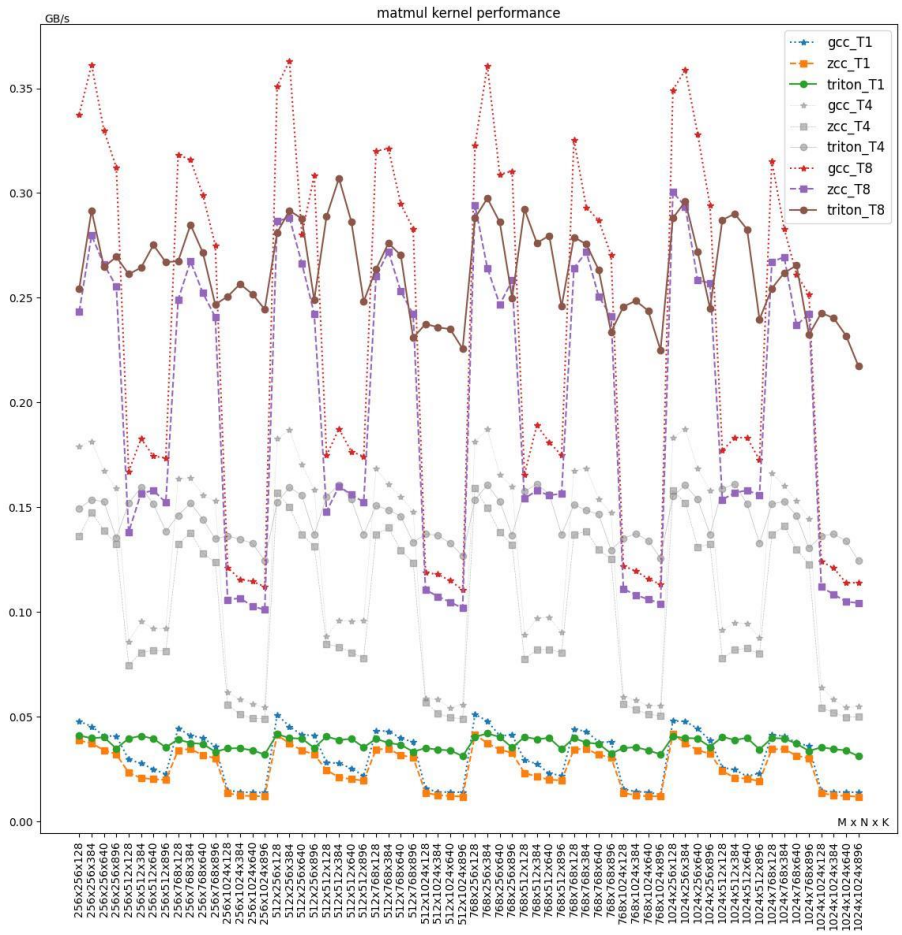

Fig3: matmul kernel performance on different shape

Average performance on different compiler							
Average(GB/s)	GCC	ZCC	Triton	Triton/ZCC	Triton/GCC	(Triton/ZCC)/(TN/T1)	(Triton/GCC)/(TN/T1)
T1	0.03113	0.02568	0.03716	144.71 %	119.35 %		
T4	0.12045	0.10183	0.14484	142.24 %	120.25 %	98.94 %	98.82%

T8	0.22966	0.19614	0.26273	133.95 %	114.40 %	99.12 %	98.88%
T4/T1	320.55 %	271.35 %	390.08 %	143.76 %	121.69 %		
T8/T1	611.60 %	523.58 %	707.58 %	135.14 %	115.69 %		

> Note: Both Triton and the C kernels use tile-based matrix multiplication implementation. But GCC and ZCC cannot automatically vectorize the outer K dimension after tiling M and N dimensions, resulting in significant differences in the generated assembly code.

In the current block size (16x16), Triton outperforms the C kernel compiled by GCC and ZCC, and this remains consistent across different shapes. The primary reasons for performance differences may include:

- Memory access pattern: GCC identifies strided loads and takes advantage of RVV's strided load instructions, while ZCC and Triton do not. Triton uses scalar loops for memory load (see: Vectorization of Discrete Memory Access).
- RVV Register grouping: ZCC uses larger register grouping (m4), which are fully utilized only in a 32x32 block, so the performance is not fully optimized under the current 16x16 block size. Triton adjusts its grouping size based on the block size after tiling, while GCC uses m1 as the grouping size.
- Compute operations: Since GCC generates strided loads, it can perform vector-vector operations, while ZCC is limited to scalar-vector computation. Triton loads the entire block into registers, unrolls the dot products (highlighted in the code snippet below), and fully utilizes vector instructions for computation. Register spills may occur during computation, but register spills can be avoided by adjusting block size.

In TTIR as shown in the code blocks below, for `tensor<16x8xf32>`, the `16x8 f32`elements can be loaded into 8 registers of 16 elements of e32 type using `m2` grouping, performing the multiplication and reduction directly with each row in `tensor<64x16xf32>`.

```
C
// C Matmul: Tiling on M, N loop dimension
#pragma omp parallel for collapse(2) schedule(static)
num_threads(max_threads.value())
  for (int i = 0; i < M; i += BLOCK_SIZE_M) {
    int i_end = std::min(M, i + BLOCK_SIZE_M);
```

```
    for (int j = 0; j < N; j += BLOCK_SIZE_N) {
      int j_end = std::min(N, j + BLOCK_SIZE_N);
#pragma omp simd
      for (int kk = 0; kk < K; ++kk) {
        for (int ii = i; ii < i_end; ++ii) {
          for (int jj = j; jj < j_end; ++jj) {
            arg2[ii * N + jj] += arg0[ii * K + kk] * arg1[kk * N +
jj];
          }
        }
      }
    }
  }
```

```
Python
# Triton Matmul
accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N),
dtype=tl.float32)
for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
    # Load the next block of A and B, generate a mask by checking
the K dimension.
    # If it is out of bounds, set it to 0.
    a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k *
BLOCK_SIZE_K, other=0.0)
    b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k *
BLOCK_SIZE_K, other=0.0)
    # We accumulate along the K dimension.
    accumulator += tl.dot(a, b)
    # Advance the ptrs to the next K block.
    a_ptrs += BLOCK_SIZE_K * stride_ak
    b_ptrs += BLOCK_SIZE_K * stride_bk
```

```
Assembly language
// Triton IR (TTIR)
%40:3 = scf.for %arg9 = %c0_i32 to %37 step %c1_i32
iter_args(%arg10 = %cst_0, %arg11 = %26, %arg12 = %35) ->
(tensor<64x8xf32>, tensor<64x16x!tt.ptr<f32>>,
tensor<16x8x!tt.ptr<f32>>)  : i32 {
    %57 = arith.muli %arg9, %c16_i32 : i32 loc(#loc31)
    %58 = arith.subi %arg5, %57 : i32 loc(#loc32)
    %59 = tt.splat %58 : i32 -> tensor<1x16xi32> loc(#loc33)
```

```
    %60 = arith.cmpi slt, %21, %59 : tensor<1x16xi32> loc(#loc33)
    %61 = tt.broadcast %60 : tensor<1x16xi1> -> tensor<64x16xi1>
loc(#loc34)
    %62 = tt.load %arg11, %61, %cst_2 : tensor<64x16x!tt.ptr<f32>>
loc(#loc34)
    %63 = tt.splat %58 : i32 -> tensor<16x1xi32> loc(#loc35)
    %64 = arith.cmpi slt, %27, %63 : tensor<16x1xi32> loc(#loc35)
    %65 = tt.broadcast %64 : tensor<16x1xi1> -> tensor<16x8xi1>
loc(#loc36)
    %66 = tt.load %arg12, %65, %cst_1 : tensor<16x8x!tt.ptr<f32>>
loc(#loc36)
    %67 = tt.dot %62, %66, %arg10, inputPrecision = tf32 :
tensor<64x16xf32> * tensor<16x8xf32> -> tensor<64x8xf32>
loc(#loc37)
    %68 = tt.addptr %arg11, %cst : tensor<64x16x!tt.ptr<f32>>,
tensor<64x16xi32> loc(#loc38)
    %69 = tt.addptr %arg12, %39 : tensor<16x8x!tt.ptr<f32>>,
tensor<16x8xi32> loc(#loc29)
    scf.yield %67, %68, %69 : tensor<64x8xf32>,
tensor<64x16x!tt.ptr<f32>>, tensor<16x8x!tt.ptr<f32>> loc(#loc39)
} loc(#loc30)
```

