Triton CPU

Its Approach and Performance Studies on x86/ARM

Minjang Kim, Meta

Ilya Enkovich, Intel

Agenda

- Minjang
  - Overview of the approach
  - GEMV optimizations on ARM
- Ilya
  - Performance deep dives on x86
  - Summary and future work

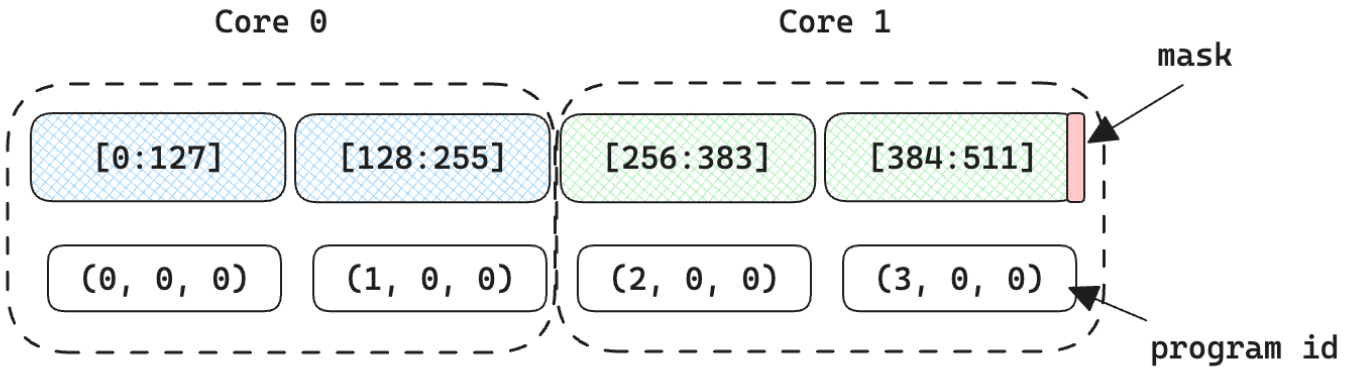
Why Triton CPU?

- OSS demands for computers without GPU
- Internal business demands
  - Small batch jobs, PyTorch for Edge (20% efforts for 80% perf)

- Intel/Meta started Triton CPU as a fork of Triton
- Feature completeness as the first milestone

CPU Execution Model

- HW parallelism: Multi-core + SIMD
- A vector of 511 elements, BLOCK = 128 (elements)
- Each program id maps to a CPU thread



Overview of Our Approach

- SIMD: Using the vector MLIR dialect as the initial approach
  - Converting TTIR types/ops to vector-dialect types/ops
  - LLVM backend with auto-vectorizer generates AVX512 and Neon (ARM)

- We can generate hand-vectorized code for special cases
  - E.g., exploiting Neon's special instruction for BF16

- Multi-core: Using OpenMP with a simple static scheduling

Compiler Overview (third_party/cpu/backend/compiler.py)

- `make_ttir`: high-level optimizations (same with GPU)
- `make_ttcir`: TTIR → TTCIR, lowering to vector dialect
- `make_tttcir`: TTCIR → Target-specific TTCIR (TTTCIR)
- `make_llir`: Target-specific TTCIR → LLVM IR
- `make_asm`: LLVM IR to .asm (static compilation)
- `make_so`: Compile .asm into .so
- The CPU driver and OpenMP launcher

Artifact Examples: vector-add

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

Feature support status

- Most of Triton features are supported for CPU
  - Some of FP8 types are not supported
  - Debug operations (print, assert) functionality is limited
- Compilation pipeline is target-independent, works on x86 and ARM
- Unit test pass rate > 97%, all Triton tutorials work on CPU
- Triton CPU as a Torch Inductor (draft)
  - Torchbench pass rate: 51%
  - Huggingface pass rate: 93%
  - TIMM models pass rate: 15%
  - Almost all failures are timeouts due to long compilation time

Optimizing GEMV (Matrix x Vector)

- The pattern: a reduction + elementwise multiplications

```
acc += tl.sum(a * x[None, :], axis=1)
```

