Vulkanised 2025

The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

# Machine Learning in Vulkan with Cooperative Matrix 2

Jeff Bolz, NVIDIA

# Overview

- Background
- Cooperative Matrix 1 examples
- Limitations of Cooperative Matrix 1
- Motivating use cases from llama.cpp/ggml
- New features in Cooperative Matrix 2 and how they help
- Perf Results


stable-diffusion.cpp using Vulkan backend
https://github.com/leejet/stable-diffusion.cpp

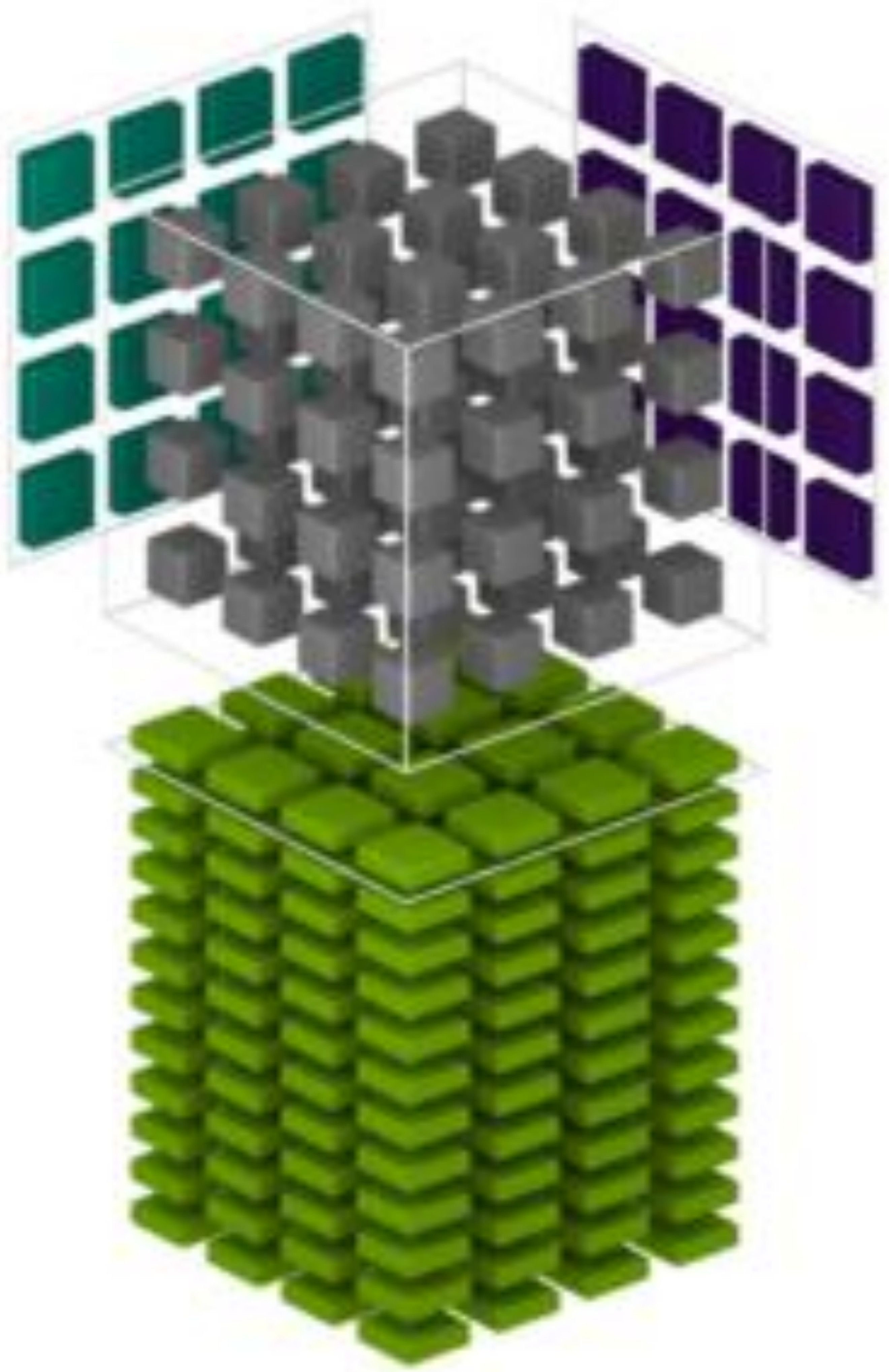
sd --diffusion-model flux1-dev-Q3_K.gguf --vae ae.safetensors --clip_l clip_l.safetensors --t5xxl t5xxl_fp16.safetensors --cfg-scale 1.0 --sampling-method euler -v --diffusion-fa -W 640 -H 640 -p "**an orange tabby cat typing on a laptop computer displaying the text 'Vulkan Machine Learning'**"

# Goal / Motivation

- Goal: Accelerate machine learning
  - Critical operation: accelerating large matrix multiplies

- Problem: SIMT was never the right programming model for large matrix multiplies
  - Shader author over-prescribes how to perform the multiply
  - Decomposed into tiny math ops, dominated by shepherding data between lanes or reading from shared memory
  - This decomposition is optimized for a particular HW platform

- New Functionality: *Group-wide matrix multiply*
  - **Uses the tensor cores**
  - Expose "medium size" matrix multiplies as a primitive that can be optimized
  - All invocations in a (complete) group *cooperate* to compute the result
  - Matrix is stored opaquely, spread across the group
  - Shaders can build larger GEMMs or other networks out of it