Core Loop Assembly Code

```
Assembly language
# ZCC
vsetvli a2, s0, e32, m4, ta, ma
vle32.v v8, (a4)
flw fa5, 0x0(a6)
vle32.v v12, (t1)
slli    s1, a2, 0x2
sub s0, s0, a2
add a4, a4, s1
vfmacc.vf   v12, fa5, v8
vse32.v v12, (t1)
add t1, t1, s1
bnez    s0, 0x5020 <_Z6matmulPfS_S_iii.omp_outlined.7+0x216>
```

```
Assembly language
# GCC
vsetvli  t4,zero,e32,m1,ta,ma
vlse32.v v3,(s7),zero
mv s1,s3
```

```
mv t4,a5
mv s11,s9

vsetvli  a3,s1,e32,m1,ta,ma
vle32.v  v1,(s6)
vle32.v  v2,(s11)
slli  s4,a3,0x2
sub   s1,s1,a3
add   s6,s6,s4
add   s11,s11,s4
vfmacc.vv   v1,v3,v2
vse32.v  v1,(t4)
add   t4,t4,s4
bnez  s1,1369e <_Z6matmulPfS_S_iii._omp_fn.1+0x152>

addiw s8,s8,1
add   a5,a5,t5
add   s7,s7,t6
bne   s8,t0,13662 <_Z6matmulPfS_S_iii._omp_fn.1+0x116>
```

```
Assembly language
# Triton
...
vsetivli    zero, 16, e32, m2, ta, ma
csrr    a2, vlenb
li  a3, 89
mul a2, a2, a3
add a2, a2, sp
addi    a2, a2, 2047
addi    a2, a2, 929
vl2r.v  v16, (a2)                       # Unknown-size Folded
Reload
vfmul.vv    v8, v16, v24
vfredusum.vs    v8, v8, v1
vfmul.vv    v10, v16, v22
vfredusum.vs    v9, v10, v1
vfmul.vv    v10, v16, v28
vfredusum.vs    v10, v10, v1
vfmul.vv    v12, v16, v2
vfredusum.vs    v11, v12, v1
vfmul.vv    v12, v16, v6
vfredusum.vs    v12, v12, v1
vfmul.vv    v14, v16, v30
```

```
vfredusum.vs    v13, v14, v1
vfmul.vv    v14, v16, v18
vfredusum.vs    v14, v14, v1
vfmul.vv    v16, v16, v26
vfredusum.vs    v15, v16, v1
vfmv.f.s    fa4, v8
vfmv.f.s    fa3, v9
vfmv.f.s    fa2, v10
vfmv.f.s    fa1, v11
vfmv.f.s    fa0, v12
vfmv.f.s    ft0, v13
vfmv.f.s    ft1, v14
vfmv.f.s    ft2, v15
vsetivli    zero, 8, e32, m1, ta, mu
vfmv.v.f    v8, fa4
vfslide1down.vf v8, v8, fa3
vfmv.v.f    v9, fa0
vfslide1down.vf v9, v9, ft0
vfslide1down.vf v8, v8, fa2
vfslide1down.vf v9, v9, ft1
vfslide1down.vf v8, v8, fa1
vfslide1down.vf v9, v9, ft2
vslidedown.vi   v9, v8, 4, v0.t
vfadd.vf    v8, v9, fa5
csrr    a2, vlenb
li  a3, 89
mul a2, a2, a3
add a2, a2, sp
addi    a2, a2, 2047
addi    a2, a2, 929
vs1r.v  v8, (a2)                        # Unknown-size Folded
Spill
...
```

Softmax

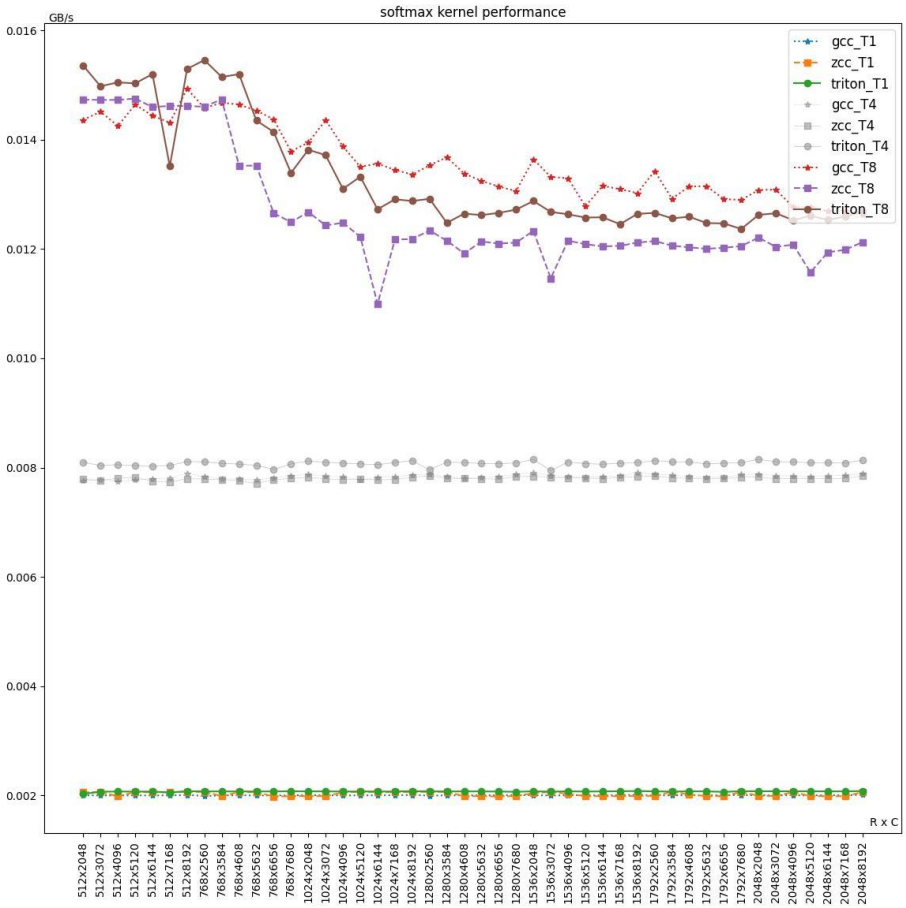

Fig4: Softmax kernel performance on different shapes

Average performance on different compiler							
Average (GB/s)	GCC	ZCC	Triton	Triton/ZCC	Triton/GCC	(Triton/ZCC)/(TN/T1)	(Triton/GCC)/(TN/T1)
T1	0.00200	0.00202	0.00207	102.53 %	103.50 %		
T4	0.00783	0.00780	0.00808	103.59 %	103.17 %	100.00 %	100.00 %
T8	0.01356	0.01267	0.01330	104.99	98.09%	100.00	100.00

				%		%	%
T4/T1	378.27 %	376.71 %	390.25 %	103.59 %	103.17 %		
T8/T1	654.91 %	611.87 %	642.39 %	104.99 %	98.09%		

The performance of Triton, ZCC, and GCC are similar in softmax. Triton shows 2-4% higher performance than both GCC and ZCC in single and four threaded executions. However, in the case of eight-thread, GCC outperformed Triton by 2%. The primary reasons for performance difference may include:

- First step in Safe-Softmax kernel (max element calculation): Both Triton and GCC generate vfredsum (vector reduction). Triton uses m4 grouping, while GCC uses m1. ZCC generates an unroll of 8 scalar fmax operations instead generating vfredsum.

```
Assembly language
// Triton
.LBB0_2:                                # %.lr.ph
                                        # =>This Inner Loop
Header: Depth=1
  vsetvli zero, zero, e32, m4, ta, mu
  vor.vx  v16, v12, a2
  slli  a4, a2, 2
  vmslt.vx  v0, v16, s6
  vmv4r.v v16, v20
  add a4, a4, s5
  vle32.v v16, (a4), v0.t
  addiw a2, a2, 32
  vfmax.vv  v8, v16, v8
  blt a2, s7, .LBB0_2
# %bb.3:                                # %._crit_edge
  bgtz  s7, .LBB0_5
  j .LBB0_20
.LBB0_4:
  vmv4r.v v8, v20
  blez  s7, .LBB0_20
.LBB0_5:                                # %.lr.ph6.preheader
  vsetvli zero, zero, e32, m4, ta, ma
  vfredmax.vs v8, v8, v8
```

```
Assembly language
```

```
// GCC
  vsetvli a5,zero,e32,m1,ta,ma
  vfmv.v.f  v1,fs0
  addi  a1,s2,1
  subw  a1,s0,a1
  addi  a2,s4,4
  li  a4,0
.L14:
  vl1re32.v v2,0(a2)
  mv  a5,a4
  addw  a4,a4,s2
  add a2,a2,s3
  vfmax.vv  v1,v1,v2
  bgeu  a1,a4,.L14
  vfmv.s.f  v2,fs2
  ld  a2,24(sp)
  vfredmax.vs v1,v1,v2
```

```
Assembly language
// ZCC
.LBB1_9:
  flw fa5, -12(a5)
  flw fa4, -8(a5)
  flw fa3, -4(a5)
  flw fa2, 0(a5)
  flw fa1, 4(a5)
  flw fa0, 12(a5)
  fmax.s  fa5, fa5, fs1
  flw ft0, 16(a5)
  fmax.s  fa4, fa3, fa4
  flw fa3, 8(a5)
  fmax.s  fa2, fa1, fa2
  fmax.s  fa1, ft0, fa0
  fmax.s  fa5, fa4, fa5
  fmax.s  fa4, fa3, fa2
  addi  a3, a3, 8
  add a4, a2, a3
  fmax.s  fa5, fa4, fa5
  fmax.s  fs1, fa1, fa5
  addi  a5, a5, 32
  bne a4, s11, .LBB1_9
```