- Enable fast-math optimizations

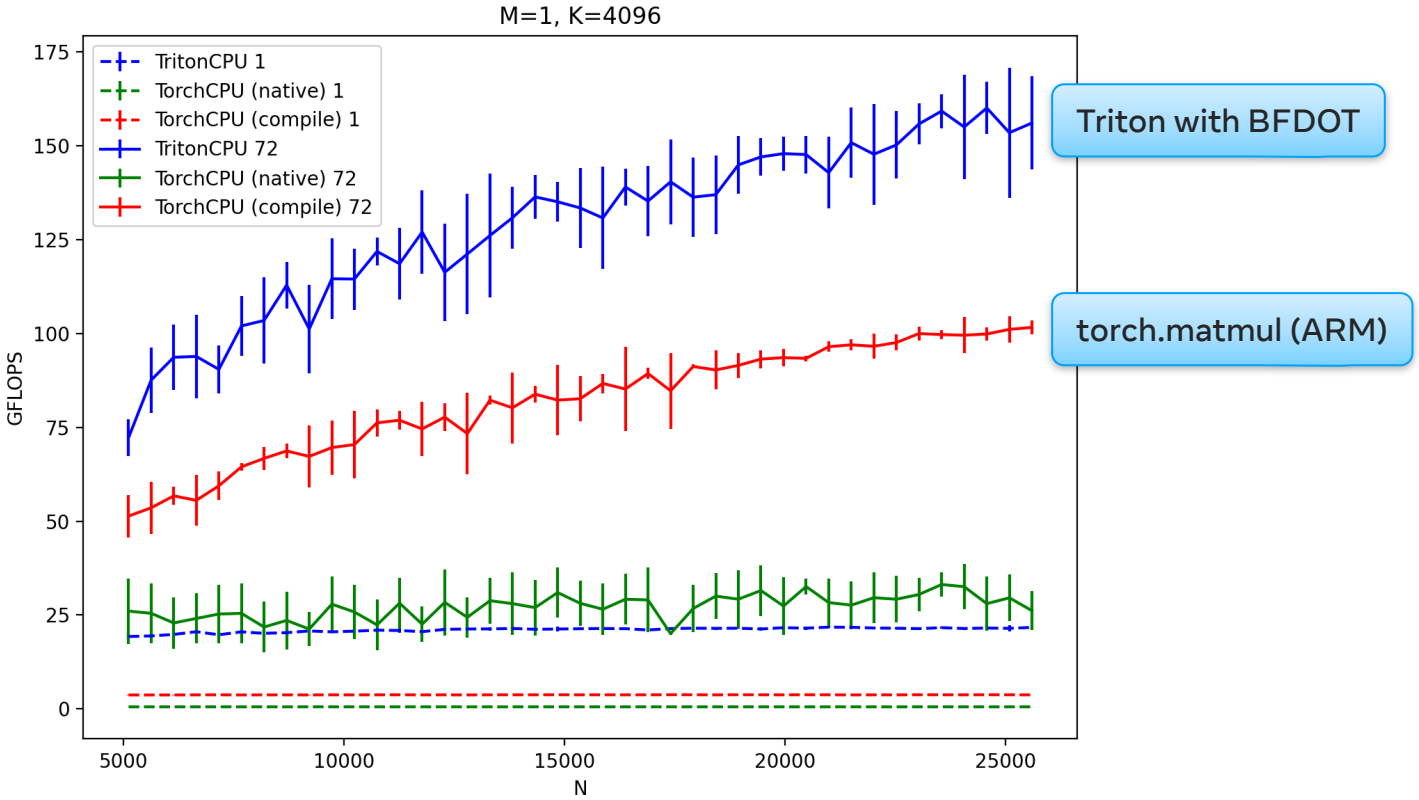
```
%27 = load <512 x float>, ptr %26
%28 = fmul <512 x float> %25, %27
%29 = call reassoc float @llvm.vector.reduce.fadd.v512f32(0., %28)

vfmadd231ps 0x300(%rdx),%zmm4,%zmm3 # fused multiply-add
```

