Cooperative Matrix Multiply

Pierre Boudier
May 2022

Goal / Motivation

- **Goal: Accelerate machine learning**
  - Critical operation: accelerating large, low-precision matrix multiplies
  - NVIDIA currently supports FP16 and INT8

- **Problem: SIMT was never the right 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: *Subgroup-wide matrix multiply***
  - Expose “medium size” matrix multiplies as a primitive that can be optimized
  - All invocations in a (complete) subgroup *cooperate* to compute the result
  - Matrix is stored opaquely, spread across the subgroup
  - Shaders can build larger GEMMs 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

- “D = A*B+C”

- “MxNxK” matrix multiply
  - A = MxK, B = KxN, C,D=MxN (rows x columns)
  - Supported sizes queried from VK extension
  - NVIDIA supports 16x8x8, 16x8x16, 16x16x16, a few others

- Precision
  - NVIDIA supports A=B=fp16, C=D={fp16 or fp32} (precision of C and D must match)
  - NVIDIA supports A=B=INT8, C=D=INT32

Types

- **GLSL:**
  - u/i/fcoopmatNV<bits, scope, rows, cols>
  - Adds limited “parameterized type” support to GLSL (yay!)

- **SPIR-V:**
  - **OpTypeCooperativeMatrixNV** %componenttype %scope %rows %cols

- **Scope, rows, and cols can all be specialization constants**
  - Goal is to be able to write a single shader for lots of hardware
  - GLSL and SPIR-V have very few constraints on valid combinations, mostly leaving it to the SPIR-V environment spec

Types

- Example (GLSL):

```
layout(constant_id = 0) const int rows = 16;
layout(constant_id = 1) const int cols = 8;
fcoopmatNV<16, gl_ScopeSubgroup, rows, cols> m;
```

- Example (SPIR-V):

```
%half = OpTypeFloat 16
%scope = OpConstant %i32 3
%rows = OpSpecConstant %i32 16
%cols = OpSpecConstant %i32 8
%mtype = OpTypeCooperativeMatrixNV %half %scope %rows %cols
```

Load/Store

- New load/store built-ins (GLSL):

```
void coopMatLoadNV(out fcoopmatNV m, float[] buf, uint element, uint stride, bool colMajor);
void coopMatLoadNV(out fcoopmatNV m, float16_t[] buf, uint element, uint stride, bool colMajor);
void coopMatStoreNV(fcoopmatNV m, out float[] buf, uint element, uint stride, bool colMajor);
void coopMatStoreNV(fcoopmatNV m, out float16_t[] buf, uint element, uint stride, bool colMajor);
```

  - `buf` must be in buffer or shared storage, `element` is array index of the start of the matrix

- New load/store ops (SPIR-V):

```
%result = OpCooperativeMatrixLoadNV %resultType %pointer %stride %colmajor
OpCooperativeMatrixStoreNV %pointer %object %stride %colmajor
```

- Frontend compiler computes %pointer = OpAccessChain(buf, element)
- colMajor must be constant boolean expression
- All parameters must be equal across the whole scope
- All invocations in the scope must be active

Matrix Multiply and Add

- New built-in (GLSL):

```
fcoopmatNV coopMatMulAddNV(fcoopmatNV A, fcoopmatNV B, fcoopmatNV C);
```

- New SPIR-V OP:

```
%result = OpCooperativeMatrixMulAddNV %resultType %A %B %C
```

- Dimensions/types must form an MxNxK multiply that is supported by the implementation
- Precision and order of operations is implementation-dependent
- GLSL return type is derived from type of “C”

Other Operations

- **Component-wise arithmetic:**
  - GLSL: `+, -, *, /`
  - SPIR-V: `OpFAdd, OpFNegate/OpFSub, OpMatrixTimesScalar, OpFDiv`

- **Construct matrix with different component type but same size:**
  - GLSL: Constructor
  - SPIR-V: `OpFConvert`

Cooperative Matrices as Composite Types

- **Cooperative matrices act as *Composite Types***
  - As if they were vectors with an implementation-dependent component count
  - Mapping of (InvocationID, index) -> (row, column) is implementation-dependent