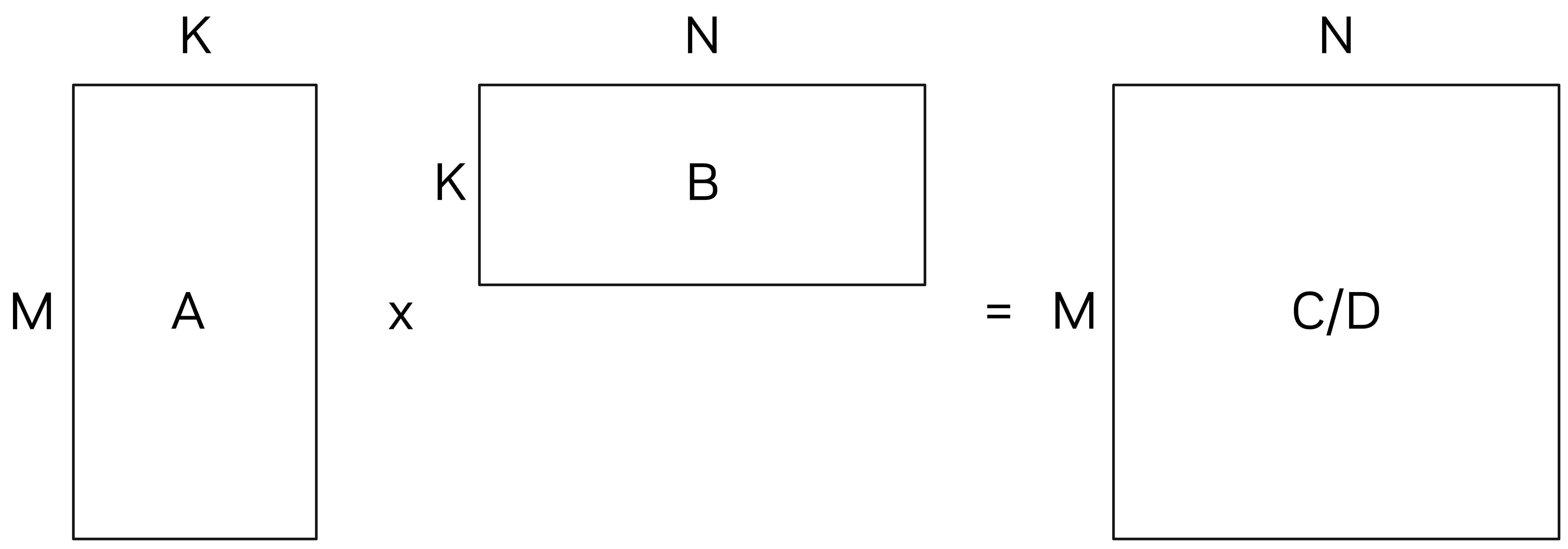


# Terminology

- “Cooperative Matrix” – a new matrix type where the storage for and computations performed on the matrix are spread across a set of invocations such as a subgroup (KHR_coopmat) or workgroup (NV_coopmat2)
  - KHR_cooperative_matrix (coopmat1): released summer 2023, widely supported today
  - NV_cooperative_matrix2: released October 2024, currently NVIDIA-only

- “D = A*B+C”
  - Matrix types are templated by component type, scope, dimensions, “Use” (A/B/Accumulator)
  - E.g. `coopmat<float16_t, gl_ScopeSubgroup, M, K, gl_MatrixUseA> matA;`

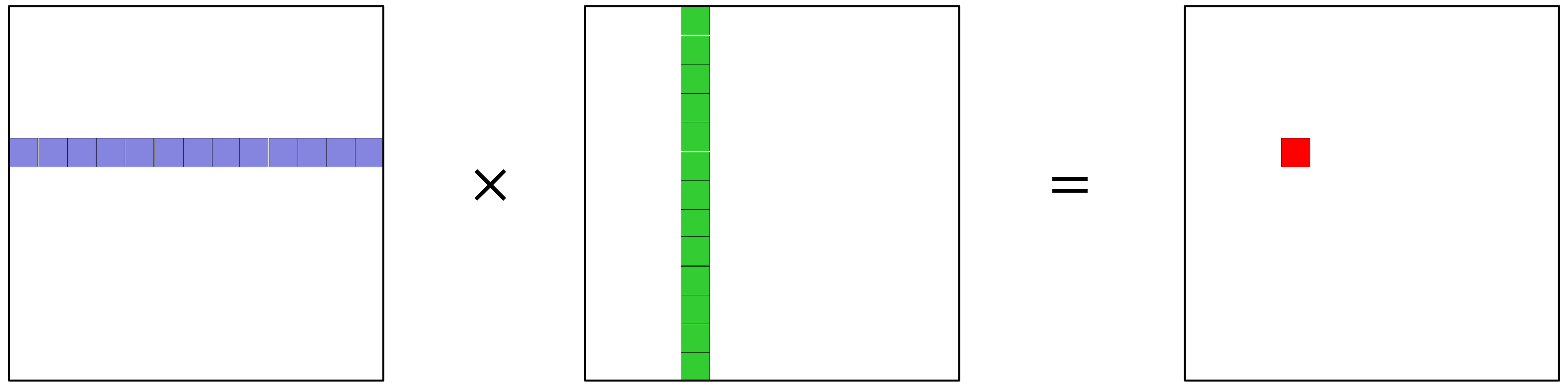
- “MxNxK” matrix multiply
  - A = MxK, B = KxN, C,D=MxN (rows x columns)
  - Supported sizes queried from Vulkan extension




- Precision
  - A=B=fp16, C=D={fp16 or fp32} (precision of C and D must match)
  - A=B=(s/u)int8, C=D=(s/u)int32
  - Hopefully more in the future

