SGLang x Wave

The Wave Team

AMD
together we advance_

Overview

- Brief Introduction to Wave DSL
- Comparison to other DSLs
- Wave kernels in SGLang
- Vanilla Attention Example
- Performance Results
- Coming Soon to SGLang

Brief Introduction to Wave DSL

- Experimental DSL for high performance machine learning (less than a year old)
- Key project goals are to enable:
  - **Easy implementation of new algorithms**

    a) Python-based DSL

    b) Leverages MLIR & LLVM for code generation so that don’t need new hand-written implementations for different operators

    c) Has print support (breakpoint support and visualization planned in future releases)

    d) Has support for distributed kernels (single-node)

  - **Quick turnaround on performance optimizations**
  - a) Exposes shared memory, MMA variants, auto-tuning knobs
  - b) Compiler supports range of optimizations
  - c) Lower-level user control can also be exposed

  - Initial focus on GEMM, Attention & Convolutions
  - Currently supports CDNA GPUs (MI25x, MI30x, MI35x)

Comparison to other DSLs

	Wave DSL	CuTe DSL
DSL Language	Python	Python
Layout Control	Symbolic Expressions	Layout Algebra
GPU Partitioning Strategy	Explicit through symbolic variables & constraints	Implicit in Kernel
Performance Optimizations	Compiler optimizations & User-specified	User-specified
Debugging Support	Print supported	Print supported
Distributed Kernels	Support for single-node	None

Wave Kernels in SGLang

- **Thank you to Sglang team & Hai for your continued support!**
- Recently merged (https://github.com/sgl-project/sglang/pull/8660)
- Enable by using –attention-backend=wave
- Currently supporting following attention types:
  - Prefill Attention
  - Decode Attention
  - Extend Attention

```python
@wave(constraints)
def attention(
   query: Memory[B, M, H, K1, GLOBAL_ADDRESS_SPACE, f16],
   key: Memory[B, K2, H, K1, SHARED_ADDRESS_SPACE, f16],
   value: Memory[B, K2, H, N, SHARED_ADDRESS_SPACE, f16],
   output: Memory[B, M, H, N, GLOBAL_ADDRESS_SPACE, f32],
):
```

Inputs are specified in terms of symbols and wave decorator

```python
constraints = [WorkgroupConstraint(M, BLOCK_M, 0),
               WorkgroupConstraint(N, BLOCK_N, 1),
               WorkgroupConstraint(B, BLOCK_B, 2),
               WorkgroupConstraint(H, BLOCK_H, 3),
               TilingConstraint(K2, BLOCK_K2),
               WaveConstraint(M, BLOCK_M / num_waves),
               WaveConstraint(N, BLOCK_N / 1),
               HardwareConstraint(threads_per_wave=64,
                 mma_type=MMAType.F32_16x16x16_F16,
                 vector_shapes={B:0, H:0})]
```

Distribution strategy specified by constraints on symbolic dimensions

```python
c_reg = Register[B, N, M, f32](0.0)
init_sum = Register[B, M, f32](0.0)
init_max = Register[B, M, f32](-1e6)
qk_scaling = Register[B, M, K2, f32](scale)

@iterate(K2, init_args=[init_max, init_sum, c_reg])
def loop(partial_max: Register[B, M, f32], partial_sum: Register[B, M, f32], acc: Register[B, N, M, f32],):
    imm_reg = Register[B, K2, M, f32](0.0)
    q_reg = read(q)
    k_reg = read(k)
    inner_acc = mma(k_reg, q_reg, imm_reg, mfma_variant[0])
    x_j = permute(inner_acc, target_shape=[B, M, K2])
    x_j *= qk_scaling
    m_j = max(x_j, partial_max, dim=K2)
    e_delta_max = exp2(partial_max - m_j)
    e_delta = exp2(x_j - m_j)
    e_init = partial_sum * e_delta_max
    d_j = sum(e_delta, e_init, dim=K2)
    imm_f16 = cast(e_delta, f16)
    v_reg = read(v, mapping=v_mapping)
    new_acc = acc * e_delta_max
    acc = mma(v_reg, imm_f16, new_acc)
    return m_j, d_j, acc

res_max, res_sum, res_mm = repeat
reciprocal_sum = reciprocal(res_sum)
res = res_mm * reciprocal_sum
write(res, c, mapping=mapping)
```

