Optimizing Triton for RISC-V heterogenous AI computing

Aries Wu

Terapines (Wuhan) Technology

兆松科技
Tera Pines

RISC-V

Agenda

01 | Intro to Triton

02 | Heterogeneous SoC

03 | Code Generation

04 | Optimization

What is Triton DSL?

- Open-source GPU programming language for neural networks originally designed by OpenAI
- Adopted by many other chip vendors
- DSL embedded in Python
- Performance is on par with CUDA on NVDIA GPUs
- Block level programming language

What is a block level language?

```
@jit
def add(X, Y, Z, N):
    pid = program_id(0)
    idx= pid * N + arange(N)
    mask = idx < N
    x = load(X + idx, mask=mask)
    y = load(Y + idx, mask=mask)
    store(Z + idx, x + y, mask=mask)
```

* N - block size

Triton - block level

```
void add(int* x, int* y, int* z) {
    tid = threadIdx.x;
    z[tid] = x[tid] + y[tid];
}
```

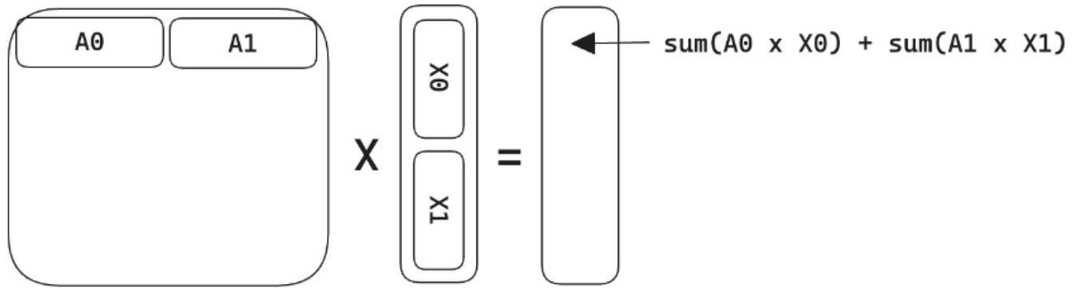
CUDA – thread level

```
void add(int* x, int* y, int* z, int N) {
    for (int i = 0; i < N; i++)
        z[i] = x[i] + y[i];
}
```

C – whole data

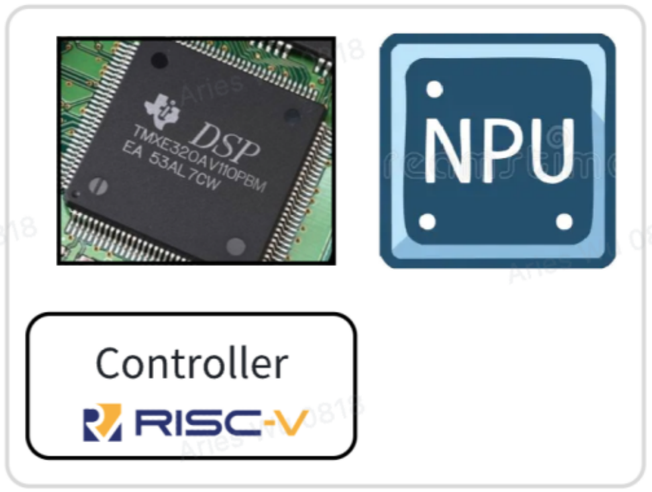
Matrix-Vector Multiplication in Triton

```
# Y = A @ X, where A is a matrix of M x N, X is a vector of N.
@triton.jit
def gemv_kernel(Y, A, X, M, N, stride_am, BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr):
    start_m = tl.program_id(0)
    rm = start_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    rn = tl.arange(0, BLOCK_SIZE_N)

    A = A + (rm[:, None] * stride_am + rn[None, :])
    X = X + rn

    acc = tl.zeros((BLOCK_SIZE_M, ), dtype=tl.float32)
    for n in range(N, 0, -BLOCK_SIZE_N):
        a = tl.load(A)
        x = tl.load(X)
        acc += tl.sum(a * x[None, :], axis=1)
        A += BLOCK_SIZE_N
        X += BLOCK_SIZE_N

    Y = Y + rm
    tl.store(Y, acc)
```