# Simple Cooperative Multiply

- Straightforward application of coopmat1 types and functions - sum($A_{ik}B_{kj}$) to accumulate one result tile
- Memory bandwidth-limited, not designed to get good reuse of memory
  - Tiles are too small!



```
lM = 16; lN = 8; lK = 16;
coopmat<float16_t, gl_ScopeSubgroup, lM, lK, UseA> matA;
coopmat<float16_t, gl_ScopeSubgroup, lK, lN, UseB> matB;
coopmat<float16_t, gl_ScopeSubgroup, lM, lN, UseAcc> matC;

uvec2 matrixID = uvec2(gl_WorkGroupID);
uint cRow = lM * matrixID.y;
uint cCol = lN * matrixID.x;

coopMatLoad(matC, inputC.x, sC * cRow + cCol, sC, RowMajor);

for (uint k = 0; k < K; k += lK) {
  uint aRow = lM * matrixID.y;
  uint aCol = k;
  coopMatLoad(matA, inputA.x, sA * aRow + aCol, sA, RowMajor);

  uint bRow = k;
  uint bCol = lN * matrixID.x;
  coopMatLoad(matB, inputB.x, sB * bRow + bCol, sB, RowMajor);

  matC = coopMatMulAdd(matA, matB, matC);
}

coopMatStore(matC, outputD.x, sD * cRow + cCol, sD, RowMajor);
```

~8 TFLOPS (RTX 4070) ☹

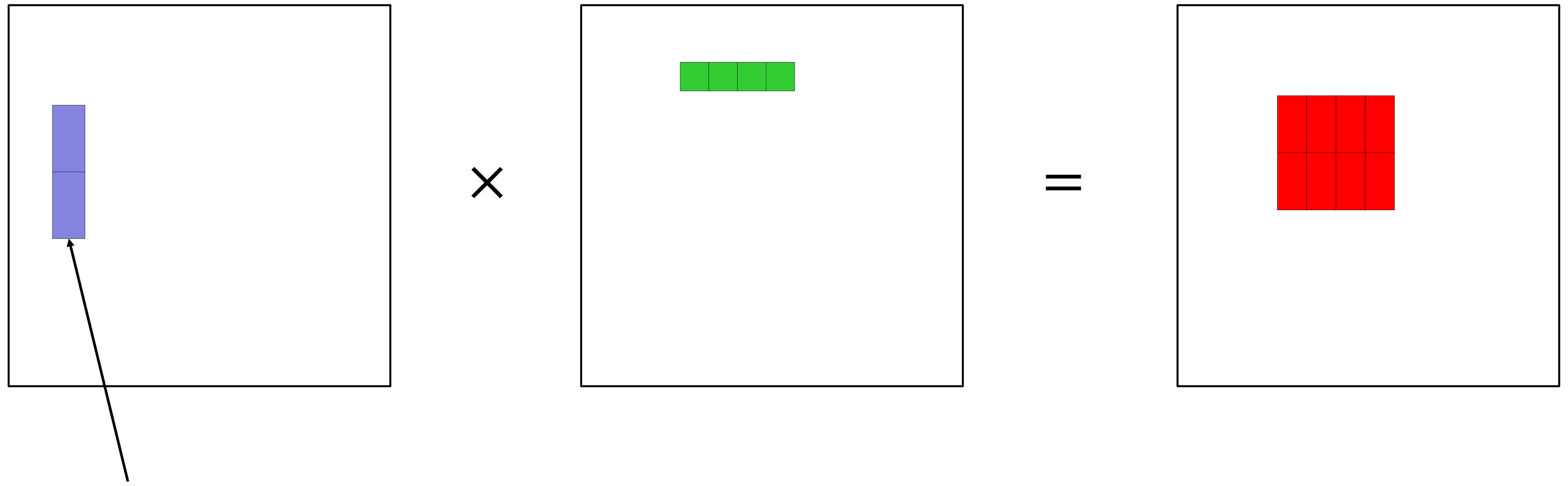
(RTX 4070 peak tensor core FP16 rate is around 116 TFLOPS)