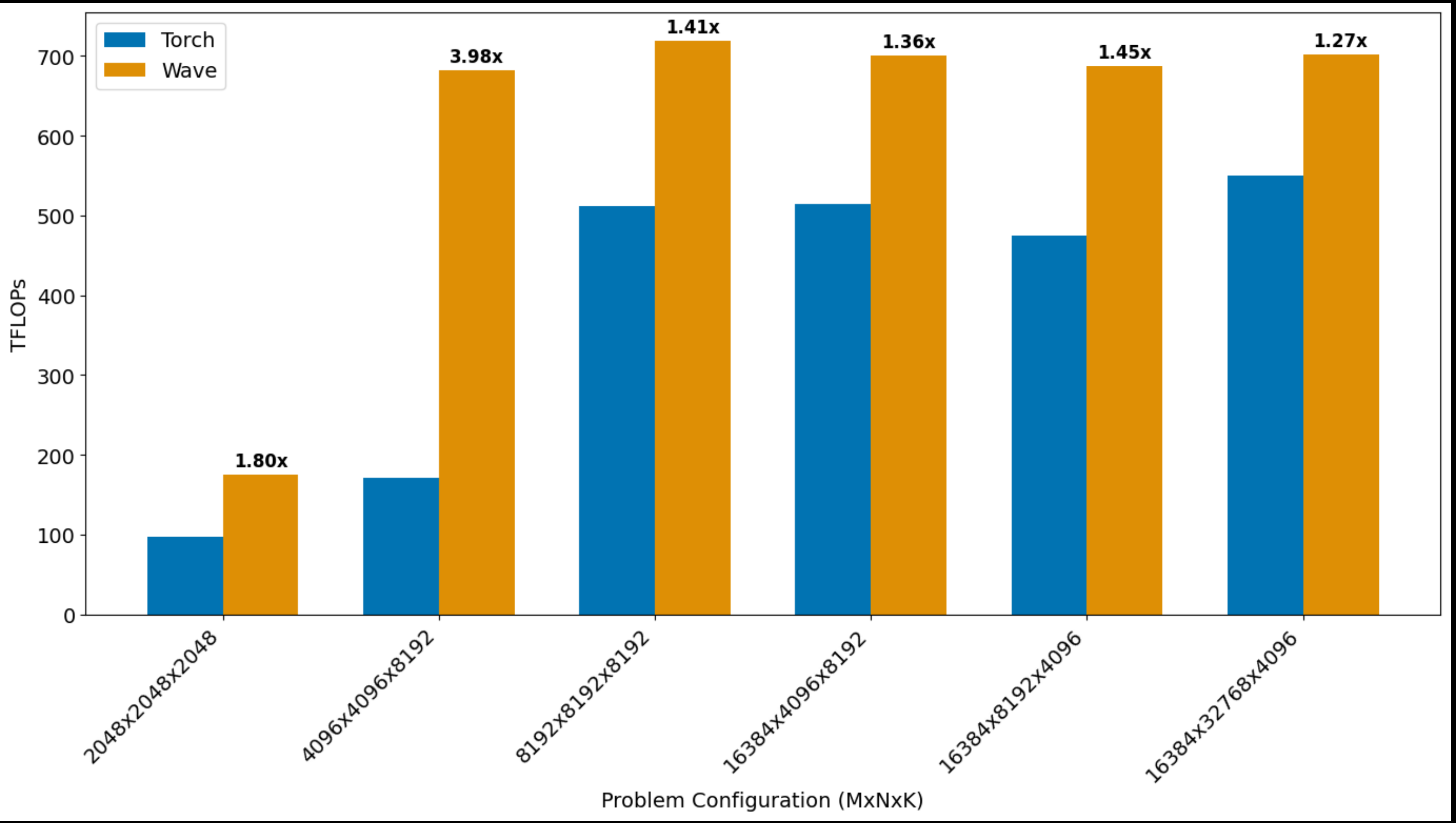
Computation graph expressed without explicit indexing

Flash Attention Example

- Adding new variants
  - Insert into computation graph while maintaining the same distribution strategy
- Performance Tuning
  - Tile sizes are exposed in constraints and can be modified
  - Compiler provides several additional flags for performance such as multi-buffering, scheduling, coalescing loads, swizzling for bank conflicts & more (these can be specified in the WaveCompileOptions before invoking wave_compile. Examples: waves_per_eu which is an LLVM optimization hint, use_fast_math, etc.)
  - If even more control is required, can export MLIR Vector Dialect IR that can be hand-modified and fed back into the compiler for a closed-loop with the profiling tools