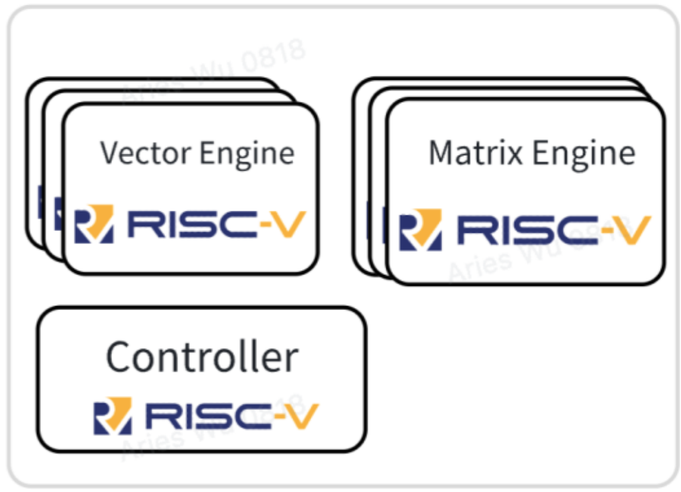


Typical RISC-V based DSAs

RISC-V Controller + NPU

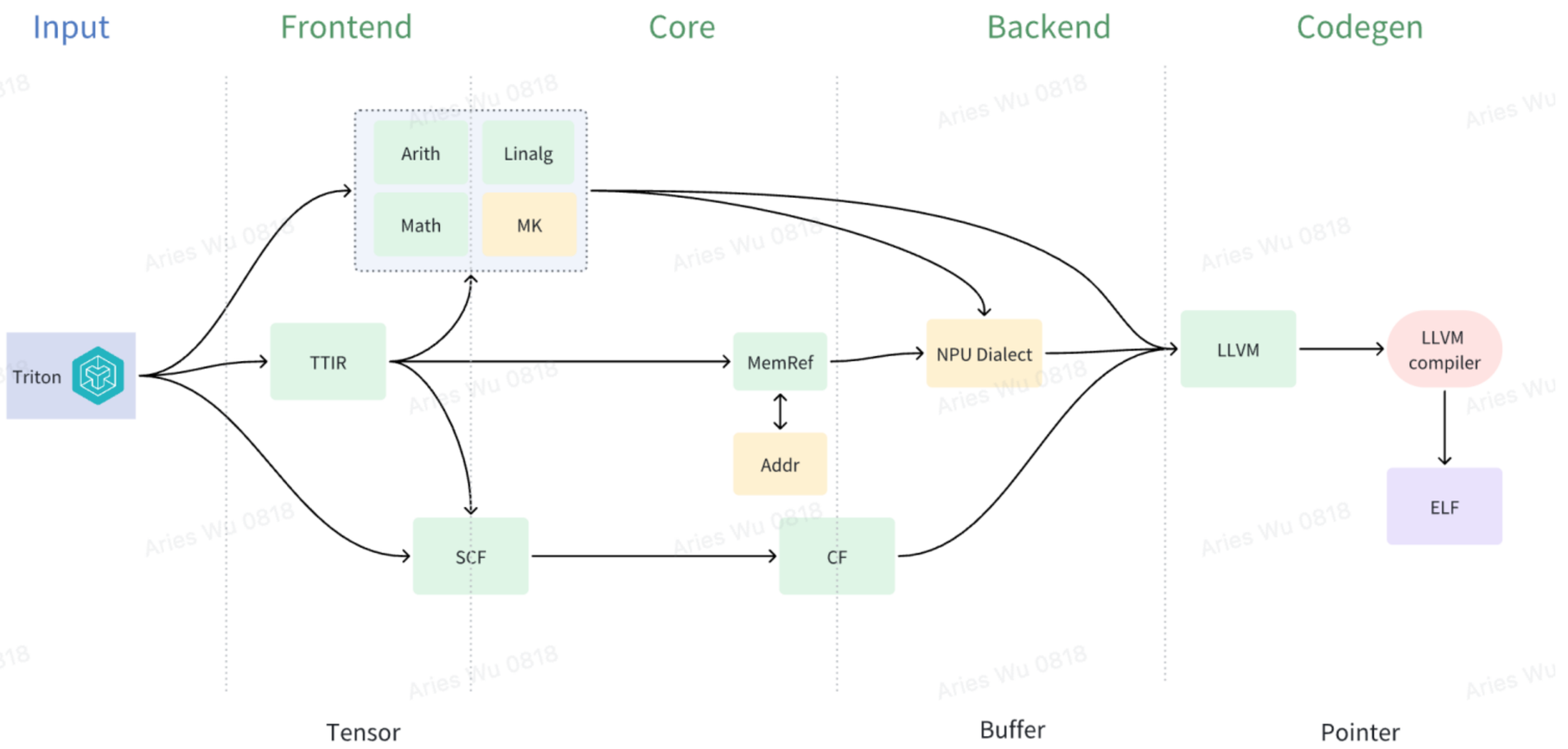


RISC-V Controller + NPU + DSP



RISC-V + RVV + AME/IME/…

A Triton Compiler Stack for RISC-V + NPU



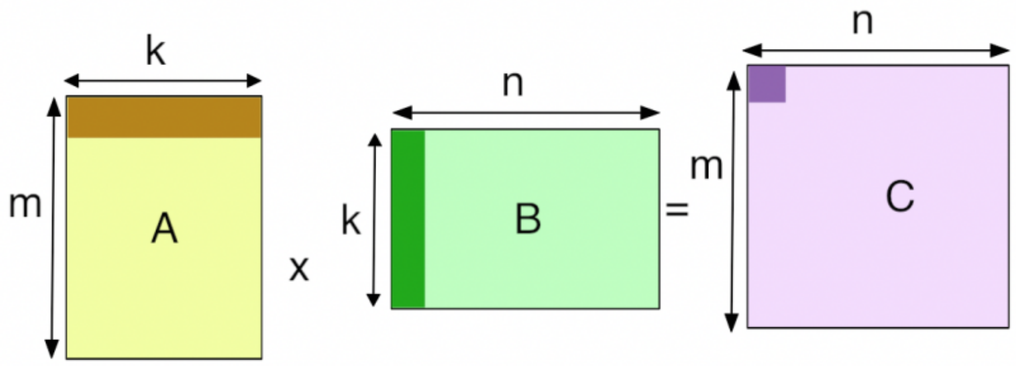
Our MLIR Triton Compiler

RISC-V + NPU Optimization Strategies

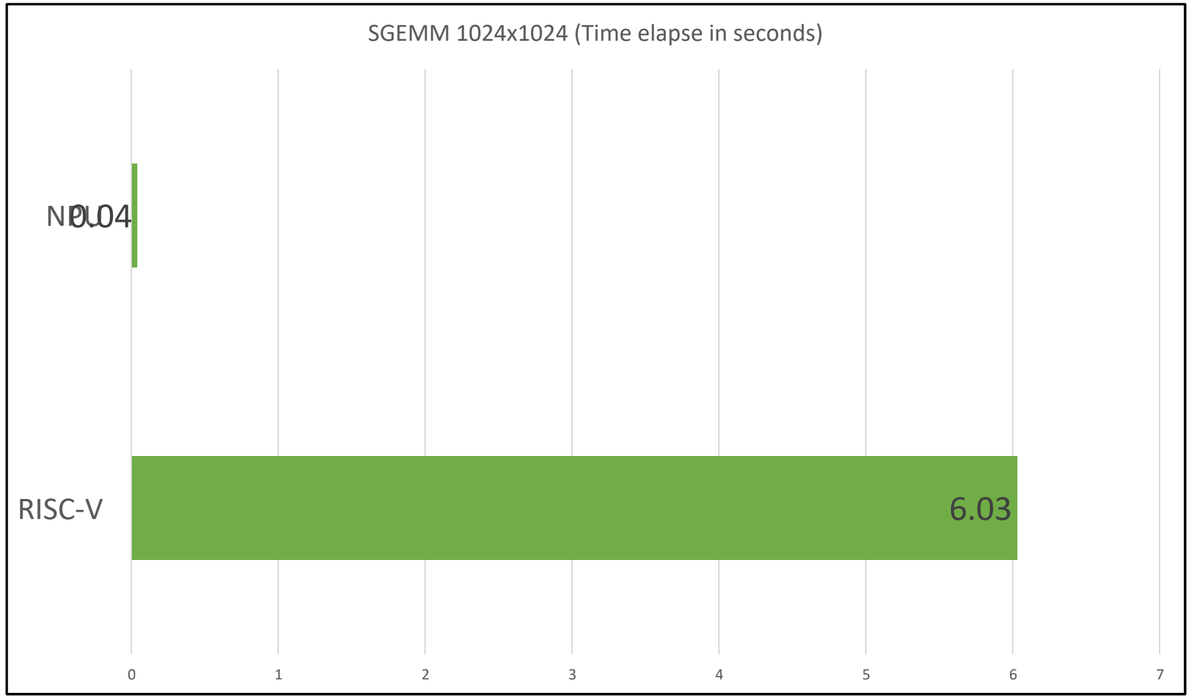
- Recognize coarse-grain operators which can be mapped onto NPU
- Operator fuse and decompose
- Fallback unsupported NPU operators onto RISC-V
- Generate async DMA data fetch for NPU
- Generate structured memory access
- Auto tune block size
  - Balance between memory bandwidth and compute power
  - Fit limited on-chip SRAM
- libdevice extensions for NPU