# Optimized Coopmat1 Multiply

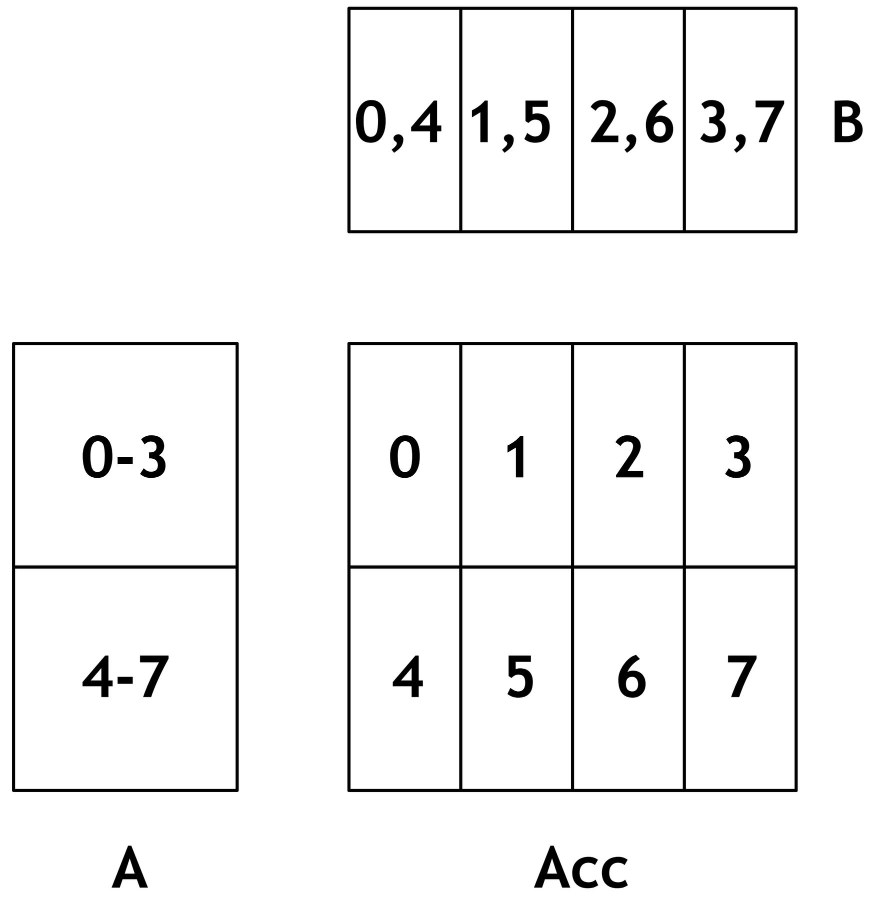
- Goal is to maximize the result tile size, which minimizes how many times A/B are loaded
- Subgroups cooperate to copy slivers of A/B from global to shared, then load out of shared
- Example (FP16): 8 subgroups split a 256x256 tile into 8 128x64 tiles (K=32)
  - Cooperate to copy A block (256x32) and B block (32x256) into shared memory
  - Then each subgroup loads the portions it needs from shared memory
- Optimal sizes depend on HW

```
fetch A,B for tile k=0 into register file

for (uint k = 0; k < K; k += TILE_K) {
  barrier() to wait for shmem loads in previous iteration

  copy tile k from register file to shared memory
  barrier() to wait for shmem stores to finish

  fetch A,B for tile k+1 into register file

  math loop {
    load from shared memory
    result[...] = coopMatMulAddNV(...);
  }
}
```

Full source code available at
https://github.com/jeffbolznv/vk_cooperative_matrix_perf




One iteration in 'k'




~98 TFLOPS (RTX 4070) ☺

(RTX 4070 peak tensor core FP16 rate is around 116 TFLOPS)

# Problems with Coopmat1

- GEMM wants large matrices split across a workgroup
  - Decomposing larger sizes into implementation-dependent tiles is something a compiler should do
- Manually staging through shared memory is clunky
  - Get bogged down in addressing math
  - Issues with type punning (want 16B loads, but elements are only 2B)
- Tensor isn't always a multiple of the HW matrix size
  - Need bounds-checking on loads/stores
  - Storing a matrix with bounds checking is surprisingly difficult
    - No knowledge of (row,col) information, so you have to go through shared memory
    - The whole matrix may not fit in shared memory! Go one warp at a time?
    - Workarounds on top of workarounds when you just want to say "store"
- Writing any non-trivial "fused" network needs matrix "Use" conversion (Acc->A/B), reductions, etc.

Manual pipelining,
shared memory staging,
tiling,
bounds checking

Compiler does it for you

# Triton Language

- A good level of abstraction
- Looks a lot like the “Simple Cooperative Multiply”
- https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html

```
# -----------------------------------------------------------
# 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, accumulator)
    # 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.float16)
```

| | CUDA | TRITON (and coopmat2!) |
|---|---|---|
| Memory Coalescing | Manual | Automatic |
| Shared Memory Management | Manual | Automatic |
| Scheduling (Within SMs) | Manual | Automatic |
| Scheduling (Across SMs) | Manual | Manual |

Compiler optimizations in CUDA vs Triton.

# Cooperative Matrix 2

## Seven New Features

- Flexible dimensions
- Workgroup scope matrices
- Tensor addressing
- Block loads
- Reductions
- Conversions
- Per-element operations

```
lM = 256; lN = 256; lK = 32;
coopmat<float16_t, gl_ScopeWorkgroup, lM, lK, UseA> matA;
coopmat<float16_t, gl_ScopeWorkgroup, lK, lN, UseB> matB;
coopmat<float16_t, gl_ScopeWorkgroup, lM, lN, UseAcc> matC;

uvec2 matrixID = uvec2(gl_WorkGroupID);
uint row = lM * matrixID.y;
uint col = lN * matrixID.x;

tensorLayoutNV<2> tensorA = createTensorLayoutNV(2);
tensorLayoutNV<2> tensorB = createTensorLayoutNV(2);
tensorLayoutNV<2> tensorC = createTensorLayoutNV(2);

tensorA = setTensorLayoutDimensionNV(tensorA, M, K);
tensorB = setTensorLayoutDimensionNV(tensorB, K, N);
tensorC = setTensorLayoutDimensionNV(tensorC, M, N);

coopMatLoadTensor(matC, inputC.x, slice(tensorC, row, lM, col, lN));

for (uint k = 0; k < K; k += lK) {
  coopMatLoadTensor(matA, inputA.x, slice(tensorA, row, lM, k, lK));
  coopMatLoadTensor(matB, inputB.x, slice(tensorB, k, lK, col, lN));

  matC = coopMatMulAdd(matA, matB, matC);
}

coopMatStoreTensor(matC, outputD.x, slice(tensorC, row, lM, col, lN));
```

~97 TFLOPS (RTX 4070) ☺

(RTX 4070 peak tensor core FP16 rate is around 116 TFLOPS)

# Tensor Layout

## What?

- Tensor layout is a *software structure* constructed in the shader
  - A convenient, *extensible* way to specify addressing calculations
  - `coopMatLoadTensorNV(coopmat, T[] buf, uint element, tensorLayout[, tensorView][, decodeFunc]);`
- Tensor layout describes the shape of the *tensor in memory*
- Tensor layouts can be up to 5D. Each dimension has size, stride, offset, span, that define the shape of the region and layout in memory
  - Defines the tensor size for bounds-checking
- Tensor layout logically maps:
  - matrix (row,col) -> 1D index -> ND coordinate -> address