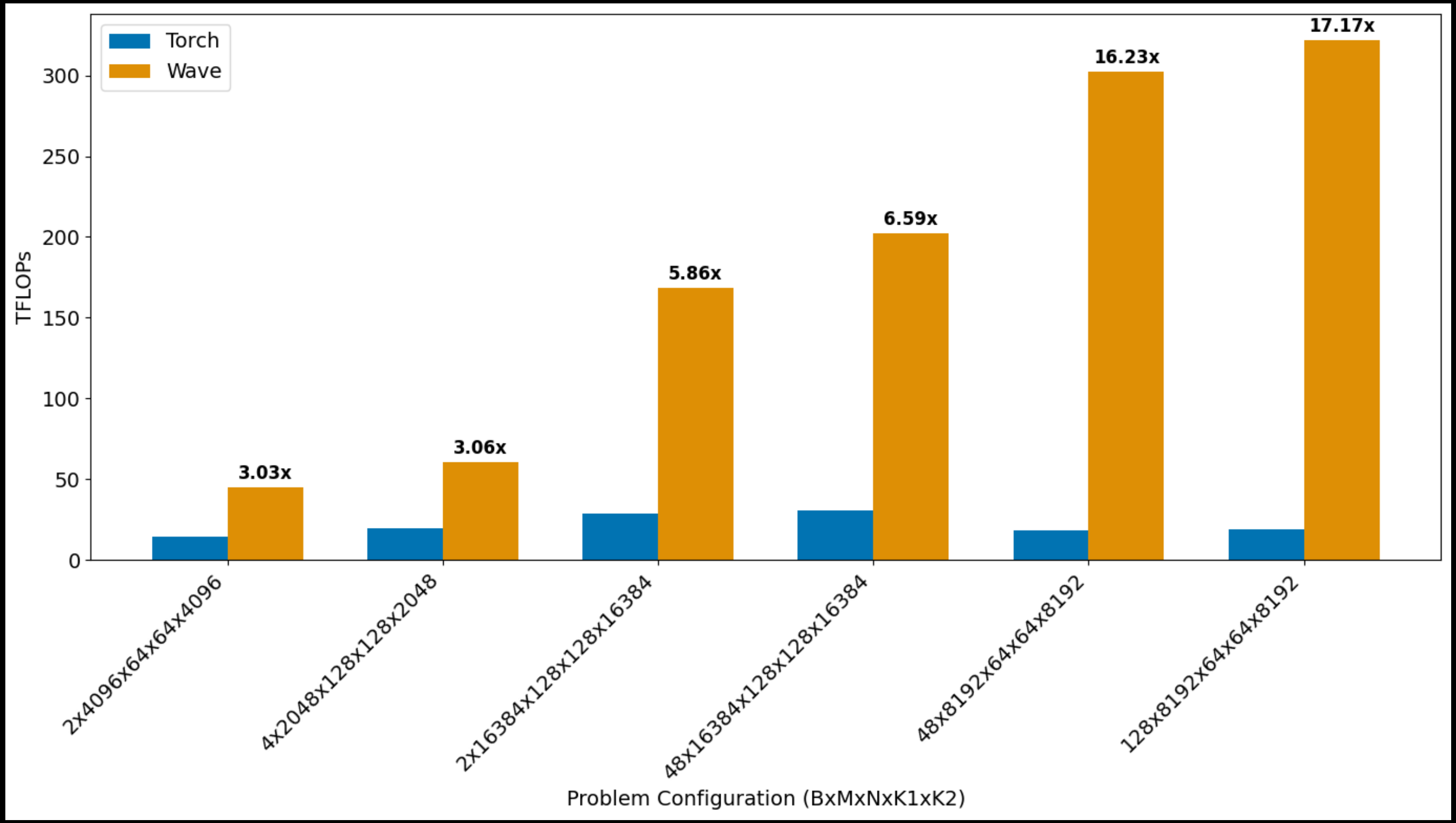
Performance Results – GEMM (TFLOP/s)

- Hardware:
  - AMD Instinct MI300X (gfx942)
- Software:
  - wave-lang v1.0.2
  - torch v2.7.1
  - ROCm user-space version - 6.4.2-120
  - ROCk driver version - 6.8.0-65-generic



Performance Results - Attention

- Vanilla Attention
  - Hardware:
    - AMD Instinct MI300X (gfx942)
  - Software:
    - wave-lang v1.0.2
    - torch v2.7.1
    - ROCm user-space version - 6.4.2-120
    - ROCk driver version - 6.8.0-65-generic



AMD
together we advance_

Coming Soon to SGLang

- Speculative Decoding Kernels
- Mixture of Experts Kernels
- MI35x Performance Optimizations

More Information on Wave DSL

- Github repo: https://github.com/iree-org/wave
- Documentation: https://wave-lang.readthedocs.io/en/latest/wave/wave.html
- Discord: https://discord.gg/VnhYNhujjV

AMD
together we advance_

AMD