- **“Opaque indexing” within an invocation**
  - Query number of components per invocation
    - GLSL: `m.length()`
    - SPIR-V: `OpCooperativeMatrixLengthNV`
  - Indexing within an invocation
    - GLSL: `m[i]` (including as lvalue)
    - SPIR-V: `OpCompositeExtract, OpCompositeInsert, OpAccessChain`
  - Can be used to do element-wise tensor ops

- **Construct from scalar type:**
  - GLSL: `fcoopmat<…>(float)`
  - SPIR-V: `OpCompositeConstruct/OpConstantComposite` with one scalar operand

Vulkan Extension

- Just advertises capabilities

```
VKAPI_ATTR VkResult VKAPI_CALL vkGetPhysicalDeviceCooperativeMatrixPropertiesNV(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pPropertyCount,
    VkCooperativeMatrixPropertiesNV*            pProperties);
```

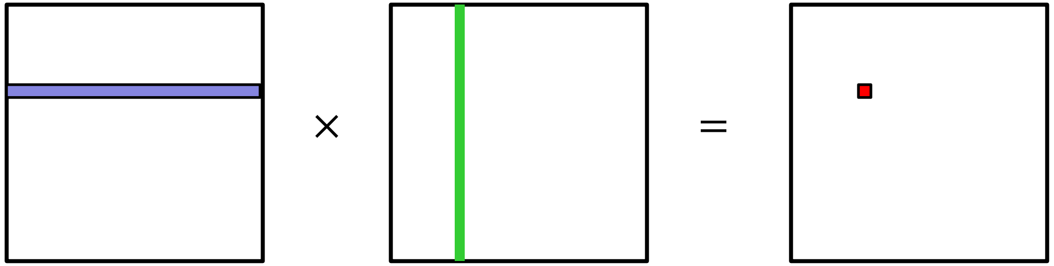
- In NVIDIA's implementation, we support:
  - `AType = BType = fp16  CType = DType = fp16  MxNxK = 16x16x16 scope = Subgroup`
  - `AType = BType = fp16  CType = DType = fp16  MxNxK = 16x8x16  scope = Subgroup`
  - `AType = BType = fp16  CType = DType = fp16  MxNxK = 16x8x8   scope = Subgroup`
  - `(same for C=D=fp32)`
  - `AType = BType = u8    CType = DType = u32   MxNxK = 16x16x32 scope = Subgroup`
  - `AType = BType = u8    CType = DType = u32   MxNxK = 16x8x32  scope = Subgroup`
  - `AType = BType = u8    CType = DType = u32   MxNxK = 8x8x32   scope = Subgroup`
  - `(same for signed integer)`

Performance

Scalar Loop

- Each invocation computes one element of the result matrix
- Lots of redundant loads
- Example of what NOT to do

```
uint i = gl_GlobalInvocationID.y;
uint j = gl_GlobalInvocationID.x;
float16_t C = inputC.x[sC * i + j];

for (uint k = 0; k < K; ++k) {
  float16_t A = inputA.x[sA * i + k];
  float16_t B = inputB.x[sB * k + j];
  C += A*B;
}

outputD.x[sD * i + j] = C;
```

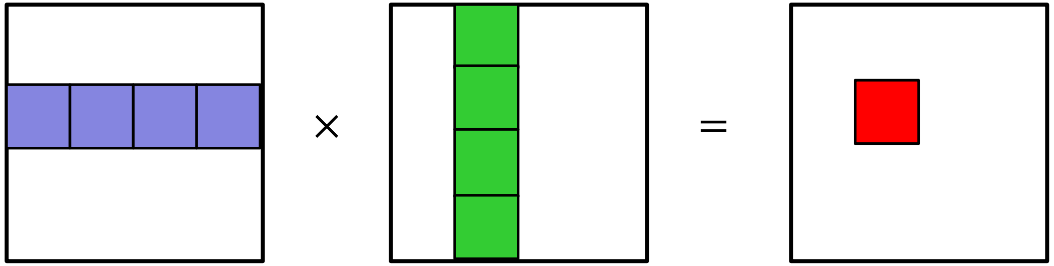


GPU	TFLOPS
RTX 2070	0.232
RTX TITAN	0.528

Simple Cooperative Multiply

- A bit more coordinate calculation, but still a simple sum($A_{ik}B_{kj}$)
- Still very memory-limited, but improved

```
lM = 16; lN = 8; lK = 8;
fcoopmatNV<16, gl_ScopeSubgroup, lM, lK> matA;
fcoopmatNV<16, gl_ScopeSubgroup, lK, lN> matB;
fcoopmatNV<16, gl_ScopeSubgroup, lM, lN> matC;

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

coopMatLoadNV(matC, inputC.x, sC * cRow + cCol, sC, false);

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

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

  matC = coopMatMulAddNV(matA, matB, matO);
}

coopMatStoreNV(matC, outputD.x, sD * cRow + cCol, sD, false);
```