# Tensor Layout

## Why?

- Accesses all come from the slice, even if addressing is complex
  - Loading through shared memory only needs to load the slice
- These layouts are amenable to compiler optimization and HW acceleration
  - Often, parts of the calculations can be easily optimized away
    - E.g. for 2D rowmajor, ND coordinate == (row,col) because matrix dim == span dim
    - “Pay for what you use”
- Considered an alternative with “address calculation callback”
  - Not practical to optimize
  - Tensor layout/view are **expressive** and **optimizable**

# Tensor Layout

## Logical Definition in the Specification

```
struct tensorLayout<uint32_t Dim,
   TensorClampMode Mode = TensorClampModeUndefined>
{
  static constexpr uint32_t LDim = Dim;
  static constexpr TensorClampMode clampMode = Mode;
  uint32_t blockSize[LDim];
  uint32_t layoutDimension[LDim];
  uint32_t stride[LDim];
  int32_t offset[LDim];
  uint32_t span[LDim];
  uint32_t clampValue;
};
```

```
uint32_t computeIndex(tensorLayout t, uint32_t row,
                      uint32_t col, uint32_t N)
{
    uint32_t index = row * N + col;
    return computeIndex(t, index);
}

uint32_t computeIndex(tensorLayout t, uint32_t index)
{
    uint32_t coord[t.LDim];
    for (int32_t dim = t.LDim-1; dim >= 0; --dim) {
        coord[dim] = index % t.span[dim];
        index /= t.span[dim];
    }

    index = 0;
    uint32_t coordInBlock[t.LDim];
    uint32_t blockCoord[t.LDim];

    for (uint32_t dim = 0; dim <= t.LDim-1; ++dim) {
        int32_t c = coord[dim] + t.offset[dim];

        if (c < 0 || c >= t.layoutDimension[dim]) {
            // handle OOB
        }

        coordInBlock[dim] = c % t.blockSize[dim];
        blockCoord[dim] = c / t.blockSize[dim];

        index += blockCoord[dim] * t.stride[dim];
    }
    return index;
}
```

# Tensor View

Why?

- Tensor view is optionally applied in addition to a layout. View can reinterpret layout and dimensionality, and permute coordinates
  - Similar to torch.view/permute/reshape/transpose/etc

- Tensor view logically maps:
  - (row, col) -> 1D index -> ND coord -> permute -> 1D index
  - Then continue on with the tensor layout calculation

- Tensor view is needed when you want to permute coordinates or change the number of dimensions
  - transpose
  - space_to_depth/depth_to_space

# Tensor View

## Logical Definition in the Specification

```
struct tensorView<uint Dim, bool hasDimensions,
    uint32_t p0, ..., uint32_t p<Dim-1>>
{
  static constexpr uint32_t VDim = Dim;
  static constexpr bool hasDim = hasDimensions;
  static constexpr uint32_t permutation[VDim] =
       {p0, ..., p<Dim-1>};

  uint32_t viewDimension[VDim];
  uint32_t viewStride[VDim];
  uint32_t clipRowOffset, clipRowSpan,
           clipColOffset, clipColSpan;
};
```

```
uint32_t computeIndex(tensorLayout t, tensorView v, uint32_t index)
{
    auto &dimensions = v.hasDimensions ? v.viewDimension : t.span;

    uint32_t stride[v.VDim];
    if (v.hasDimensions) {
        stride = v.viewStride;
    } else {
        // set stride to match t.span
        stride[v.VDim-1] = 1;
        for (int32_t dim = v.VDim-2; dim >= 0; --dim) {
            stride[dim] = stride[dim+1] * t.span[dim+1];
        }
    }

    uint32_t result = 0;
    for (int32_t dim = v.VDim-1; dim >= 0; --dim) {
        uint32_t i = v.permutation[dim];

        uint32_t coord = index % dimensions[i];
        index /= dimensions[i];

        result += coord * stride[i];
    }

    return computeIndex(t, result);
}
```