- Second step (denominator calculation, the sum of the exponents of all elements): Triton, GCC, and ZCC all generate scalar operations, with Triton unrolling core loops 8 times, while GCC and ZCC do not.

```
Assembly language
// Triton
.LBB0_8:                                # %.lr.ph6
                                        # =>This Inner Loop
Header: Depth=1
  flw fa5, -16(s1)
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, -16(s0)
  flw fa5, -12(s1)
  fadd.s  fa4, fs1, fa0
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, -12(s0)
  flw fa5, -8(s1)
  fadd.s  fa4, fs1, fa0
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, -8(s0)
  flw fa5, -4(s1)
  fadd.s  fa4, fs1, fa0
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, -4(s0)
  flw fa5, 0(s1)
  fadd.s  fa4, fs1, fa0
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, 0(s0)
  flw fa5, 4(s1)
  fadd.s  fa4, fs1, fa0
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, 4(s0)
  flw fa5, 8(s1)
  fadd.s  fa4, fs1, fa0
```

```
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fsw fa0, 8(s0)
  flw fa5, 12(s1)
  fadd.s  fa4, fs1, fa0
  fadd.s  fs1, fa0, fa4
  fsub.s  fa0, fa5, fs0
  call  expf
  fadd.s  fa5, fs1, fa0
  fsw fa0, 12(s0)
  addi  s8, s8, 8
  addi  s1, s1, 32
  fadd.s  fs1, fa0, fa5
  addi  s0, s0, 32
  bne s10, s8, .LBB0_8
```

```
Assembly language
// GCC
.L11:
  flw fa0,0(s9)
  addi  s9,s9,4
  addi  s10,s10,4
  fsub.s  fa0,fa0,fs0
  call  expf@plt
  fsw fa0,-4(s10)
  fadd.s  fs1,fs1,fa0
  bne s9,s1,.L11
```

```
Assembly language
// ZCC
.LBB1_20:                               # %for.body16
                                        #   Parent Loop BB1_5
Depth=1
                                        # =>  This Inner Loop
Header: Depth=2
  flw fa5, 0(s10)
  fsub.s  fa0, fa5, fs1
  call  expf
  fsw fa0, 0(s3)
  lw  a0, 0(s1)
```

```
  fadd.s  fs2, fs2, fa0
  addi  s7, s7, 1
  addi  s3, s3, 4
  addi  s10, s10, 4
  blt s7, a0, .LBB1_20
```

- Third step (Softmax calculation): Triton, GCC, and ZCC all use `vfdiv.vf`, Triton uses m4 grouping, GCC uses m1 grouping, and ZCC uses m8 grouping.

```
Assembly language
// Triton
.LBB0_19:
  vsetvli zero, zero, e32, m4, ta, mu
  vor.vx  v12, v8, a0
  slli  a1, a0, 2
  vmslt.vx  v0, v12, s6
  vmv4r.v v12, v20
  add a1, a1, s3
  vle32.v v12, (a1), v0.t
  vfdiv.vf  v12, v12, fs1
  addiw a0, a0, 32
  vse32.v v12, (a1), v0.t
  blt a0, s7, .LBB0_19
```

```
Assembly language
// GCC
.L9:
  vsetvli a5,a3,e32,m1,ta,ma
  vle32.v v1,0(s11)
  slli  a1,a5,2
  sub a3,a3,a5
  add s11,s11,a1
  vfdiv.vv  v1,v1,v2
  vse32.v v1,0(a4)
  add a4,a4,a1
  bne a3,zero,.L9
```

```
Assembly language
// ZCC
.LBB1_28:
  vsetvli a2, a1, e32, m8, ta, ma
```

```
vle32.v v8, (s8)
slli  a3, a2, 2
sub a1, a1, a2
vfdiv.vf  v8, v8, fs2
vse32.v v8, (s8)
add s8, s8, a3
bnez  a1, .LBB1_28
```

Layernorm

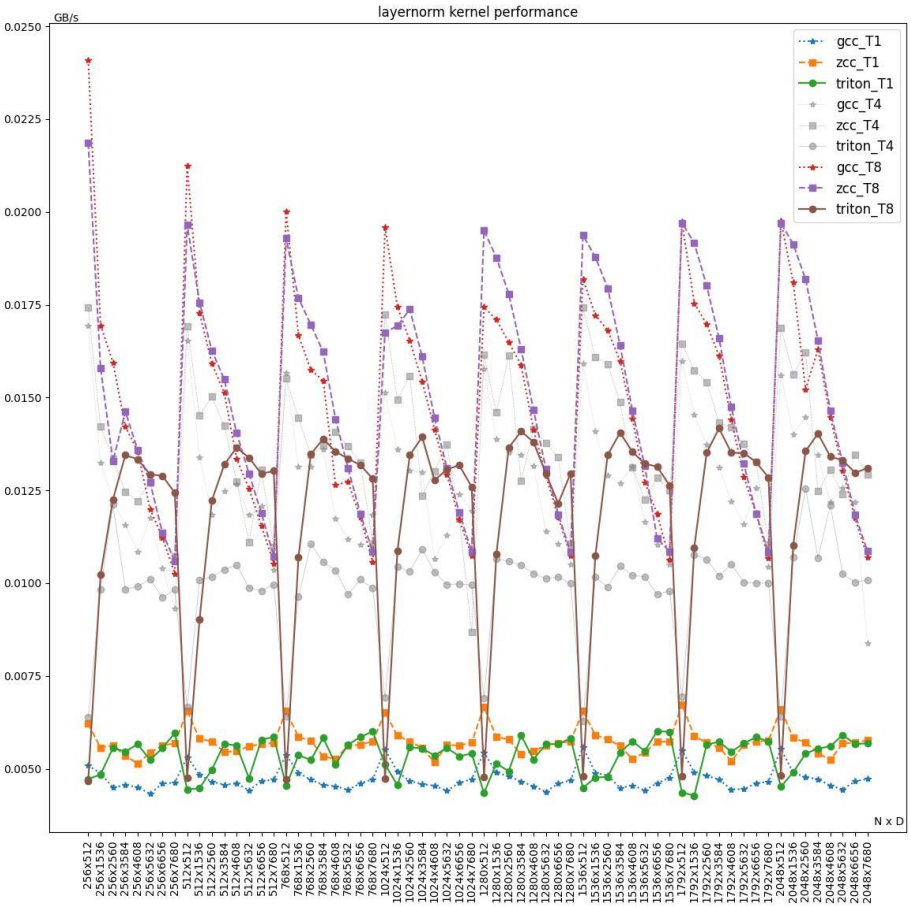

Fig5: Layernorm kernel performance on different shapes

Average performance on different compiler							
Average	GCC	ZCC	Triton	Triton/ZCC	Triton/GCC	(Triton/ZCC)/(	(Triton/GCC)/(

(GB/s)						TN/T1)	TN/T1)
T1	0.0047	0.0057	0.0053	93.28%	112.83 %		
T4	0.0127	0.0138	0.0098	71.21%	77.52%	104.29 %	105.23 %
T8	0.0147	0.0151	0.0118	78.67%	80.33%	101.70 %	101.56 %
T4/T1	237.86 %	256.63 %	175.23 %	68.28%	73.67%		
T8/T1	275.60 %	281.81 %	217.99 %	77.35%	79.09%		

> Note: The Layernorm performance data is calculated by adding the performance data of both forward kernel and backward kernel.

In single-threaded scenario, Triton outperforms GCC by 13%, but lags behind ZCC by 7%. However, in four and eight threads scenarios, ZCC and GCC outperform Triton by 20% to 25%. In addition, the performance of ZCC and GCC in multi-threaded scenarios can vary significantly in different shapes. The primary reasons for performance difference may include:

- The implementation of the Layernorm backward operator leads to significant performance differences under multi-threaded scenarios.
    - The reduction operation in the C kernel is implemented using OpenMP's reduction pragma.
    - The reduction operation in the Triton kernel requires the use of atomic operation.

```
C++
// C Layernorm backward
#pragma omp parallel for schedule(static)
num_threads(max_threads.value()) reduction(+ : dbias[:D])
reduction(+ : dweight[:D])
  for (int i = 0; i < N; i++) {
    // ...
    for (int j = 0; j < D; j++) {
      float norm = (inp_r[j] - mean_r) * rstd_r;
      float dnorm_j = weight[j] * dout_r[j];
```

```
      // gradient contribution to bias
      dbias[j] += dout_r[j];
      // gradient contribution to weight
      dweight[j] += norm * dout_r[j];
    }
  }
}
```

```
Python
   # Triton Layernorm backward
   for off in range(0, N, BLOCK_SIZE_N):
      # ...

      while tl.atomic_cas(Lock + (off /
BLOCK_SIZE_N).to(tl.int32), 0, 1) == 1:
        pass
      partial_dw += tl.load(DW + cols, mask=mask)
      partial_db += tl.load(DB + cols , mask=mask)
      tl.store(DW + cols,  partial_dw, mask=mask)
      tl.store(DB + cols , partial_db, mask=mask)

      # Release the lock
      tl.atomic_xchg(Lock + (off / BLOCK_SIZE_N).to(tl.int32), 0)
```

- Register grouping in the core loop of Layernorm
  - GCC tends to use smaller register groupings, with both forward and backward kernels using SEW/LMUL of e32/m1.
  - ZCC uses larger register groupings without causing register spills, with forward kernel using SEW/LMUL of e32/m4 and backward kernel using SEW/LMUL of e32/m2.
  - Triton selects the smallest register grouping that satisfies the given block size, with both forward and backward operators using SEW/LMUL of e32/m2.

By passing the compiler option `-mrvv-max-lmul=m2` to GCC, the SEW/LMUL is adjusted to e32/m2, which can lead to 10% performance gain in single-threaded mode.

