Instrumenting MLIR Based ML Compilers for GPU Performance Analysis and Optimization

Corbin Robeck, AMD
ML Compilers/AMD Research

LLVM Developers' Meeting – October 2024

AMD
together we advance_

Outline

Background and Motivation

- Modern machine learning compiler frameworks make heavy use of MLIR and JIT style GPU kernels
  - This poses a challenge from a compiler-based performance analysis and optimization perspective
    - Dynamic binary instrumentation tools can often be too coarse grained and do not have access to the full set of higher-level source level information generated in the various lowering layers (multiple MLIR IRs → LLVM IR → ISA)
    - Without instrumentation that is integrated into the compiler, tracing a specific kernel defect leading to bottlenecks through the various lowering passes can be difficult and tedious
    - It is often useful to correlate bottleneck analysis to not just program source but also higher-level operations and data structures (e.g., middle IR layer dot operation on a specific tensor object).
- The overarching project goal is to develop a set of light-weight, customizable, open-source MLIR/LLVM based compiler passes to perform performance analysis and optimization instrumentation for a set of popular MLIR+GPU based ML compilers
- Instrumentation in this case: injecting bottleneck and optimization analysis code into performance critical sections of the GPU code through compiler passes at various places along the compiler pipeline

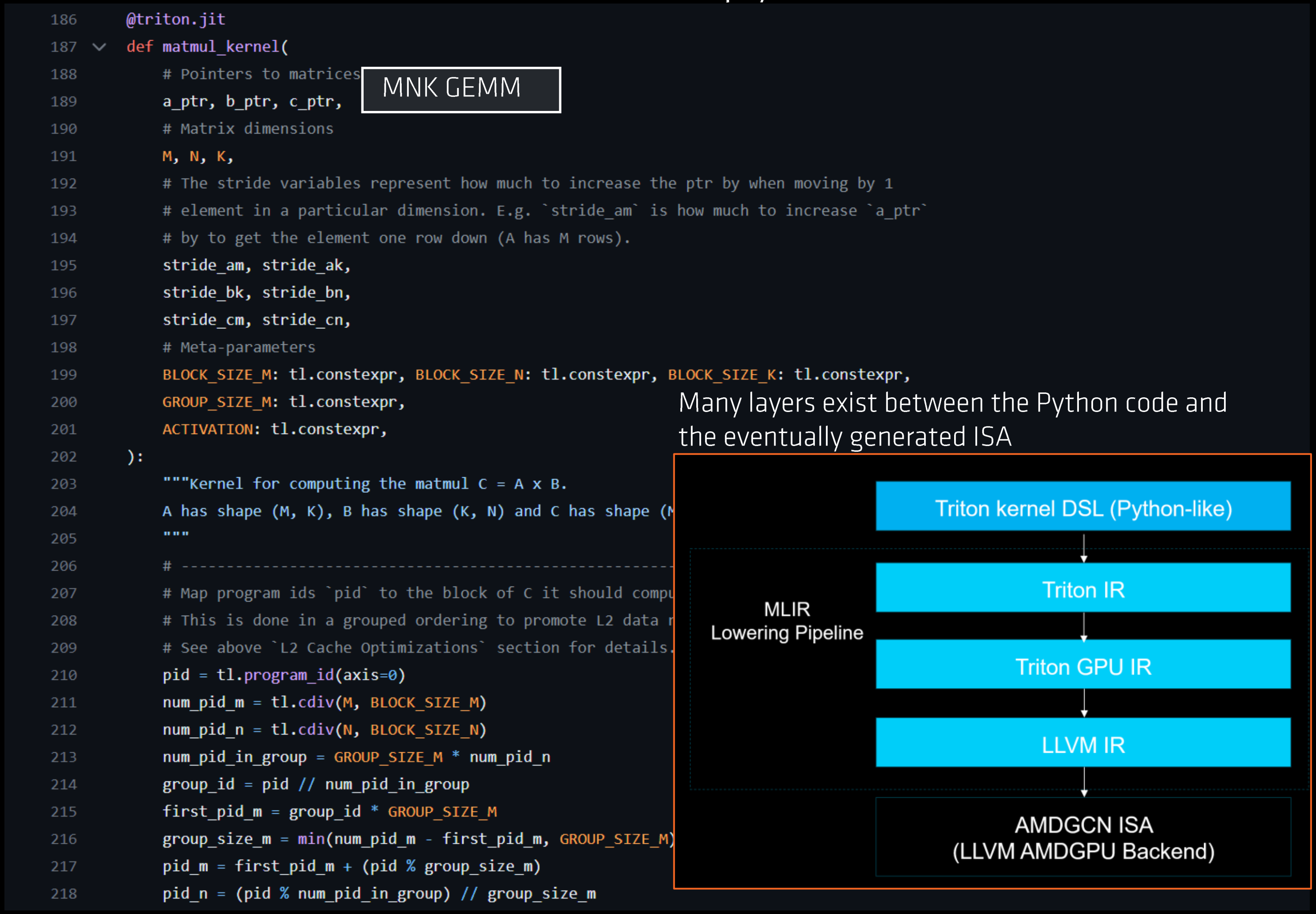
Challenges Instrumenting MLIR-GPU Frameworks