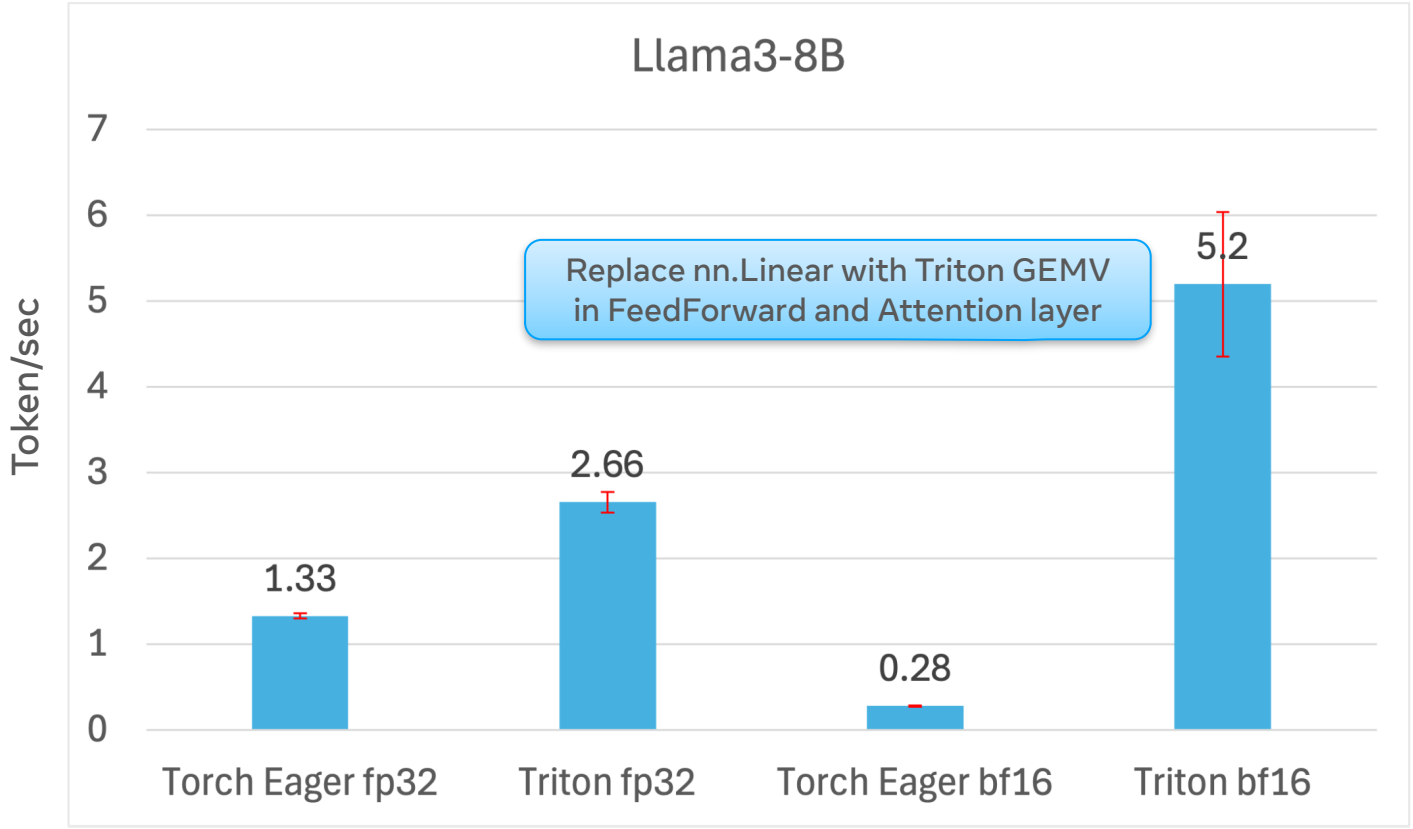
Optimizing BF16 GEMV for ARM

- By default, `tt.reduce` is converted to `vector.multi_reduction`, and then to a machine-independent intrinsic
- `BFDOT`: `BF16 dot product instruction for ARM`
- Introduce a special optimization for the GEMV reduction pattern
- Emits `llvm.aarch64.neon.bfdot.v4f32.v8bf16`
- Assembly code gets `bfdot`
- Measured on Neoverse ARM 72-core (GH200)