Core loops of forward kernel:

```
Assembly language
// GCC
.L21:
```

```
  vsetvli a5,a4,e32,m1,ta,ma
  vle32.v v1,0(a3)
  vfmv.s.f  v2,fa5
  slli  a2,a5,2
  sub a4,a4,a5
  add a3,a3,a2
  vfredosum.vs  v1,v1,v2
  vfmv.f.s  fa5,v1
  bne a4,zero,.L21
// ...
.L10:
  vsetvli a5,s0,e32,m1,ta,ma
  vle32.v v1,0(a4)
  vfmv.s.f  v2,fa0
  slli  a3,a5,2
  sub s0,s0,a5
  add a4,a4,a3
  vfsub.vv  v1,v1,v3
  vfmul.vv  v1,v1,v1
  vfredosum.vs  v1,v1,v2
  vfmv.f.s  fa0,v1
  bne s0,zero,.L10
// ...
.L18:
  vsetvli a5,a2,e32,m1,ta,ma
  vle32.v v1,0(s1)
  vle32.v v2,0(a1)
  vle32.v v4,0(a0)
  slli  a4,a5,2
  sub a2,a2,a5
  add s1,s1,a4
  vfsub.vv  v1,v1,v3
  add a1,a1,a4
  add a0,a0,a4
  vfmul.vv  v1,v5,v1
  vfmadd.vv v2,v1,v4
  vse32.v v2,0(a3)
  add a3,a3,a4
  bne a2,zero,.L18
```

```
Assembly language
// ZCC
.LBB1_19:
```

```
  vsetvli a2, a1, e32, m8, ta, ma
  vle32.v v16, (a0)
  slli  a3, a2, 2
  sub a1, a1, a2
  vfredosum.vs  v8, v16, v8
  add a0, a0, a3
  bnez  a1, .LBB1_19
// ...
.LBB1_22:
  vsetvli a2, a1, e32, m4, ta, ma
  vle32.v v12, (a0)
  slli  a3, a2, 2
  sub a1, a1, a2
  vfsub.vf  v12, v12, fs2
  vfmul.vv  v12, v12, v12
  vfredosum.vs  v8, v12, v8
  add a0, a0, a3
  bnez  a1, .LBB1_22
// ...
.LBB1_33:
  vsetvli a0, a4, e32, m4, ta, ma
  vle32.v v8, (s2)
  vle32.v v12, (a1)
  vle32.v v16, (a2)
  slli  a5, a0, 2
  sub a4, a4, a0
  add s2, s2, a5
  add a1, a1, a5
  add a2, a2, a5
  vfsub.vf  v8, v8, fs2
  vfmul.vf  v8, v8, fa5
  vfmadd.vv v16, v8, v12
  vse32.v v16, (a3)
  add a3, a3, a5
  bnez  a4, .LBB1_33
```

```
Assembly language
// Triton
  vsetivli  zero, 16, e32, m2, ta, ma
  vmv.v.i v10, 0
  vmv.v.i v8, 0
.LBB0_2:
  slli  a0, t3, 2
```

```
  add a0, a0, t2
  vle32.v v12, (a0)
  addiw t3, t3, 16
  vfadd.vv  v8, v8, v12
  blt t3, t0, .LBB0_2
  li  t3, 0
  vmv.s.x v12, zero
  fcvt.s.w  fa4, a7
  vfredusum.vs  v8, v8, v12
  vfmv.f.s  fa5, v8
// ...
.LBB0_4:
  vsetvli zero, zero, e32, m2, ta, mu
  slli  a0, t3, 2
  add a0, a0, t2
  vle32.v v18, (a0)
  vor.vx  v20, v16, t3
  vmslt.vx  v0, v20, a7
  vmv2r.v v20, v10
  vfsub.vv  v20, v18, v8, v0.t
  vfmul.vv  v18, v20, v20
  addiw t3, t3, 16
  vfadd.vv  v14, v14, v18
  blt t3, t0, .LBB0_4
  vfredusum.vs  v10, v14, v12
  vfmv.f.s  fa3, v10
// ...
.LBB0_9:
  slli  a4, a0, 2
  vsetvli zero, zero, e32, m2, ta, ma
  add a5, a2, a4
  vle32.v v10, (a5)
  add a5, t2, a4
  vle32.v v12, (a5)
  add a5, a3, a4
  add a4, a4, a1
  vle32.v v14, (a5)
  vfsub.vv  v12, v12, v8
  vfmul.vf  v12, v12, fa4
  vfmul.vv  v10, v10, v12
  vfadd.vv  v10, v14, v10
  addiw a0, a0, 16
  vse32.v v10, (a4)
  blt a0, t0, .LBB0_9
```

Core loops of backward kernel:

```
Assembly language
// GCC
.L66:
  vsetvli a3,a2,e32,m1,ta,ma
  vle32.v v1,0(a7)
  vle32.v v2,0(t4)
  vle32.v v6,0(t6)
  vfmv.s.f  v4,fa4
  vfmv.s.f  v3,fa5
  slli  s1,a3,2
  vfsub.vv  v1,v1,v7
  sub a2,a2,a3
  vfmul.vv  v2,v2,v6
  add a7,a7,s1
  vfmul.vv  v1,v1,v5
  add t4,t4,s1
  add t6,t6,s1
  vfredosum.vs  v3,v2,v3
  vfmul.vv  v1,v1,v2
  vfredosum.vs  v1,v1,v4
  vfmv.f.s  fa5,v3
  vfmv.f.s  fa4,v1
  bne a2,zero,.L66
// ...
.L62:
  vsetvli a5,a1,e32,m1,ta,ma
  vle32.v v3,0(t3)
  vle32.v v4,0(t6)
  vle32.v v2,0(t1)
  vle32.v v1,0(t5)
  slli  a3,a5,2
  sub a1,a1,a5
  add t1,t1,a3
  vfadd.vv  v4,v3,v4
  vfsub.vv  v2,v2,v7
  vfmadd.vv v1,v3,v6
  add t5,t5,a3
  vse32.v v4,0(a6)
  vfmul.vv  v2,v2,v5
  add t3,t3,a3
  add t6,t6,a3
  vfsub.vv  v1,v1,v9
  add a6,a6,a3
```

```
  vle32.v v4,0(s1)
  add s1,s1,a3
  vfnmsac.vv  v1,v2,v8
  vfmadd.vv v2,v3,v4
  vse32.v v2,0(a7)
  add a7,a7,a3
  vle32.v v2,0(t4)
  add t4,t4,a3
  vfmadd.vv v1,v5,v2
  vse32.v v1,0(a2)
  add a2,a2,a3
  bne a1,zero,.L62
```

```
Assembly language
// ZCC
.LBB3_24:
  vsetvli a0, s1, e32, m2, ta, ma
  vle32.v v12, (a4)
  vle32.v v14, (a3)
  vle32.v v16, (a1)
  slli  a5, a0, 2
  sub s1, s1, a0
  add a4, a4, a5
  add a3, a3, a5
  vfsub.vf  v12, v12, fa3
  vfmul.vv  v14, v14, v16
  vfmul.vf  v12, v12, fa2
  vfredosum.vs  v9, v14, v9
  vfmul.vv  v12, v14, v12
  vfredosum.vs  v10, v12, v10
  add a1, a1, a5
  bnez  s1, .LBB3_24
// ...
.LBB3_33:
  vsetvli a0, s1, e32, m2, ta, ma
  vle32.v v10, (t2)
  vle32.v v12, (a1)
  vle32.v v14, (a4)
  vle32.v v16, (t1)
  vle32.v v18, (t2)
  vfmul.vv  v12, v10, v12
  vfadd.vv  v10, v10, v14
  vle32.v v14, (a3)
```

```
  vse32.v v10, (a4)
  vfsub.vf  v10, v16, fa3
  vfmul.vf  v10, v10, fa2
  vfmacc.vv v14, v10, v18
  vle32.v v16, (a7)
  slli  a2, a0, 2
  sub s1, s1, a0
  add t1, t1, a2
  add t2, t2, a2
  add a1, a1, a2
  vfadd.vf  v12, v12, fa5
  add a4, a4, a2
  vfsub.vf  v12, v12, fa1
  vse32.v v14, (a3)
  vfnmsac.vf  v12, fa0, v10
  add a3, a3, a2
  vfmul.vf  v10, v12, fa2
  vfadd.vv  v10, v10, v16
  vse32.v v10, (a7)
  add a7, a7, a2
  bnez  s1, .LBB3_33
```

```
Assembly language
// Triton
.LBB0_2:
  vsetvli zero, zero, e32, m2, ta, mu
  vor.vx  v14, v8, a3
  slli  a7, a3, 2
  vmv2r.v v16, v10
  vmslt.vx  v0, v14, t0
  add a4, a6, a7
  vle32.v v14, (a4)
  add a4, t2, a7
  add a7, a7, a5
  vle32.v v18, (a4)
  vle32.v v20, (a7)
  vfsub.vf  v14, v14, fa5
  vfmul.vf  v16, v14, fa4, v0.t
  vmv2r.v v14, v10
  addiw a3, a3, 16
  vfmul.vv  v14, v18, v20, v0.t
  vfmul.vv  v16, v16, v14
```

```
  vfredusum.vs  v13, v14, v12
  vfredusum.vs  v14, v16, v12
  vfmv.f.s  fa1, v14
  fadd.s  fa3, fa3, fa1
  vfmv.f.s  fa1, v13
  fadd.s  fa2, fa2, fa1
  blt a3, t0, .LBB0_2
// ...
.LBB0_6:
  vsetvli zero, zero, e32, m2, ta, mu
  vor.vx  v14, v8, t4
  slli  t3, t4, 2
  vmv2r.v v12, v10
  vmv2r.v v16, v10
  vmslt.vx  v0, v14, t0
  add a3, a6, t3
  vle32.v v18, (a3)
  add a3, t2, t3
  vle32.v v14, (a3)
  add a3, a5, t3
  vfsub.vf  v18, v18, fa5
  vfmul.vf  v12, v18, fa4, v0.t
  vle32.v v18, (a3)
  fcvt.s.w  fa0, t4
  add a3, t1, t3
  fmul.s  fa0, fa0, fa1
  vfmul.vv  v16, v14, v18, v0.t
  vfmul.vf  v18, v12, fa3
  vfadd.vf  v18, v18, fa2
  vfsub.vv  v16, v16, v18
  vfmul.vf  v16, v16, fa4
  vse32.v v16, (a3)
  fcvt.w.s  a3, fa0, rtz
  slli  a3, a3, 2
  add a3, a3, a7
.LBB0_7:
.LBB0_17:
  lr.w.aq a4, (a3)
  bnez  a4, .LBB0_19
  sc.w.rl a0, t5, (a3)
  bnez  a0, .LBB0_17
.LBB0_19:
  bne a4, t5, .LBB0_15
.LBB0_20:
```

```
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_22
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_20
.LBB0_22:
  bne a0, t5, .LBB0_15
.LBB0_23:
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_25
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_23
.LBB0_25:
  bne a0, t5, .LBB0_15
.LBB0_26:
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_28
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_26
.LBB0_28:
  bne a0, t5, .LBB0_15
.LBB0_29:
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_31
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_29
.LBB0_31:
  bne a0, t5, .LBB0_15
.LBB0_32:
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_34
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_32
.LBB0_34:
  bne a0, t5, .LBB0_15
.LBB0_35:
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_37
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_35
.LBB0_37:
  bne a0, t5, .LBB0_15
.LBB0_38:
  lr.w.aq a0, (a3)
  bnez  a0, .LBB0_40
```

```
  sc.w.rl a4, t5, (a3)
  bnez  a4, .LBB0_38
.LBB0_40:
  beq a0, t5, .LBB0_7
.LBB0_15:
  add a0, a1, t3
  add t3, t3, a2
  vle32.v v16, (a0)
  vle32.v v18, (t3)
  vfmul.vv  v12, v14, v12
  vfadd.vv  v12, v12, v16
  vfadd.vv  v14, v14, v18
  vse32.v v12, (a0)
  vse32.v v14, (t3)
  amoswap.w.aqrl  zero, zero, (a3)
  addiw t4, t4, 16
  blt t4, t0, .LBB0_6
```

