Huawei Summer Intership: Enabling SVE/SME on triton-cpu

Daniel Antonio Martínez Sánchez
Compiler Engineer Intern (OSPP)
Huawei

September 24, 2025

Overview

Goal 🏁

- Enable the new SVE and SME ARM vector extensions into the open euler triton-cpu 🍃

+ SVE/SME

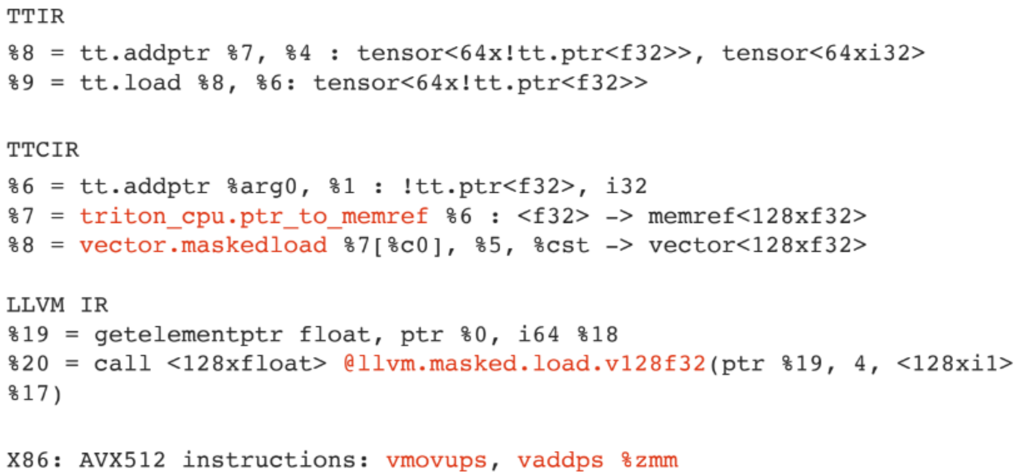
Approach 💡

So, how do we do that?

- We can just hardcode the intrinsics for relevant ops 😮‍💨
- But we have some really nice SVE/SME work already done
    - The team at Cambridge already has really good performance with SVE/SME
    - Why don't just use that in triton?
    - That's a challenge because triton is not really easy to handle in this aspect

The issue with triton 🚩

Triton Uses it's own IR almost entirely



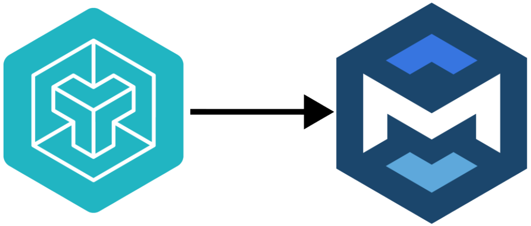
```
TTIR
%8 = tt.addptr %7, %4 : tensor<64x!tt.ptr<f32>>, tensor<64xi32>
%9 = tt.load %8, %6: tensor<64x!tt.ptr<f32>>

TTCIR
%6 = tt.addptr %arg0, %1 : !tt.ptr<f32>, i32
%7 = triton_cpu.ptr_to_memref %6 : <f32> -> memref<128xf32>
%8 = vector.maskedload %7[%c0], %5, %cst -> vector<128xf32>

LLVM IR
%19 = getelementptr float, ptr %0, i64 %18
%20 = call <128xfloat> @llvm.masked.load.v128f32(ptr %19, 4, <128xi1>
%17)

X86: AVX512 instructions: vmovups, vaddps %zmm
```

The issue with triton 🚩

- But on Cambridge they use standard MLIR...
- is there any way to make it work? 🤔

Triton-shared 💱

- Triton shared is a middle layer between triton and MLIR
- It translates from the triton dialects to standard MLIR
- This enables us to still use triton but have all of the advantages of MLIR!