Performance: TritonCPU vs. torch.matmul



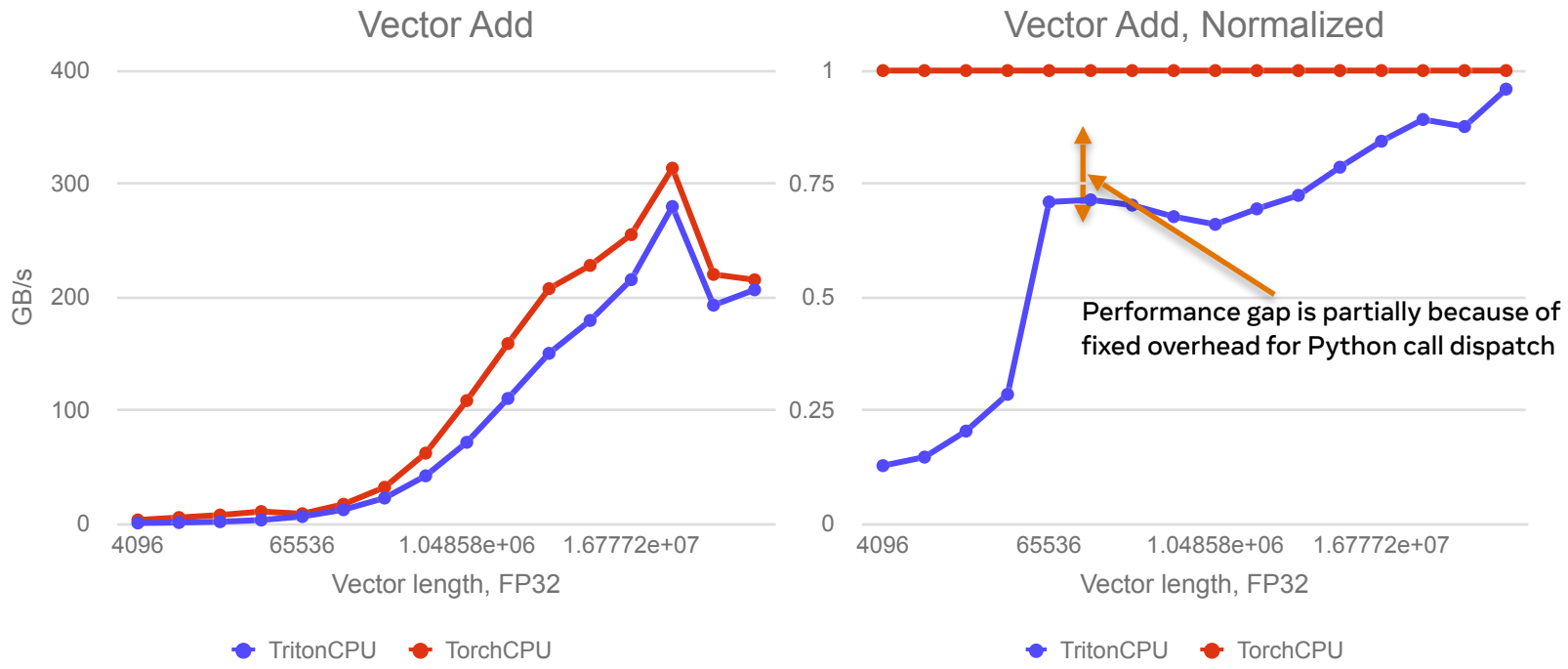
Transformer Decoder with BFDOT



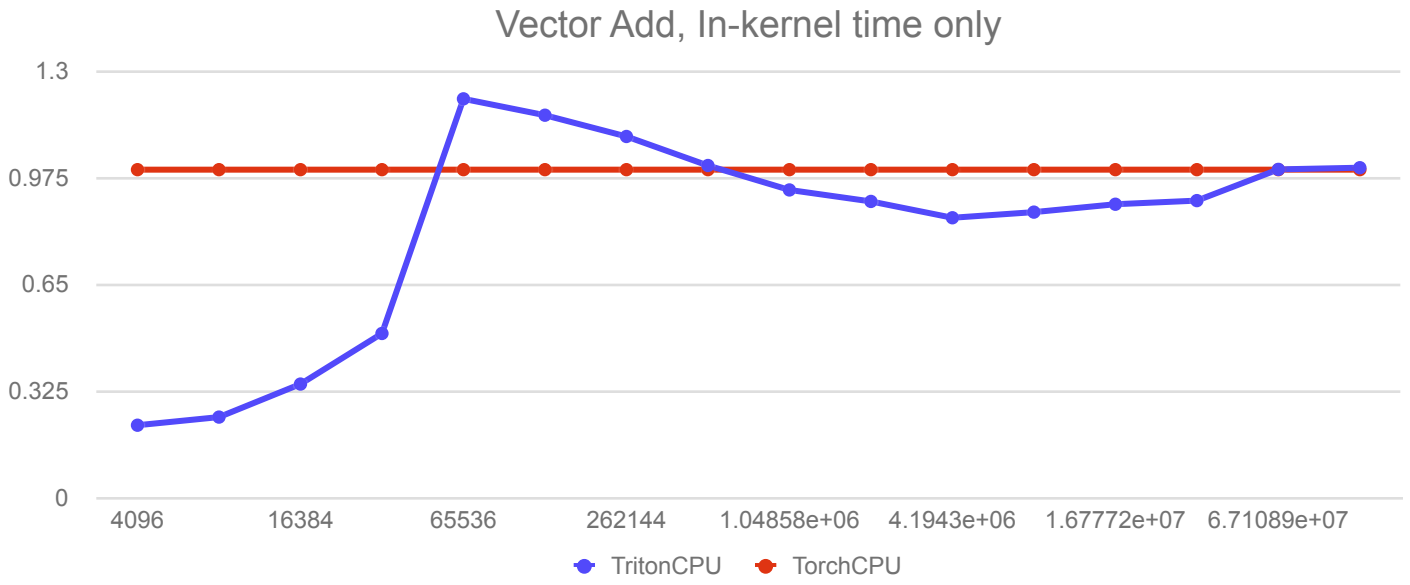
Performance optimizations

- Vectorized math functions support
  - By default, use SLEEF library for math
  - There is an option to use libmvec instead
- Optimization of masked loads/stores to remove mask
  - Works for tiled loops, based on divisibility hints to prove mask is all-ones
- Specialized target-specific lowering patterns
- No GEMM related optimizations yet