- Mask load: Triton tends to generate more instructions within loops, primarily due to handling explicit maskload operations, which may result in slightly lower single-threaded performance compared to ZCC.
- Instruction fusion: GCC/ZCC uses vfmadd for multiply-accumulate operations, while Triton performs separate mul and add operations.
- In the forward part, Triton employs `vfredusum` outside the loop and calculates inside the loop using vfadd, which typically results in better performance.

Resize

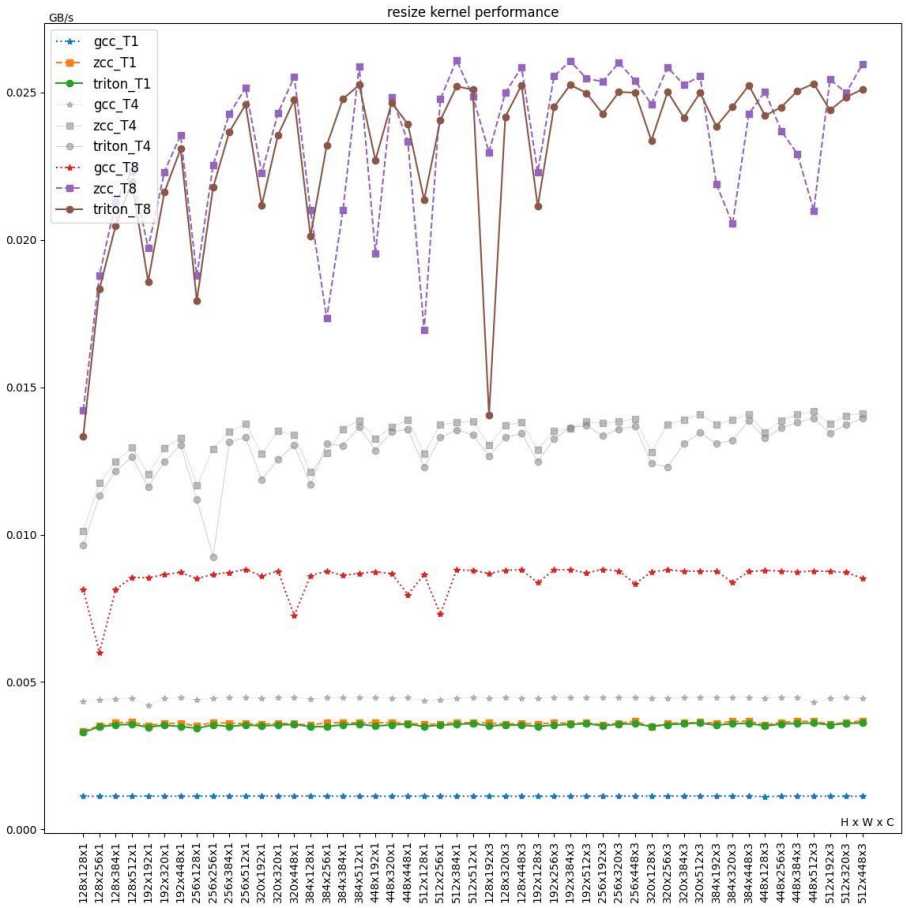

Fig6: Resize kernel performance on different shape

Average performance on different compiler							
Averag e(GB/s)	GCC	ZCC	Triton	Triton/Z CC	Triton/G CC	(Triton/ ZCC)/(TN/T1)	(Triton/ GCC)/(TN/T1)
T1	0.0011	0.0036	0.0035	98.43%	314.69 %		
T4	0.0044	0.0134	0.0129	96.35%	289.70 %	100.00 %	100.07 %
T8	0.0086	0.0231	0.0231	100.35	269.72	100.01	100.11

				%	%	%	%
T4/T1	125.67 %	377.60 %	363.81 %	96.35%	289.49 %		
T8/T1	242.47 %	650.97 %	653.24 %	100.35 %	269.42 %		

Triton and ZCC show similar performance across all threads and input shapes, while GCC's performance lags significantly. The primary reason for performance difference may include:

- GCC's use of strided load for discrete memory accesses, resulting in many additional instructions. In contrast, ZCC and Triton use index load directly, resulting in fewer instructions generated compared to GCC, without producing permutation-type instructions (such as `vslidedown`).