Benchmarking the SGEMM Kernel

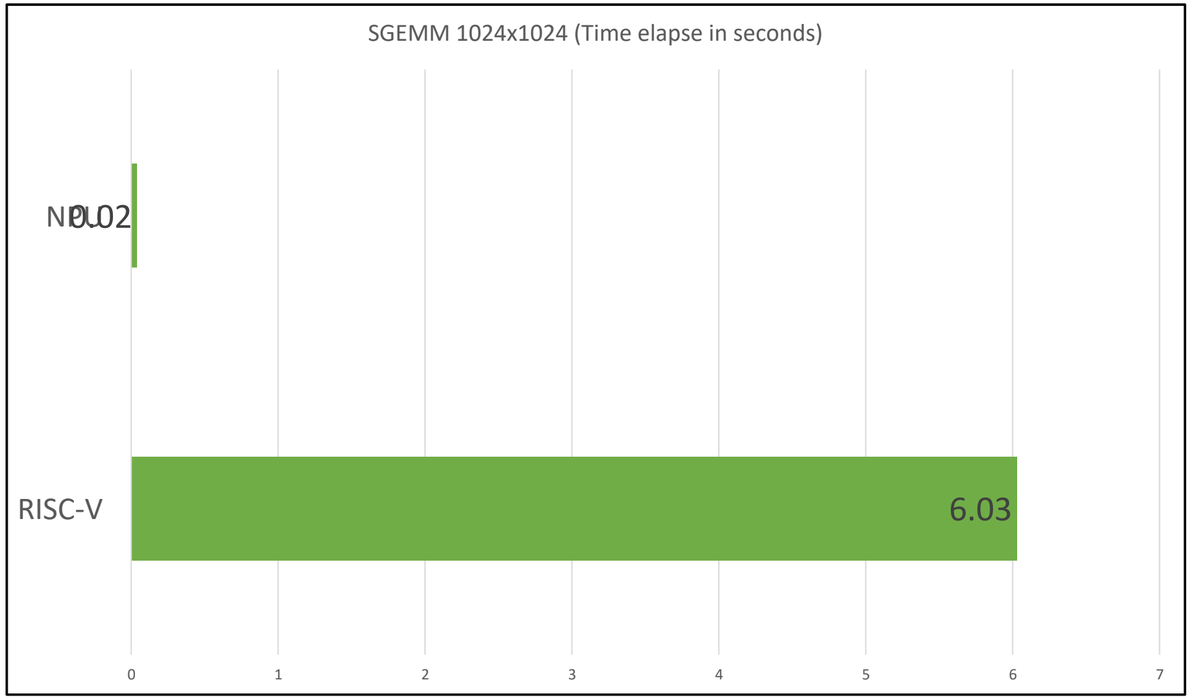
```
……
# -----------------------------------------------------------
# Iterate to compute a block of the C matrix.
# We accumulate into a `[BLOCK_SIZE_M, BLOCK_SIZE_N]` block
# of fp32 values for higher accuracy.
# `accumulator` will be converted back to fp16 after the loop.
accumulator = tl.zeros((BLOCK_SIZE_M, 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

# You can fuse arbitrary activation functions here
# while the accumulator is still in FP32!
if ACTIVATION == "leaky_relu":
    accumulator = leaky_relu(accumulator)
c = accumulator.to(tl.float32)
……
```