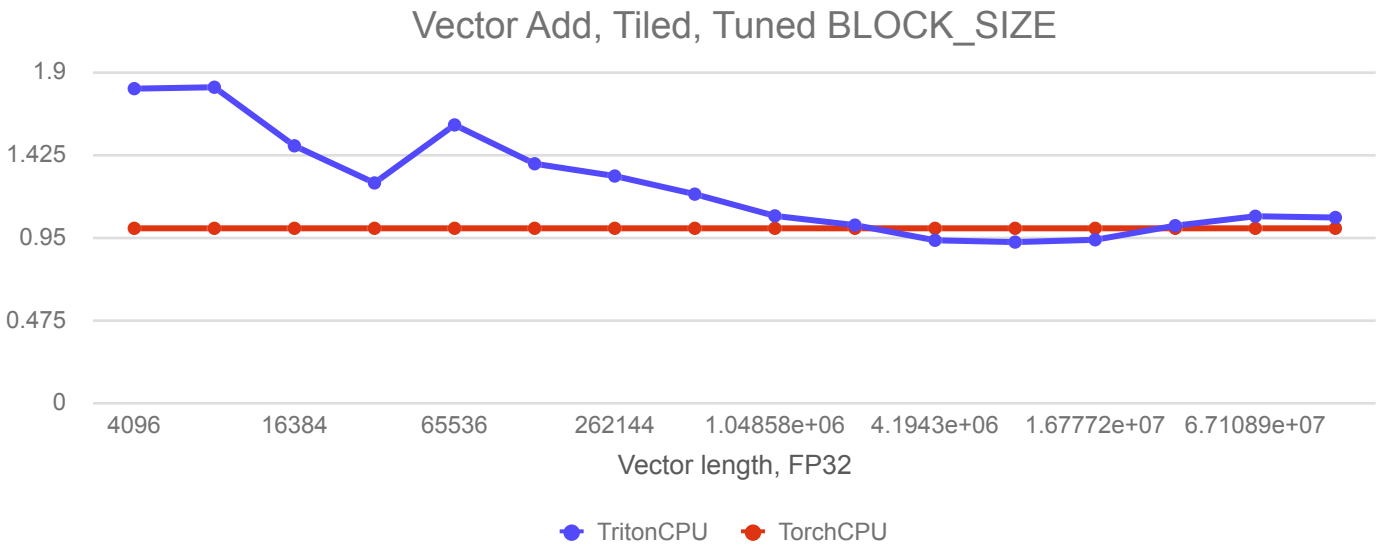
Vector-add: Original kernel



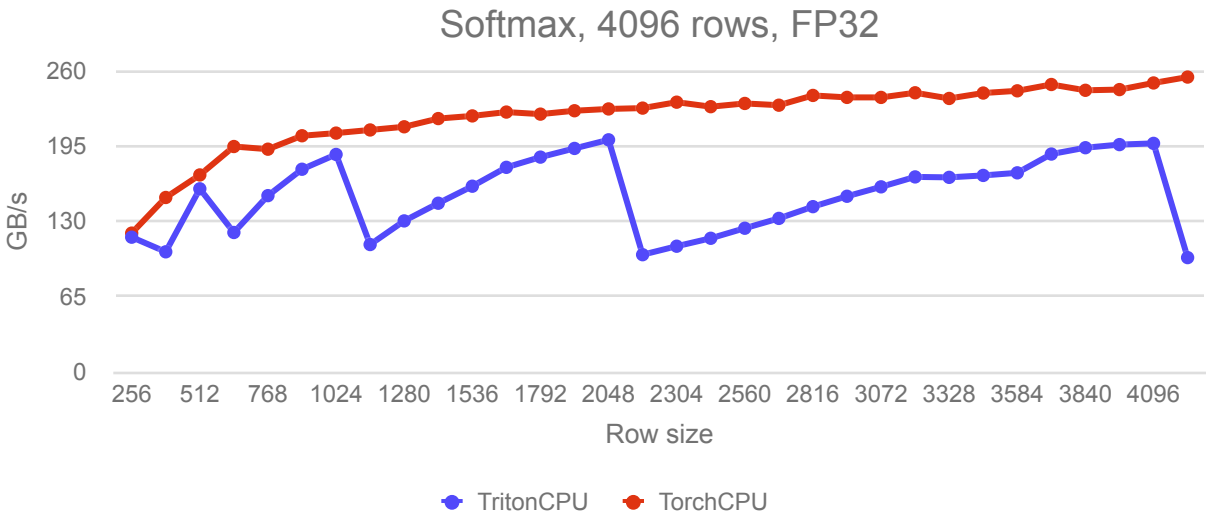
Vector-add: Pure kernel performance



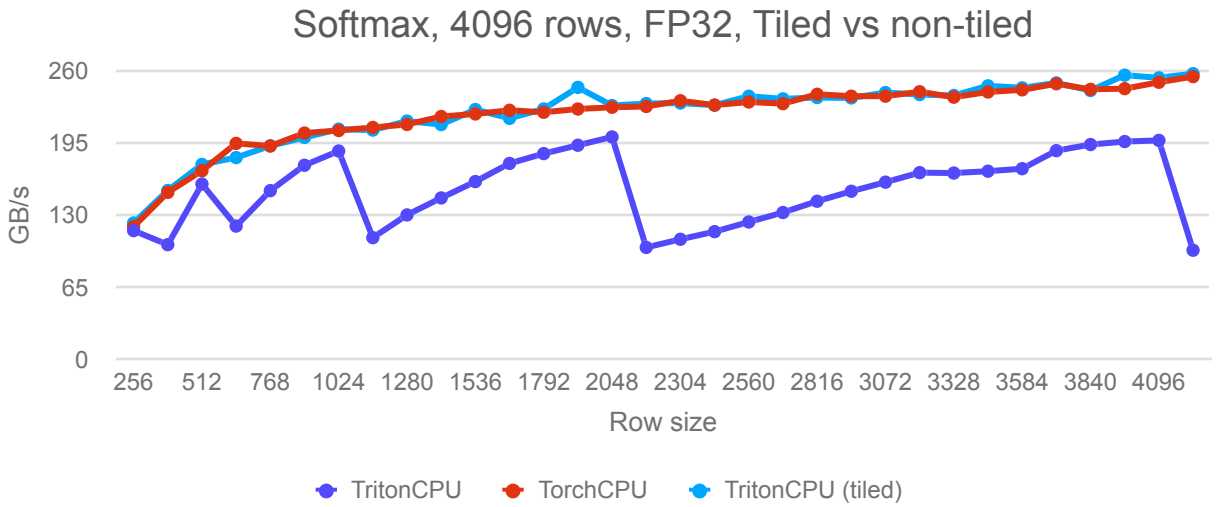
Vector-add: Tiling + BLOCK_SIZE tuning



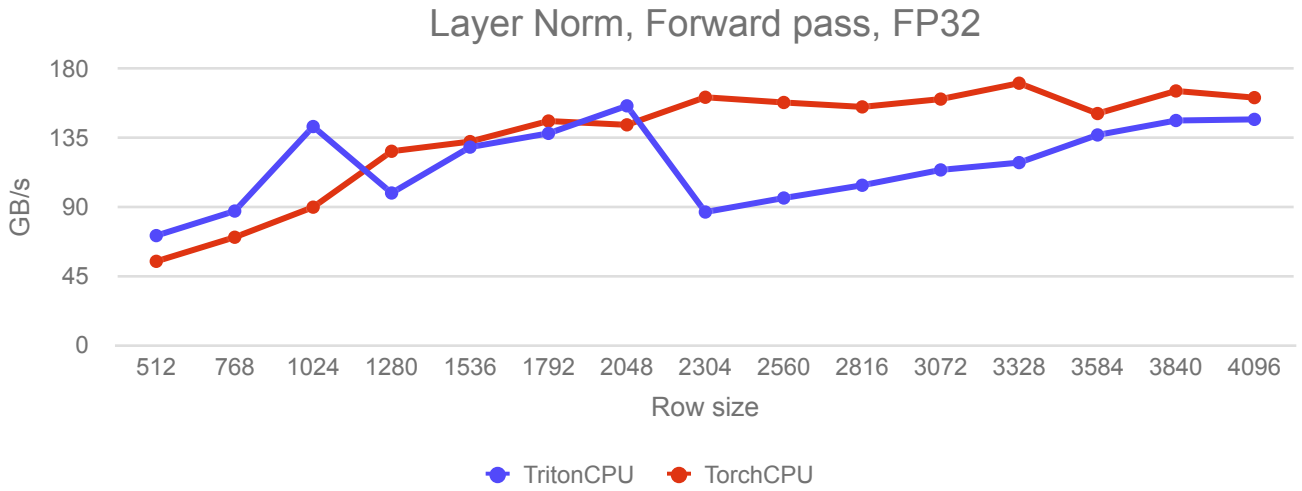
Softmax: Original kernel



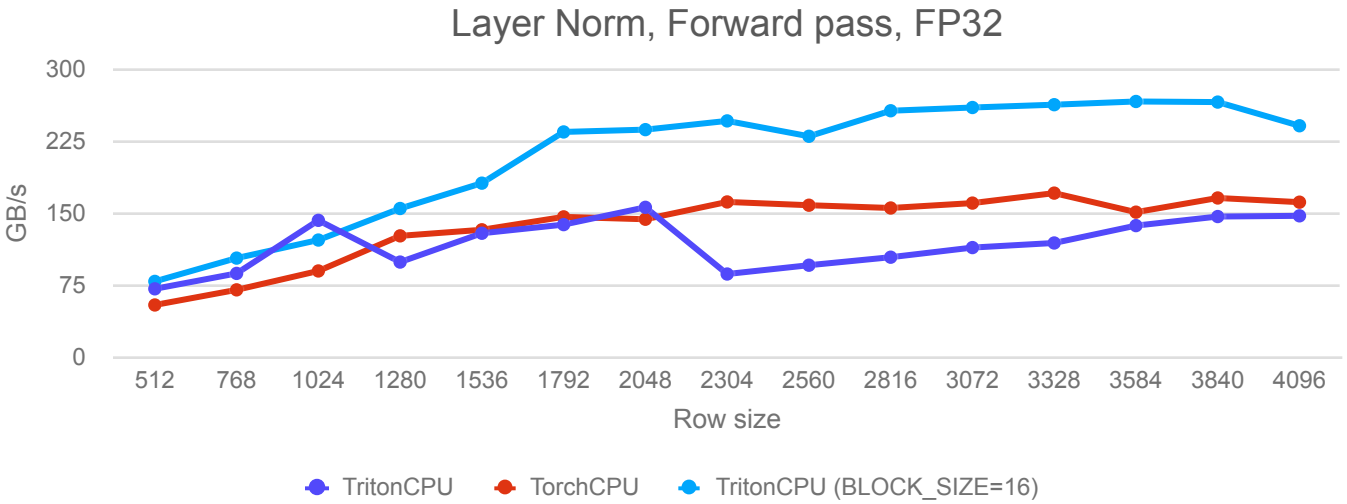
Softmax: Tiled kernel



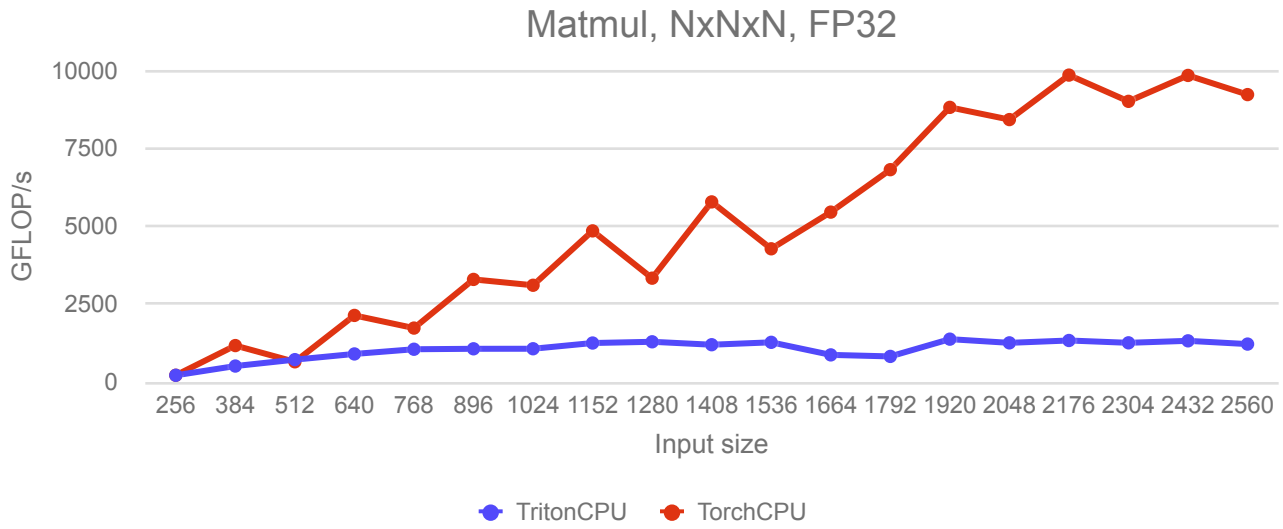
Layer Norm: Original kernel



Layer Norm: Original kernel + BLOCK_SIZE=16



Matmul


Matmul, NxNxN, FP32
GFLOP/s
10000
7500
5000
2500
0
256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048 2176 2304 2432 2560
Input size
TritonCPU
TorchCPU

Performance summary

- TritonCPU can match optimized C++ kernels performance on CPU for non-GEMM kernels, but kernel/parameters adjustment is required
- Autotuner needs to be fully enabled for CPU to help with kernels tuning
- GEMM kernels don’t utilize all available HW capabilities (e.g. AMX), and will need a lot of work in the compiler

Future work

- Complete the basic testing coverage
- Improving Autotuning for TritonCPU
- Testing with important kernels (e.g., GEMV, FA3), filling more gaps
- Utilize more HW extensions (e.g., AMX, SME)
- Setting up (internal) benchmarks and performance
- Improve compilation time
- Contributors are welcomed!

Repo: `https://github.com/triton-lang/triton-cpu`

#third-party channel in Slack: `https://triton-lang.slack.com/`

Thank you!