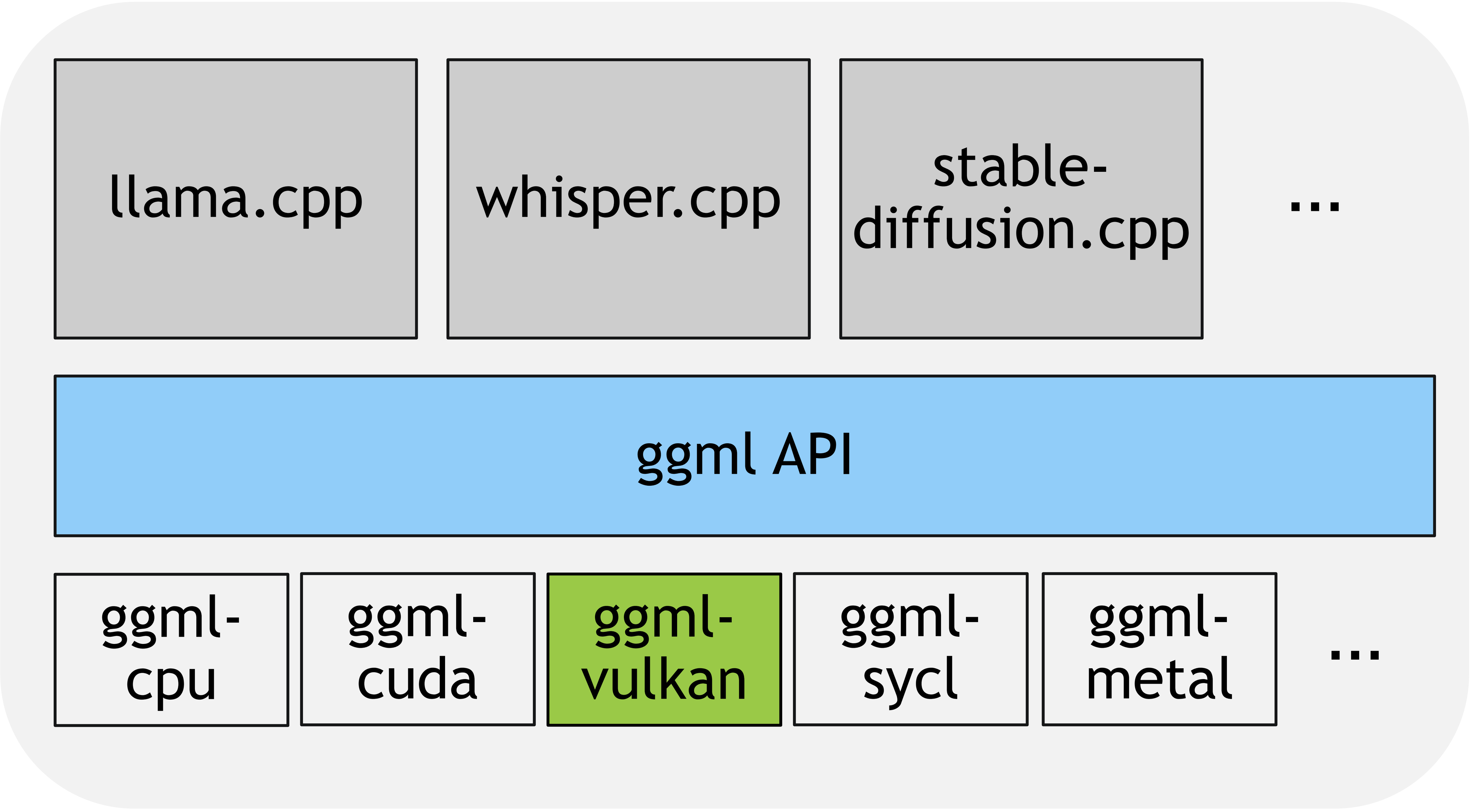
# llama.cpp/ggml

https://github.com/ggerganov/llama.cpp https://github.com/ggerganov/ggml

## Description

The main goal of `llama.cpp` is to enable LLM inference with minimal setup and state-of-the-art performance on a wide variety of hardware - locally and in the cloud.

- Plain C/C++ implementation without any dependencies
- Apple silicon is a first-class citizen - optimized via ARM NEON, Accelerate and Metal frameworks
- AVX, AVX2 and AVX512 support for x86 architectures
- 1.5-bit, 2-bit, 3-bit, 4-bit, 5-bit, 6-bit, and 8-bit integer quantization for faster inference and reduced memory use
- Custom CUDA kernels for running LLMs on NVIDIA GPUs (support for AMD GPUs via HIP)
- Vulkan and SYCL backend support
- CPU+GPU hybrid inference to partially accelerate models larger than the total VRAM capacity




- Key use cases to accelerate:
  - Matrix-matrix multiply
  - Mixture-of-Experts mat-mat mul
  - Flash Attention
  - Quantization Formats

- Where we are today:
  - Vulkan backend works "everywhere" (needs basic vk1.2 support)
  - Mat-mat mul has scalar, KHR_coopmat, NV_coopmat2 paths
  - Flash Attention has an NV_coopmat2 path
  - Quantization Formats supported in all paths

# Quantization Formats

## Dequantization Example

- Job #1 is fitting the model in vidmem
- Mat-vec mul is bandwidth-limited, so this is good for perf, too

```
#if defined(DATA_A_IQ4_NL)
#extension GL_EXT_shader_16bit_storage : require
#define QUANT_K 32
#define QUANT_R 2

struct block_iq4_nl
{
    float16_t d;
    uint8_t qs[QUANT_K/2];
};

#define A_TYPE block_iq4_nl

const int8_t kvalues_iq4nl[16] = {
    int8_t(-127), int8_t(-104), int8_t(-83), int8_t(-65), int8_t(-49), int8_t(-35), int8_t(-22), int8_t(-10),
    int8_t(1), int8_t(13), int8_t(25), int8_t(38), int8_t(53), int8_t(69), int8_t(89), int8_t(113)
};
#endif
```

```
#elif defined(DATA_A_IQ4_NL)
            const uint idx = pos_a + (loadc_a + 1) * p.stride_a / LOAD_VEC_A + loadr_a;
            const uint buf_idx = (loadc_a + 1) * (BK+1) + loadr_a;

            const uint ib = idx / 16;
            const uint iqs = idx & 0xF;

            const float d = float(data_a[ib].d);
            const uint vui = uint(data_a[ib].qs[iqs]);
            const vec2 v = vec2(kvalues_iq4nl[vui & 0xF], kvalues_iq4nl[vui >> 4]) * d;

            buf_a[buf_idx     ] = FLOAT_TYPE(v.x);
            buf_a[buf_idx + 16] = FLOAT_TYPE(v.y);
#endif
```

- IQ4_NL: ~4-bit/elem -> LUT -> scale
  - (not all formats use a LUT)
- Usually 32 or 256 elements per block
- Usually ~2 to ~8 bits/element
- Stored in memory as tensor-of-blocks

https://github.com/ggerganov/llama.cpp/blob/master/ggml/src/ggml-vulkan/vulkan-shaders/types.comp#L301
https://github.com/ggerganov/llama.cpp/blob/master/ggml/src/ggml-vulkan/vulkan-shaders/mul_mm.comp#L443