For example, in the following C source code, the index `x0` and `x1` for `src_ptr0` and `src_ptr1` are variables with no fixed stride length, making them unsuitable for strided load. However, GCC still uses the strided load instruction `vlse8.vv5,0(a3),zero` (with a stride value of 0) and extracts each element from the vector register into scalar registers using `vslidedown.vi` and `vmv.x.s` for computation. This optimization approach, which results in a large number of unnecessary instructions, is essentially a deoptimization.

```C++
// Resize kernel: input data load
for (size_t w = 0; w < dst_width; w++)
{
    uint16_t input_x = (uint16_t)w << (hw_fl - 1);
    uint16_t x0 = (input_x >> hw_fl);
    uint16_t x1 = std::min(x0 + (uint16_t)1, width - (uint16_t)1);

    int16_t y0x0 = src_ptr0[x0];
    int16_t y0x1 = src_ptr0[x1];
    int16_t y1x0 = src_ptr1[x0];
    int16_t y1x1 = src_ptr1[x1];
    // ...
}
```

```Assembly language
// GCC
.L12:
  vmv1r.v v1,v6
```

```
  vadd.vi v6,v6,4
  vsetvli zero,zero,e32,mf2,ta,ma
  vncvt.x.x.w v1,v1
  vsll.vx v1,v1,t5
  vand.vv v4,v1,v9
  vsetvli zero,zero,e16,mf4,ta,ma
  vncvt.x.x.w v1,v1
  vsetvli zero,zero,e32,mf2,ta,ma
  vsra.vx v18,v4,a6
  vsetvli zero,zero,e8,mf8,ta,ma
  vncvt.x.x.w v1,v1
  vsetvli zero,zero,e32,mf2,ta,ma
  vadd.vi v4,v18,1
  vsll.vx v13,v18,a6
  vslidedown.vi v17,v18,1
  vmin.vv v4,v4,v11
  vslidedown.vi v16,v18,2
  vslidedown.vi v15,v18,3
  vsetvli zero,zero,e16,mf4,ta,ma
  vncvt.x.x.w v19,v4
  vncvt.x.x.w v13,v13
  vsetvli zero,zero,e8,mf8,ta,ma
  vsub.vv v14,v10,v1
  vsetvli zero,zero,e16,mf4,ta,ma
  vslidedown.vi v5,v19,1
  vmv.x.s a0,v19
  vslidedown.vi v4,v19,2
  vmv.x.s a1,v5
  slli  a0,a0,48
  vsetvli zero,zero,e32,mf2,ta,ma
  vmv.x.s s10,v18
  srli  a0,a0,48
  add a3,a5,a0
  vlse8.v v5,0(a3),zero
  vsetvli zero,zero,e16,mf4,ta,ma
  add a3,a5,s10
  vslidedown.vi v18,v19,3
  vmv.x.s a2,v4
  slli  a1,a1,48
  vlse8.v v4,0(a3),zero
  vsetvli zero,zero,e32,mf2,ta,ma
  vmv.x.s a7,v17
  srli  a1,a1,48
  vsetivli  zero,4,e16,mf4,ta,ma
```

```
  add a3,a5,a1
  add a0,a4,a0
  lbu s4,0(a3)
  add s10,a4,s10
  vmv.x.s a3,v18
  vlse8.v v18,0(a0),zero
  add a0,a5,a7
  vlse8.v v17,0(s10),zero
  lbu s10,0(a0)
  ...
```

```
Assembly language
// ZCC
.LBB1_21:
  vsetvli a4, t2, e32, m4, ta, ma
  lwu t1, 0(s4)
  lhu a0, 0(s2)
  sub t2, t2, a4
  addi  a1, t1, -1
  addi  a0, a0, -1
  sllw  a5, s7, t1
  vsll.vx v12, v8, a1
  sllw  a1, t0, t1
  subw  a1, a7, a1
  vand.vx v16, v12, a3
  vsrl.vx v16, v16, t1
  vadd.vi v20, v16, 1
  vmin.vx v20, v20, a0
  andi  a0, a1, 255
  subw  a1, a5, a1
  andi  a1, a1, 255
  vsll.vx v24, v16, t1
  vsub.vv v24, v12, v24
  vand.vx v12, v24, a6
  vrsub.vx  v24, v24, a5
  vsetvli zero, zero, e8, m1, ta, ma
  vluxei32.v  v28, (t3), v16
  vsetvli zero, zero, e32, m4, ta, ma
  vand.vx v20, v20, a3
  vsetvli zero, zero, e8, m1, ta, ma
  vluxei32.v  v29, (t3), v20
  vluxei32.v  v30, (a2), v16
  vluxei32.v  v16, (a2), v20
```

```
  vsetvli zero, zero, e32, m4, ta, ma
  vand.vx v20, v24, a6
  vsext.vf4 v24, v28
  vmul.vv v24, v20, v24
  vsext.vf4 v4, v29
  vmacc.vv  v24, v12, v4
  vsext.vf4 v4, v16
  vsext.vf4 v16, v30
  vmul.vv v16, v20, v16
  vsra.vx v20, v24, t1
  vmacc.vv  v16, v12, v4
  vsra.vx v12, v16, t1
  vsll.vi v16, v20, 16
  vsll.vi v12, v12, 16
  vsra.vi v16, v16, 16
  vsra.vi v12, v12, 16
  vmul.vx v16, v16, a1
  vmacc.vx  v16, a0, v12
  vsetvli zero, zero, e16, m2, ta, ma
  vnsra.wx  v12, v16, t1
  vsetvli zero, zero, e8, m1, ta, ma
  vnsrl.wi  v14, v12, 0
  vse8.v  v14, (s5)
  vsetvli zero, zero, e32, m4, ta, ma
  vadd.vx v8, v8, a4
  add s5, s5, a4
  bnez  t2, .LBB1_21
```

```
Assembly language
// Triton
.LBB0_4:
  vsetvli zero, zero, e32, m4, ta, ma
  vor.vx  v16, v4, a2
  vmv1r.v v2, v3
  vmv1r.v v1, v3
  vmslt.vx  v0, v16, a7
  vsll.vi v16, v16, 6
  vsra.vi v20, v16, 7
  vsetvli zero, zero, e64, m8, ta, ma
  vsext.vf2 v24, v20
  vsetvli zero, zero, e8, m1, ta, mu
  vluxei64.v  v2, (a4), v24, v0.t
  vluxei64.v  v1, (a5), v24, v0.t
```

```
  vmv1r.v v25, v3
  vmv1r.v v24, v3
  add a3, a1, a2
  vsetvli zero, zero, e32, m4, ta, ma
  vadd.vi v20, v20, 1
  vmin.vx v20, v20, t5
  vsetvli zero, zero, e64, m8, ta, ma
  vsext.vf2 v8, v20
  vsetvli zero, zero, e8, m1, ta, mu
  vluxei64.v  v25, (a4), v8, v0.t
  vluxei64.v  v24, (a5), v8, v0.t
  vsetvli zero, zero, e32, m4, ta, ma
  vand.vx v8, v16, a6
  vrsub.vx  v12, v8, t1
  vsext.vf4 v16, v2
  vsext.vf4 v20, v1
  vmul.vv v16, v12, v16
  vmul.vv v12, v12, v20
  vsext.vf4 v20, v25
  vmadd.vv  v20, v8, v16
  vsext.vf4 v16, v24
  vmadd.vv  v16, v8, v12
  vsrl.vi v8, v20, 7
  vsrl.vi v12, v16, 7
  vmul.vx v12, v12, t0
  vmacc.vx  v12, a0, v8
  vsetvli zero, zero, e16, m2, ta, ma
  vnsrl.wi  v8, v12, 7
  vsetvli zero, zero, e8, m1, ta, ma
  vnsrl.wi  v10, v8, 0
  addiw a2, a2, 32
  vse8.v  v10, (a3), v0.t
  blt a2, a7, .LBB0_4
```

Warp

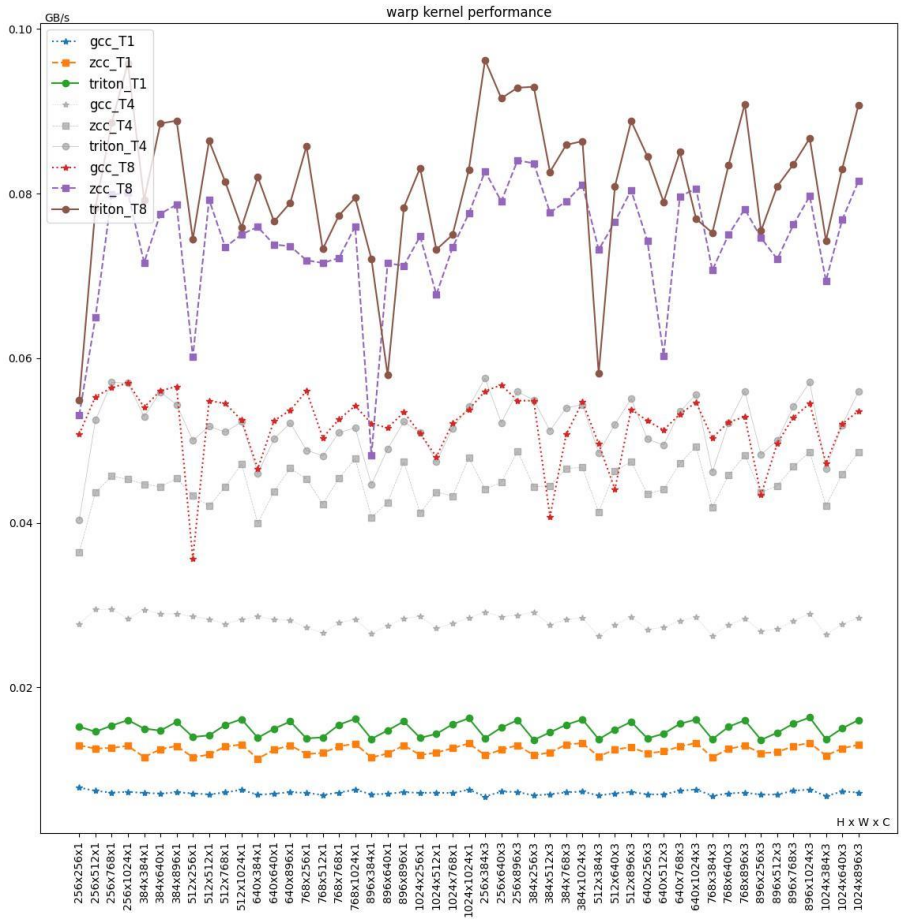

Fig7: Warp kernel performance on different shapes

Average performance on different compiler							
Average(GB/s)	GCC	ZCC	Triton	Triton/ZCC	Triton/GCC	(Triton/ZCC)/(TN/T1)	(Triton/GCC)/(TN/T1)
T1	0.007	0.012	0.015	120.39 %	208.86 %		
T4	0.028	0.045	0.052	115.27 %	184.87 %	100.00 %	100.15 %
T8	0.052	0.074	0.081	109.11	155.61	99.99%	100.04

				%	%		%
T4/T1	187.51 %	300.27 %	346.13 %	115.27 %	184.59 %		
T8/T1	349.68 %	498.48 %	543.93 %	109.12 %	155.55 %		

Triton kernel shows a 10% to 20% overall performance advantage compared to ZCC and a 50% to 100% advantage over GCC. The primary reasons for performance difference may include:

- Core loop register grouping:
  - GCC tends to use smaller register groupings, with SEW/LMUL set to e64/m1.
  - ZCC and Triton use the largest SEW/LMUL of e64/m8.
  - By passing `-mrvv-max-lmul=m8` option to GCC, the SEW/LMUL is adjusted to e64/m8, which can lead to an impressive 100% performance gain in single-threaded scenario.
- In the core loop, ZCC translates one of the sequential access operations into an index load operation, which performs worse than a unit stride load. This is evident in the highlighted three memory access instructions in the assembly code below, which are used to read data from the input array.

```
Assembly language
// GCC
.L10:
  vmv1r.v v0,v7
  vsetvli a5,a3,e32,mf2,ta,ma
  vle16.v v1,0(a1)
  vncvt.x.x.w v0,v0
  vsetvli a2,zero,e64,m1,ta,ma
  vmv.v.x v9,a5
  vsetvli zero,a5,e16,mf4,ta,ma
  vncvt.x.x.w v0,v0
  vsetvli zero,zero,e8,mf8,ta,ma
  vnsra.wi  v2,v1,8
  vncvt.x.x.w v0,v0
  vsetvli zero,zero,e16,mf4,ta,ma
  vsll.vi v1,v1,8
  vsetvli zero,zero,e8,mf8,ta,ma
  vsub.vv v0,v0,v2
  vnsra.wi  v2,v1,8
  slli  s6,a5,1
  vadd.vi v3,v0,-1
```

```
  vsetvli zero,zero,e64,m1,ta,ma
  vzext.vf8 v6,v0
  vsetvli zero,zero,e8,mf8,ta,ma
  vmsge.vi  v1,v0,0
  vsetvli zero,zero,e64,m1,ta,ma
  vzext.vf8 v4,v3
  vsetvli zero,zero,e8,mf8,ta,ma
  vluxei64.v  v6,(s5),v6
  vluxei64.v  v4,(s5),v4
  vmsge.vi  v5,v3,0
  vsetvli zero,zero,e16,mf4,ta,ma
  vsext.vf2 v3,v2
  vsetvli a2,zero,e64,m1,ta,ma
  vmv1r.v v0,v1
  vadd.vv v7,v7,v9
  vmv1r.v v9,v8
  vsetvli zero,a5,e16,mf4,ta,mu
  vsext.vf2 v10,v6
  vsext.vf2 v9,v6,v0.t
  vmv1r.v v0,v5
  vsetvli zero,zero,e8,mf8,ta,ma
  sub a3,a3,a5
  vmerge.vvm  v2,v11,v4,v0
  vsetvli zero,zero,e16,mf4,ta,mu
  vmv1r.v v0,v1
  vmv1r.v v4,v8
  vsext.vf2 v1,v2
  add a1,a1,s6
  vsll.vi v4,v10,8,v0.t
  vsub.vv v1,v1,v9
  vmadd.vv  v3,v1,v4
  vsetvli zero,zero,e8,mf8,ta,ma
  vnsra.wi  v3,v3,8
  vse8.v  v3,0(a4)
  add a4,a4,a5
  bne a3,zero,.L10
```

```
Assembly language
// ZCC
  vsetvli zero, a2, e64, m8, ta, ma
  vid.v v16
.LBB1_15:
  vsetvli a1, a2, e64, m8, ta, ma
```

```
  ld  a4, 0(s10)
  ld  a0, 0(s4)
  vadd.vv v8, v16, v16
  vsetvli zero, zero, e16, m2, ta, ma
  mul s0, a4, a5
  slli  s0, s0, 1
  add a0, a0, s0
  vluxei64.v  v24, (a0), v8
  vsrl.vi v8, v24, 8
  vsetvli zero, zero, e32, m4, ta, ma
  vzext.vf2 v12, v24
  vzext.vf2 v4, v8
  vwsubu.wv v24, v16, v4
  vnsrl.wi  v8, v24, 0
  vsetvli zero, zero, e16, m2, ta, ma
  vnsrl.wi  v6, v8, 0
  vsetvli zero, zero, e8, m1, ta, ma
  ld  a0, 0(s3)
  ld  s0, 0(s9)
  vnsrl.wi  v8, v6, 0
  vadd.vi v9, v8, -1
  vsetvli zero, zero, e64, m8, ta, ma
  vand.vx v24, v24, t1
  vsetvli zero, zero, e8, m1, ta, ma
  mul s0, s0, a3
  add s0, s0, a5
  mul a4, s0, a4
  add a0, a0, a4
  vluxei64.v  v10, (a0), v24
  vluxei8.v v11, (a0), v9
  vmsle.vi  v0, v8, -1
  vmsle.vi  v8, v9, -1
  vmerge.vim  v9, v10, 0, v0
  vmv.v.v v0, v8
  vmerge.vim  v8, v11, 0, v0
  vsetvli zero, zero, e32, m4, ta, ma
  vsll.vi v12, v12, 24
  vsra.vi v12, v12, 24
  vsetvli zero, zero, e16, m2, ta, ma
  vsext.vf2 v10, v9
  vsext.vf2 v24, v8
  vwsub.vv  v28, v24, v10
  vsetvli zero, zero, e32, m4, ta, ma
  vmul.vv v12, v28, v12
```

```
  vsetvli zero, zero, e16, m2, ta, ma
  vnsrl.wi  v10, v12, 8
  vsetvli zero, zero, e8, m1, ta, ma
  ld  a0, 0(s2)
  sub a2, a2, a1
  vnsrl.wi  v8, v10, 0
  vadd.vv v8, v9, v8
  add a0, a0, a4
  vsoxei64.v  v8, (a0), v16
  vsetvli zero, zero, e64, m8, ta, ma
  vadd.vx v16, v16, a1
  bnez  a2, .LBB1_15
```

```
Assembly language
// Triton
.LBB0_2:
  vsetvli zero, zero, e32, m4, ta, ma
  csrr  a4, vlenb
  li  t1, 6
  mul a4, a4, t1
  add a4, a4, sp
  addi  a4, a4, 16
  vl4r.v  v8, (a4)
  vor.vx  v24, v8, a3
  addw  a4, a7, a3
  csrr  t1, vlenb
  slli  t1, t1, 2
  add t1, t1, sp
  addi  t1, t1, 16
  vl2r.v  v22, (t1)
  vmslt.vx  v2, v24, a5
  slli  a4, a4, 1
  vsetvli zero, zero, e16, m2, ta, mu
  add a4, a4, a1
  vmv1r.v v0, v2
  vle16.v v22, (a4), v0.t
  vmv1r.v v21, v20
  vsetvli zero, zero, e8, m1, ta, ma
  vnsrl.wi  v9, v22, 8
  vsetvli zero, zero, e16, m2, ta, ma
  vnsrl.wi  v28, v24, 0
  vsetvli zero, zero, e8, m1, ta, ma
  vnsrl.wi  v24, v28, 0
```

```
vsub.vv v1, v24, v9
vsetvli zero, zero, e16, m2, ta, ma
vsext.vf2 v24, v1
vsetvli zero, zero, e8, m1, ta, ma
vadd.vi v3, v1, -1
vsetvli zero, zero, e16, m2, ta, ma
addi  a4, sp, 16
vl4r.v  v8, (a4)
vwadd.wv  v4, v8, v24
vsext.vf2 v30, v3
vwadd.wv  v16, v8, v30
vsetvli zero, zero, e64, m8, ta, ma
vsext.vf2 v24, v4
vsetvli zero, zero, e8, m1, ta, mu
vluxei64.v  v21, (a0), v24, v0.t
vmv1r.v v24, v20
addw  a4, a6, a3
add a4, a4, a2
vsetvli zero, zero, e64, m8, ta, ma
vsext.vf2 v8, v16
vsetvli zero, zero, e8, m1, ta, mu
vluxei64.v  v24, (a0), v8, v0.t
vmsle.vi  v0, v1, -1
vmsle.vi  v9, v3, -1
vmerge.vim  v8, v21, 0, v0
vmv.v.v v0, v9
vmerge.vim  v9, v24, 0, v0
vwsub.vv  v10, v9, v8
vsetvli zero, zero, e16, m2, ta, ma
vsll.vi v12, v22, 8
vsra.vi v12, v12, 8
vmul.vv v10, v10, v12
vsetvli zero, zero, e8, m1, ta, ma
vnsrl.wi  v9, v10, 8
vadd.vv v8, v8, v9
addiw a3, a3, 32
vmv1r.v v0, v2
vse8.v  v8, (a4), v0.t
blt a3, t0, .LBB0_2
```