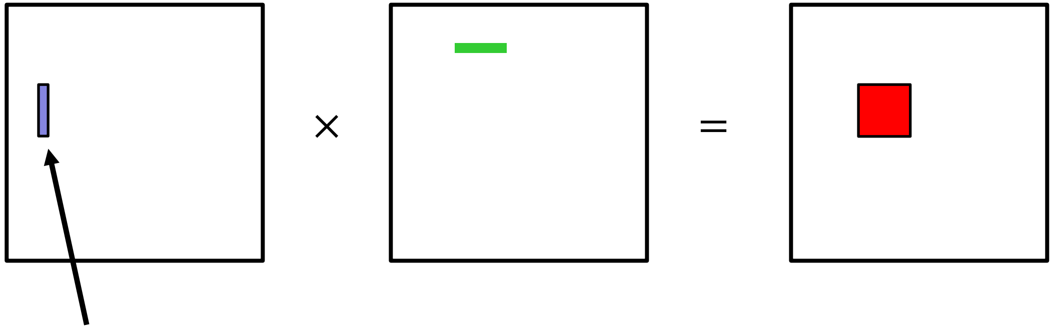


GPU	TFLOPS
RTX 2070	2.00
RTX TITAN	3.70

Tiled Scalar Multiply

- **Illustrates a way to tile the multiply**
  - Load row, load column, outer product
  - Name of the game is to load once, then perform as many multiplies as possible
  - Limited by register file size
  - Maybe 8x8 tile size per invocation

```
float16_t C[C_ROWS][C_COLS];
uvec2 tileID = uvec2(gl_WorkGroupID.xy);
uvec2 inv = uvec2(gl_LocalInvocationID.xy);
// load C
...
// iterate through K dimension and accumulate
for (uint k = 0; k < K; ++k) {
  float16_t A[C_ROWS];
  for (uint i = 0; i < C_ROWS; ++i) {
    uint gi = TILE_M * tileID.y + (C_ROWS * inv.y + i);
    uint gk = k;
    A[i] = inputA.x[sA * gi + gk];
  }
  float16_t B;
  for (uint j = 0; j < C_COLS; ++j) {
    uint gk = k;
    uint gj = TILE_N * tileID.x + (C_COLS * inv.x + j);
    B = inputB.x[sB * gk + gj];
    for (uint i = 0; i < C_ROWS; ++i) {
      C[i][j] = A[i] * B + C[i][j];
    }
  }
}
// store C
...
```



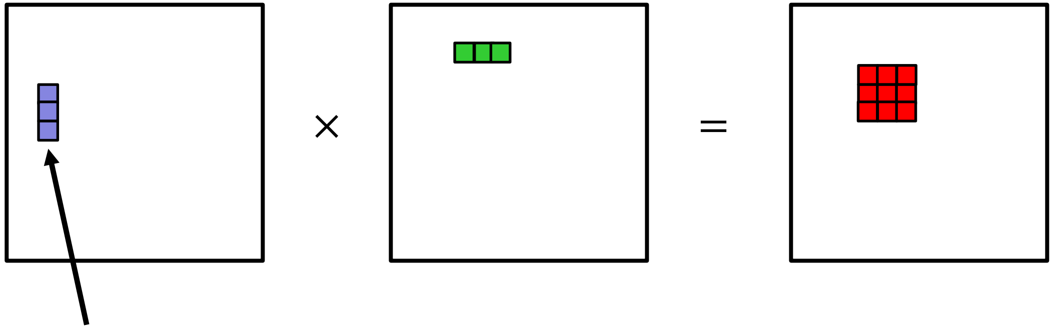
One iteration in ‘k’