Triton-shared 💱

```
tt.func @kernel(%afloat : !tt.ptr<bf16>, %res : !tt.ptr<bf16>) {
  %0 = tt.make_range {end = 128 : i32, start = 0 : i32} : tensor<128xi32>
  %1 = tt.splat %afloat : (!tt.ptr<bf16>) -> tensor<128x!tt.ptr<bf16>>
  %2 = tt.addptr %1, %0 : tensor<128x!tt.ptr<bf16>>, tensor<128xi32>
  %afm = tt.load %2 : tensor<128x!tt.ptr<bf16>>
  %3 = "tt.reduce"(%afm) ({
  ^bb0(%arg5: bf16, %arg6: bf16):
    %21 = arith.addf %arg5, %arg6 : bf16
    tt.reduce.return %21 : bf16
  }) {axis = 0 : i32} : (tensor<128xbf16>) -> bf16
  tt.store %res, %3 : !tt.ptr<bf16>
  tt.return
}
```

Triton-shared 💱

```
func.func @kernel(%arg0: memref<*xbf16>, %arg1: memref<*xbf16>, %arg2: i32, %arg3: i32, %arg4: i32) {
    %cst = arith.constant 0.000000e+00 : f32
    %reinterpret_cast = memref.reinterpret_cast %arg0 to offset: [0], sizes: [128], strides: [1] :
        memref<*xbf16> to memref<128xbf16, strided<[1]>>
    %alloc = memref.alloc() : memref<128xbf16>
    memref.copy %reinterpret_cast, %alloc : memref<128xbf16, strided<[1]>> to memref<128xbf16>
    %0 = bufferization.to_tensor %alloc restrict writable : memref<128xbf16>
    %1 = bufferization.alloc_tensor() : tensor<f32>
    %inserted = tensor.insert %cst into %1[] : tensor<f32>
    %reduced = linalg.reduce ins(%0 : tensor<128xbf16>) outs(%inserted : tensor<f32>) dimensions = [0]
      (%in: bf16, %init: f32) {
        %3 = arith.extf %in : bf16 to f32
        %4 = arith.addf %3, %init : f32
        linalg.yield %4 : f32
      }
    %extracted = tensor.extract %reduced[] : tensor<f32>
    %2 = arith.truncf %extracted : f32 to bf16
    %reinterpret_cast_0 = memref.reinterpret_cast %arg1 to offset: [0], sizes: [1], strides: [1] :
        memref<*xbf16> to memref<1xbf16, strided<[1]>>
    affine.store %2, %reinterpret_cast_0[0] : memref<1xbf16, strided<[1]>>
    return

}
```

Triton-shared 💱

How good is triton-shared?

- Performance
    - Triton itself is a DSL designed for GPU's ❗
    - Performance can get really compromised depending on the source triton kernel.
- Reliability
    - Triton-shared is an experimental project 🧪
    - It's under active development

Performance 🏎️

Example of bad code: ❌

```
for m in range(0, M, BLOCK_SIZE_M):
  # Do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

Performance 🏎️

Example of performant code:✔️

```python
def bare_matmul(X, Y, Z, M, N, K, BLOCK_SIZE: tl.constexpr):
    pid_x = tl.program_id(0)  # block row id
    pid_y = tl.program_id(1)  # block column id

    offs_x = pid_x * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    offs_y = pid_y * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)

    x = tl.load(X + offs_x[:, None] * K + offs_y[None, :])
    y = tl.load(Y + offs_x[:, None] * N + offs_y[None, :])

    z = tl.dot(x, y)

    tl.store(Z + offs_x[:, None] * N + offs_y[None, :], z)
```

Reliability 🧱

At the start of the project we had ∼5000 failing test (out of 6658) 😱:

- Unsupported datatypes (fp8)
- Unsupported triton operations
- Lots of LLVM bugs

Reliability 🧱

A quick summary of the new operations supported:

- Reductions with complex bodies (`tt.reduce`)
- Scans (`tt.scan`)
- Bitcasts (`tt.bitcast`)
- Atomic read, modify, write (`tt.atomic_rmw`)
- Batched matmul (`tt.dot` with 3d inputs)

Reliability 🧱

And a quick summary of LLVM bugs and patches

- Crash when running SME on transforms dialect
- Add support for f8e5m2 & f8e4m3fn in arith `expands-ops`
- Linalg missing `RecursiveMemoryEffects`
- Crash causes by liveness analysis in `remove-dead-values`
- New SVE/SME pipelines

Failing test ⛔

With all of those fixes and additions we have just...

```
==== 96 failed, 6562 passed ====
```

96 failing test left 🤩.

SVE/SME 🧑‍💻

What about the SME/SVE optimized pipelines?

- Pipelines had to be open-sourced from the Cambridge team 🔓
- SME pipeline was open sourced early and his already ported to triton
- SVE pipeline was open sourced recently and there is still work todo

Future work ⏳

- Finish porting SVE pipeline 🧑‍💻
- Fix remaining failing test 🛠️
- Benchmark performance 📏

Documentation 📄

I wrote some documentation about this:

- Project log here
- Reproduction instructions here