# Quantization Formats

```
#if defined(DATA_A_IQ4_NL)
#extension GL_EXT_shader_16bit_storage : require
#define QUANT_K 32
#define QUANT_R 2

struct block_iq4_nl
{
    float16_t d;
    uint8_t qs[QUANT_K/2];
};

#define A_TYPE block_iq4_nl

const int8_t kvalues_iq4nl[16] = {
    int8_t(-127), int8_t(-104), int8_t(-83), int8_t(-65), int8_t(-49), int8_t(-35), int8_t(-22), int8_t(-10),
    int8_t(1), int8_t(13), int8_t(25), int8_t(38), int8_t(53), int8_t(69), int8_t(89), int8_t(113)
};
#endif
```

Block Address

Coord in block

Dequantize element

```
#elif defined(DATA_A_IQ4_NL)
    const uint idx = pos_a + (loadc_a + 1) * p.stride_a / LOAD_VEC_A + loadr_a;
    const uint buf_idx = (loadc_a + 1) * (BK+1) + loadr_a;

    const uint ib = idx / 16;
    const uint iqs = idx & 0xF;

    const float d = float(data_a[ib].d);
    const uint vui = uint(data_a[ib].qs[iqs]);
    const vec2 v = vec2(kvalues_iq4nl[vui & 0xF], kvalues_iq4nl[vui >> 4]) * d;

    buf_a[buf_idx      ] = FLOAT_TYPE(v.x);
    buf_a[buf_idx + 16] = FLOAT_TYPE(v.y);
#endif
```

# Dequantization Callback

- Implementation computes block coordinate, coordinate in block, and pointer to start of the block
  - Passes these to shader-supplied callback function -> return value populates the matrix
- This works well with compiler-implemented staging through shared memory
- Generic, composable way to support dequantization
  - Mat-mul and FlashAttention kernels just plug in a different callback function for each format
- Consider decoding two or in some cases four elements at a time, for performance

```
#elif defined(DATA_A_IQ4_NL)

float16_t decodeFunc(const in decodeBuf bl, const in uint blockCoords[2], const in uint coordInBlock[2])
{
    const float16_t d = bl.block.d;
    const uint idx = coordInBlock[1];
    const uint iqs = idx & 0xF;
    const uint shift = (idx & 0x10) >> 2;
    uint32_t qs = bl.block.qs[iqs];
    qs >>= shift;
    qs &= 0xF;
    float16_t ret = float16_t(kvalues_iq4nl[qs]) * d;
    return ret;
}

#endif
```

```
coopMatLoadTensorNV(mat_a, data_a, pos_a, slice(tensorLayoutA, ir * BM, BM, block_k, BK), decodeFunc);
```

# Mixture of Experts

- Matrix B loads are indirected through a table to select the row

```
const uint row_i = ic * BN + loadc_b + l;
if (row_i < _ne1) {
    const u16vec2 row_idx = row_ids[row_i];
    buf_b[(loadc_b + l) * (BK+1) + loadr_b] = FLOAT_TYPE(data_b[pos_b + row_idx.y * p.batch_stride_b + (row_idx.x % p.ne11) * p.stride_b + loadr_b]);
} else {
    buf_b[(loadc_b + l) * (BK+1) + loadr_b] = FLOAT_TYPE(0.0f);
}
```

```
B_TYPE decodeFuncB(const in decodeBufB bl, const in uint blockCoords[2], const in uint coordInBlock[2])
{
    const uint row_i = blockCoords[0];

    if (row_i >= _ne1) {
        return B_TYPE(0.0);
    }

    const u16vec4 row_idx = row_ids[row_i];
    B_TYPE ret = data_b[row_idx.y * p.batch_stride_b + (row_idx.x % p.ne11) * p.stride_b + blockCoords[1]];

    return ret;
}
```

- Storing D matrix also requires remapping, use per-element operation

```
D_TYPE perElemOpD(const in uint32_t r, const in uint32_t c, const in D_TYPE elem, const in uint32_t ir, const in uint32_t ic)
{
    uint dr = ir * BM + r;
    uint dc = ic * BN + c;

    if (dr < p.M && dc < _ne1) {
        uint row_i = dc;
        const u16vec4 row_idx = row_ids[row_i];
        data_d[row_idx.y * p.batch_stride_d + row_idx.z * p.stride_d + dr] = elem;
    }
    return elem;
}

coopMatPerElementNV(mat_d, mat_d, perElemOpD, ir, ic);
```

# Flash Attention 2

## Overview of the Algorithm

- Attention formula (from Wikipedia):

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \in \mathbb{R}^{m \times d_v}$$

Softmax formula:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Softmax is “just” a component-wise exp() and per-row scale (normalization)
  - FlashAttention trick is to pretend you can compute the denominator during each K step, and then readjust it on the next step when you have more information
  - OK, there’s a bit more to it than that, but that’s the key insight

- Needs the following coopmat2 features:
  - Row reductions (for softmax)
  - Matrix Use conversions (convert QK^t from Accumulator to A matrix)
  - Per-element operations (to clear padding elements)

# Flash Attention 2

- coopMatReduceNV supports a callback function to combine a pair of values:

```
ACC_TYPE maxReduce(const in ACC_TYPE x, const in ACC_TYPE y) {
    return max(x, y);
}

coopMatReduceNV(rowmax, S, gl_CooperativeMatrixReduceRowNV, maxReduce);
```