SGEMM kernel performance

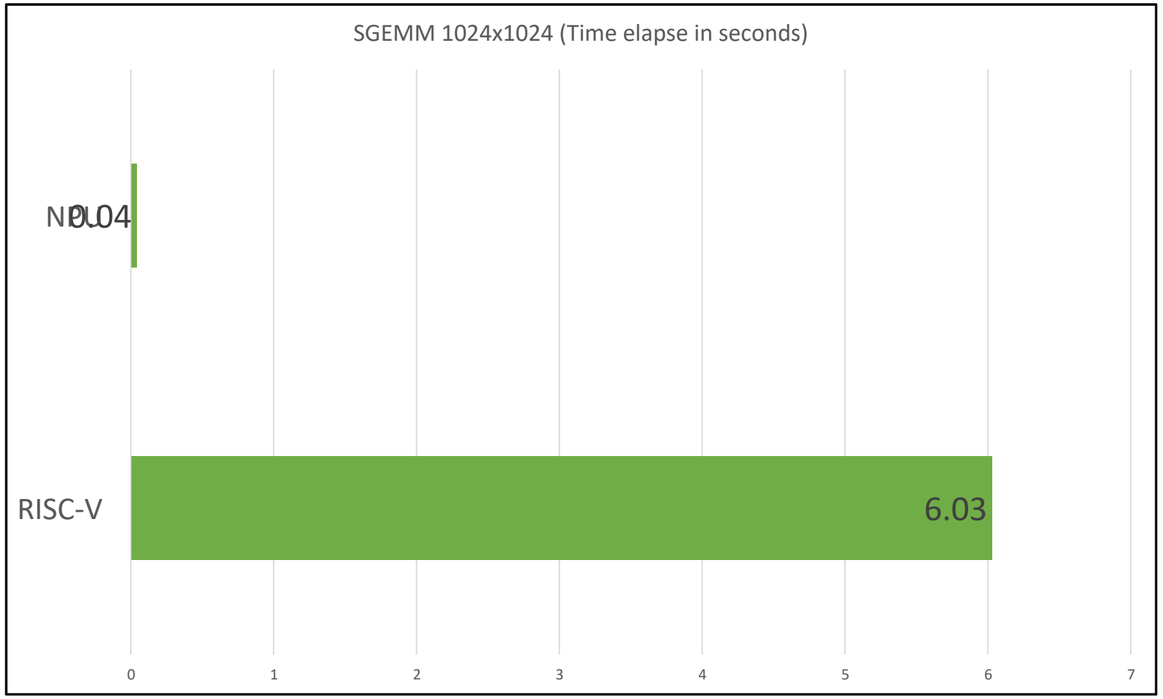


Block size 128x128x32

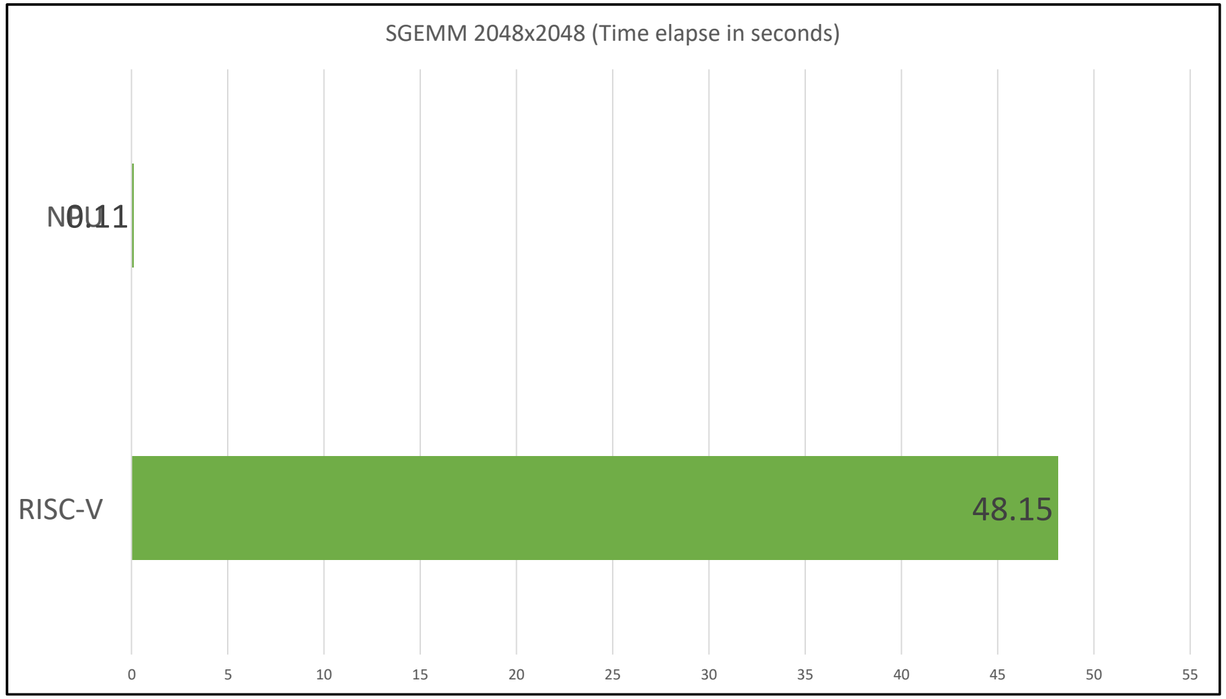


Block size 256x256x256

SGEMM kernel performance



Block size 256x256x256



Block size 256x256x256

Summary

- Triton kernels can achieve on par performance compares to C kernels on non-GPU platform
- Easy to write ML kernels – Python DSL
- Easier to optimize the performance thanks to MLIR compiler infrastructure
- Auto tune block size can maximize the NPU utilization
  - Tiling is left to user instead of compiler
  - The key optimization for ML workload

兆松科技
Tera Pines

Thanks