GPU	TFLOPS
RTX 2070	1.45
RTX TITAN	2.79

Tiled Cooperative Matrix Multiply

- Apply tiling approach to coop matrices
- Effectively 32x the register file, since matrices are spread over a subgroup
  - 112x112x16 multiply per outer loop iter
  - Takes advantage of cheap communication within subgroups to maximize reuse
- Shader code is no more complex than what you’d write for scalar

```
fcoopmatNV<16, gl_ScopeSubgroup, lM, lN> matC[C_ROWS][C_COLS];
uvec2 tileID = uvec2(gl_WorkGroupID.xy);
// load matC
...
for (uint k = 0; k < K; k += TILE_K) {
  fcoopmatNV<16, gl_ScopeSubgroup, lM, lK> matA[C_ROWS];
  for (uint i = 0; i < C_ROWS; ++i) {
    uint gi = TILE_M * tileID.y + lM * i;
    uint gk = k;
    coopMatLoadNV(matA[i], inputA.x, sA * gi + gk, sA, false);
  }
  fcoopmatNV<16, gl_ScopeSubgroup, lK, lN> matB;
  for (uint j = 0; j < C_COLS; ++j) {
    uint gk = k;
    uint gj = TILE_N * tileID.x + lN * j;
    coopMatLoadNV(matB, inputB.x, sB * gk + gj, sB, false);
    for (uint i = 0; i < C_ROWS; ++i) {
      matC[i][j] = coopMatMulAddNV(matA[i], matB, matC[i][j]);
    }
  }
}
// store matC
...
```

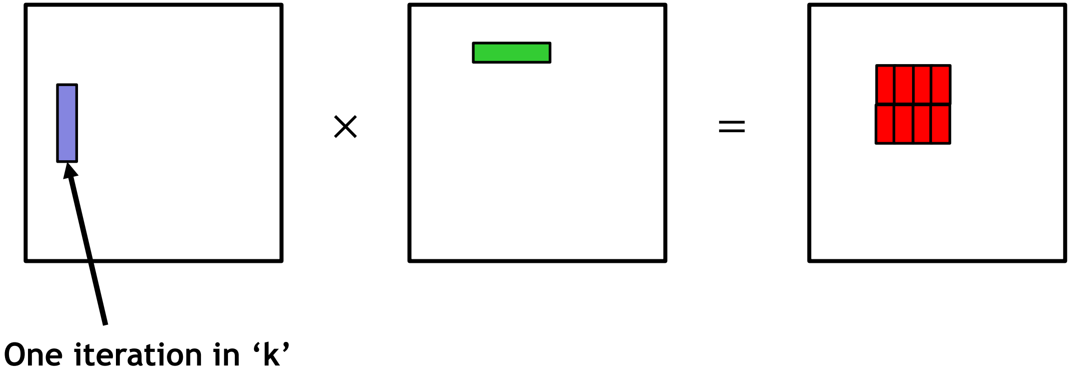


One iteration in ‘k’

GPU	TFLOPS
RTX 2070	21.8
RTX TITAN	42.1

Staging Through Shared Memory

- **Subgroups cooperate to copy data from buffer to shared, then load out of shared**
  - Overlap buffer loads with matrix math (i.e. pipeline load for next iteration)
- **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
  - Only 128B of buffer loads per thread per K iteration overlapping with 8K FMADs
- **Example (INT8): 8 subgroups split a 128x256 tile into 8 64x64 tiles (K=64)**
  - Accumulator is INT32, requires 2x register file, hurts latency hiding
  - Double math rate, half tile size = 4x harder to hide latency, much farther from SOL



GPU	INT8 TOPS
RTX 2070	66.0
RTX TITAN	~130

GPU	FP16 TFLOPS
RTX 2070	49.0
RTX TITAN	~100

Staging Through Shared Memory

- Pseudocode (this one is too large to fit in the margin):

```
fetch A,B for tile k=0 into register file
for (uint k = 0; k < K; k += TILE_K) {
  barrier() to wait for shared memory loads in previous iteration to finish

  copy tile k from register file to shared memory
  barrier() to wait for shared memory 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
  - Tiled Cooperative Matrix Multiply in tiled.comp
  - Staging Through Shared Memory in shmem.comp