- Need to fill padding elements with –inf, before doing max-reduce:

```
// Replace matrix elements >= numRows or numCols with 'replace'
ACC_TYPE replacePadding(const in uint32_t row, const in uint32_t col, const in ACC_TYPE elem, const in ACC_TYPE replace,
                        const in uint32_t numRows, const in uint32_t numCols) {
    if (row >= numRows || col >= numCols) {
        return replace;
    }
    return elem;
}

coopMatPerElementNV(S, S, replacePadding, ACC_TYPE(-1.0/0.0), R, C);
```

# Cooperative Matrix 2

## Seven New Features

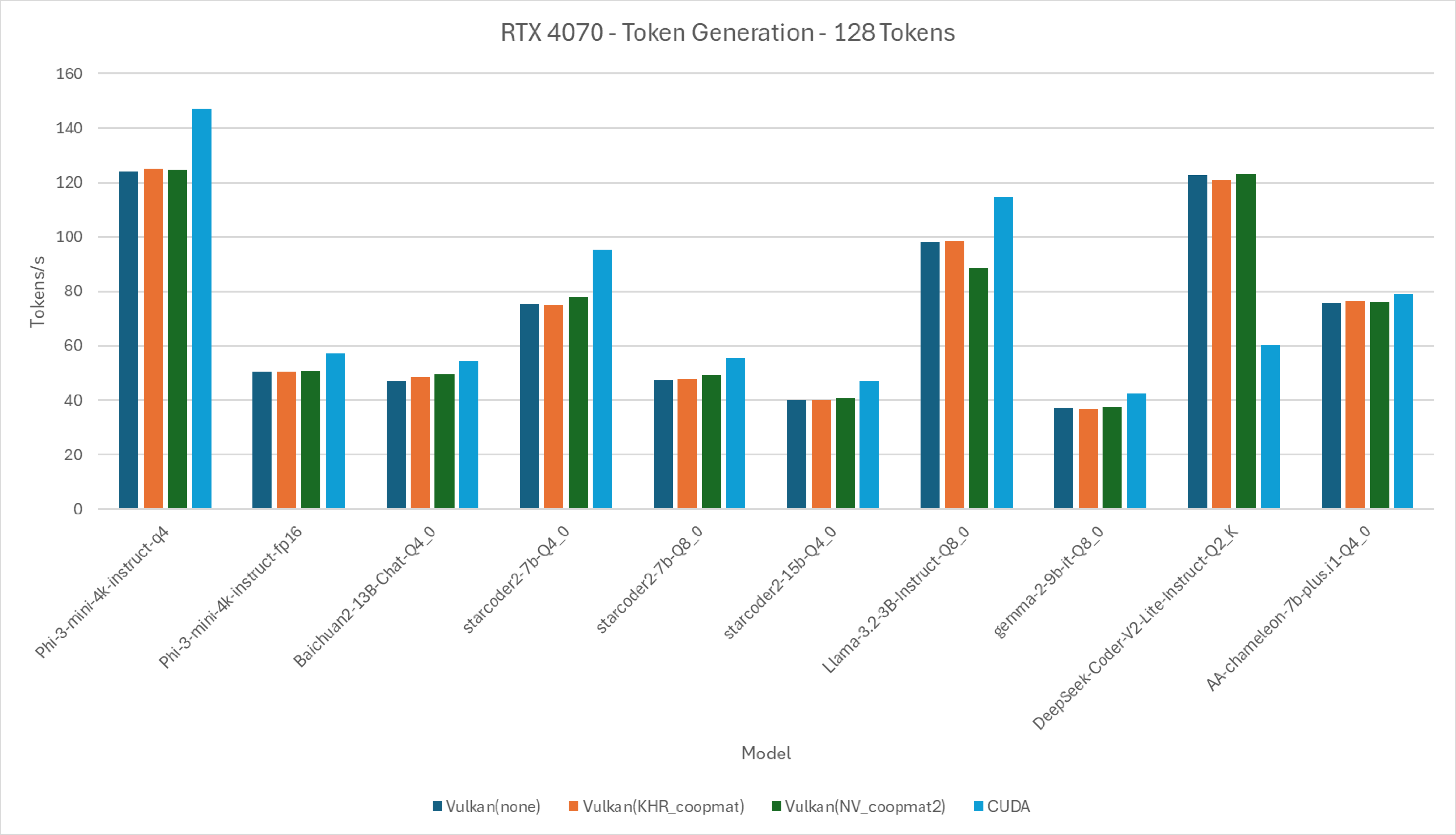
- Workgroup scope matrices
- Flexible dimensions
- Reductions
- Conversions
- Per-element operations
- TensorAddressing
- Block Loads

| | WG scope | Flex dim | Reduction | Conversion | Per-elem op | Tensor Addressing | Block Loads |
|---|---|---|---|---|---|---|---|
| MatMul | ✓ | ✓ | | | | ✓ | ✓ |
| MoE | ✓ | ✓ | | | ✓ | ✓ | ✓ |
| FA2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

- (And performance heavily relies on workgroup scope, flexible dimensions, and tensor addressing)

# Performance

# Performance



# Performance

# Performance

# Conclusion

- Machine learning acceleration is possible in Vulkan today
  - llama.cpp -> ggml -> Vulkan w/Cooperative Matrix(2)
- The Vulkan/SPIR-V/GLSL NV_coopmat2 extensions are available now
  - In the Vulkan 1.4.304 SDK
- Supported in developer drivers at https://developer.nvidia.com/vulkan-driver
  - Will be in next major release (R575)
  - Supported on all NVIDIA RTX GPUs
- Try it yourself in https://github.com/ggerganov/llama.cpp!

NVIDIA®