Correlation

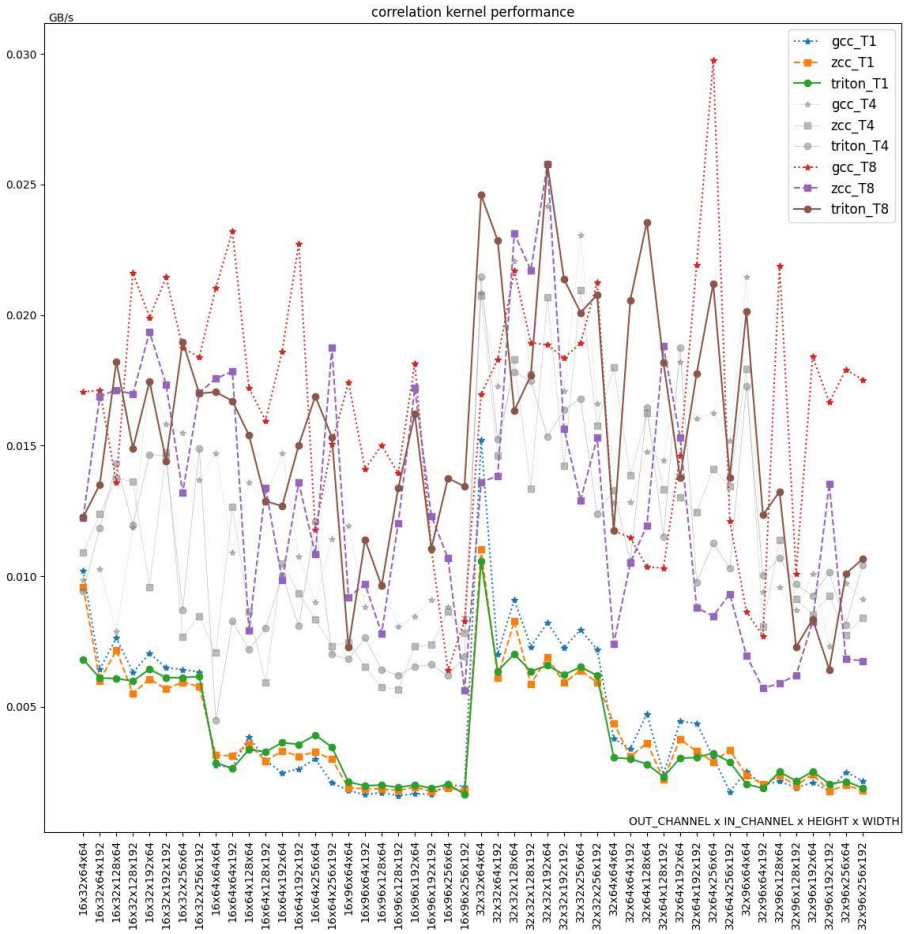

Fig8: Correlation kernel performance on different shapes

Average performance on different compiler							
Average(GB/s)	GCC	ZCC	Triton	Triton/ZCC	Triton/GCC	(Triton/ZCC)/(TN/T1)	(Triton/GCC)/(TN/T1)
T1	0.0043	0.0040	0.0039	98.34%	90.76%		
T4	0.0133	0.0115	0.0112	97.89%	84.46%	101.54%	101.85%
T8	0.0165	0.0129	0.0155	120.48%	93.83%	101.35%	101.39%

T4/T1	389.66 %	335.16 %	323.12 %	96.41%	82.92%		
T8/T1	502.17 %	390.96 %	464.75 %	118.87 %	92.55%		

In single and four-threaded scenarios, Triton's performance is close to ZCC. However, under eight threads, Triton has a 20% performance advantage over ZCC. But GCC has a 10-15% performance advantage over Triton.

The primary reasons of performance difference may include:

- Register grouping in the main loop**:**
  - GCC uses SEW/LMUL of e8/mf4.
  - ZCC uses SEW/LMUL of e8/m4.
  - Triton uses SEW/LMUL of e8/mf4.

  To align the implementations of Triton and C kernel, vectorization on the width dimension needs to be added for the C kernel. However, GCC 15.0.0 does not support this feature, and it is not enabled in ZCC either. Therefore, we manually introduced tiling operations for the width dimension to achieve vectorization. Due to the small tiling size, ZCC cannot fully utilize larger register groupings compared to GCC, resulting in bad performance.

  The implementation of the Correlation C kernel, applied after tiling the input data along the width dimension:

```C++
// Tiling on width loop. Keep the same block size as Triton
kernel.
  const size_t BLOCK_SIZE_W = 8;
  #pragma omp parallel for collapse(2)
num_threads(max_threads.value())
  for (size_t d = 0; d < out_channel; ++d) {
    for (size_t i = 0; i < height; ++i) {
      // Tiling
      for (size_t j = d; j < width; j += BLOCK_SIZE_W) {

        // Correlation: reduction on input data in_channel
dimension
        int16_t sum_data[BLOCK_SIZE_W] = {0};
        for (size_t k = 0; k < in_channel; ++k) {
          size_t vl = std::min(BLOCK_SIZE_W, width - j);
          #pragma omp simd
          for (size_t w = 0; w < vl; ++w) {
            size_t in_idx1 = k * width * height + i * width +
```

```
j + w;
            size_t in_idx2 = in_idx1 - d;
            sum_data[w] += (int16_t)(src0_arr[in_idx1]) *
src1_arr[in_idx2];
          }
        }
        // Normalilze for input data in_channel dimension
reduction result
        size_t vl = std::min(BLOCK_SIZE_W, width - j);
        #pragma omp simd
        for (size_t w = 0; w < vl; ++w) {
          size_t out_idx = d * width * height + i * width + j
+ w;
          out_arr[out_idx] = (int8_t)(sum_data[w] >>
out_shift);
        }

      }
    }
  }
```

- ZCC RVV register grouping optimization by manually adjusting the register groupings: Change e8/m4 to e8/mf2 and e32/m4 to e32/m1. It can be observed that this optimization yields a 10% improvement in single-threaded mode, a 9% improvement in four-threaded mode, and a 30% improvement in eight-threaded mode.

Average performance on different compiler			
Average(GB/s)	ZCC	Triton	Triton/ZCC
T1	0.0044	0.0039	88.44%
T4	0.0126	0.0112	88.95%
T8	0.0185	0.0155	83.89%

- Instruction fusion: GCC/ZCC uses vwmacc for multiply-accumulate operations, while Triton performs separate mul and add operations.

```
Assembly language
// GCC
vsetvli    zero,s8,e8,mf2,ta,ma
vle16.v    v1,0(a0)
```

```
vle8.v  v4,0(s10)
vle8.v  v3,0(s9)
vwmacc.vv   v1,v3,v4
vse16.v v1,0(a0)
addi   a7,a7,1
add s10,s10,a6 // address calc
add s9,s9,a6
bne    t3,a7,.L19
```

```
Assembly language
// ZCC
vsetvli    a2, a3, e8, m4, ta, ma
vle8.v  v16, (s1)
vle8.v  v20, (a4)
vle16.v v8, (a5)
slli   a1, a2, 1
sub a3, a3, a2 // address calc
add s1, s1, a2
add a4, a4, a2
vwmacc.vv   v8, v20, v16
vse16.v v8, (a5)
add    a5, a5, a1
bnez    a3, .LBB1_17
```

```
Assembly language
// Triton
vsetvli    zero, zero, e64, m2, ta, ma
vmv.x.s a0, v10 // address calc
vmv1r.v v18, v15
vsetvli zero, zero, e8, mf4, ta, mu
vle8.v  v18, (a0), v0.t
vsetvli zero, zero, e64, m2, ta, ma
vadd.vv v20, v8, v12 // address calc
vmv.x.s a0, v20
vmv1r.v v19, v15
vsetvli zero, zero, e8, mf4, ta, mu
vle8.v  v19, (a0), v0.t
vmul.vv v18, v19, v18
vwadd.wv    v14, v14, v18

vsetvli zero, zero, e64, m2, ta, ma
```

```
vadd.vv v10, v10, v16 // address calc
vadd.vv v8, v8, v16
```

- Address calculation: Triton uses vector instructions for address calculation but only utilizes the first element of the vector register, behaving like a scalar operation. This results in redundant `vmv1r.v` operations. In contrast, GCC and ZCC use scalar computations directly.

Triton-CPU Performance Issues

Summary

Despite facing performance issues on the RISC-V platform, such as register spill, fixed-length vector, discrete memory access vectorization, and the context storage overhead introduced by multithreading, testing results indicate that the performance of Triton-CPU on the RISC-V architecture can nearly reach that of traditional C kernel compilers through a series of experimental optimizations. This result not only validates Triton's immense potential in the high-performance computing field but also highlights its broad application prospects in the open-source RISC-V architecture.

Looking ahead, as more and more optimization to be applied to Triton compiler, Triton is expected to further overcome current performance bottlenecks, thereby becoming the preferred kernel library programming language solution for RISC-V platform. Additionally, with the ongoing proliferation of the RISC-V architecture in the industry and the continuous expansion of application scenarios, the Triton kernel library, with its high usability and maintainability, as well as the compatibility and scalability brought by the MLIR (Multi-Level Intermediate Representation) compiler technology stack, will significantly enhance its competitiveness in the ecosystem.

Moreover, the synergistic development of related ecosystems, such as RISC-V, Triton, and MLIR, will promote the shared prosperity of open-source technology, further driving the deep integration of high-performance computing and open-source hardware architectures. Through this synergy, not only can technological innovation be accelerated, but a positive ecological cycle can also be formed, providing stronger support for future computing demands.

References

[1] Triton Conference @ Silicon Valley: Chip and AI Giants Gather (Triton 大会@硅谷：芯片、AI 大厂齐站台) *https://mp.weixin.qq.com/s/euX2nxQ4lhG6yaLYMugyrw*

[2] Triton Conference 2024 *https://www.youtube.com/@Triton-openai/videos*

[3] Opening a New Era in Large Models: The Evolution and Impact of Triton (开启大模型时代新纪元：Triton 的演变与影响力)
https://Triton.csdn.net/66f22b7759bcf8384a63a1c9.html

[4] August Triton Community Meetup
https://www.youtube.com/watch?v=dfL3L4_3ujg&t=634s

[5] Triton-CPU Repository *https://github.com/Triton-lang/Triton-cpu*

[6] Triton's documentation *https://Triton-lang.org/main/python-api/Triton.language.html*

[7] Triton-Shared Repository *https://github.com/microsoft/Triton-shared*

[8] Triton-Linalg Repository *https://github.com/Cambricon/Triton-linalg*

[9] AI-Benchmark Repository *https://github.com/Terapines/AI-Benchmark*