- Instrumentation on the GPU is not as straightforward as CPUs
- Instrumentation data, generated on the GPU, must somehow be moved to the CPU to be available for the user (i.e., displayed to the terminal or written to disk)
  - "hostcalls", how printf is implemented on the GPU, incur substantial runtime overheads
  - Example case study: a *very small* vLLM attention model memory trace generates 5-10 GBs of output data
- GPUs have a very specific programming and memory model
  - Ex: Sharing data across the entire set of all GPU threads is not straightforward and very expensive
- In LLVM, CPU and GPU code is compiled into separate modules that can not be linked together
- Instrumentation functions are often written with Clang (HIP/CUDA) based tool chains
  - MLIR based tool chains have no reason to link in the Clang driver and runtime libraries
  - Inline ASM is usually treated as a "black box" by the compiler and can even break some optimizations
- In Triton, for example, only the GPU code is compiled through MLIR/LLVM pipeline. Host (CPU) code is called through the interpreted Python wrapper layer
  - Instrumenting through the MLIR/LLVM pipeline will only see the GPU kernel itself not the associated host kernel launch API code

MLIR Based ML Compiler Framework Overview

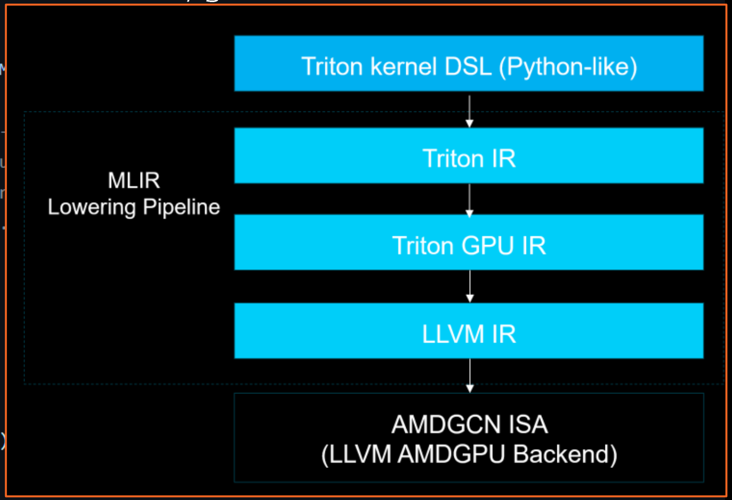
- MLIR has become a popular frontend for Python based machine learning compilers
  - Triton, IREE, PyTorch
- MLIR-based ML framework GPU kernel code is intentionally hidden from the user
- The machine learning compiler makes performance critical (tuning, tiling, etc.) choices behind the scenes
- The focus for an analysis tools depends on the perspective
  - HW architects: memory access patterns
  - ML compiler developers: data movement and compute overlap/pipelining opportunities, intra-kernel timing
  - Users/kernel writers: memory coalescing, bank conflicts, tuning bottlenecks

Triton Matrix Multiply



```
@triton.jit
def matmul_kernel(
    # Pointers to matrices
    a_ptr, b_ptr, c_ptr,
    # Matrix dimensions
    M, N, K,
    # The stride variables represent how much to increase the ptr by when moving by 1
    # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
    # by to get the element one row down (A has M rows).
    stride_am, stride_ak,
    stride_bk, stride_bn,
    stride_cm, stride_cn,
    # Meta-parameters
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
    GROUP_SIZE_M: tl.constexpr,
    ACTIVATION: tl.constexpr,
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M
    """
    # -------------------------------------------------------------
    # Map program ids `pid` to the block of C it should compu
    # This is done in a grouped ordering to promote L2 data r
    # See above `L2 Cache Optimizations` section for details.
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + (pid % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m
```

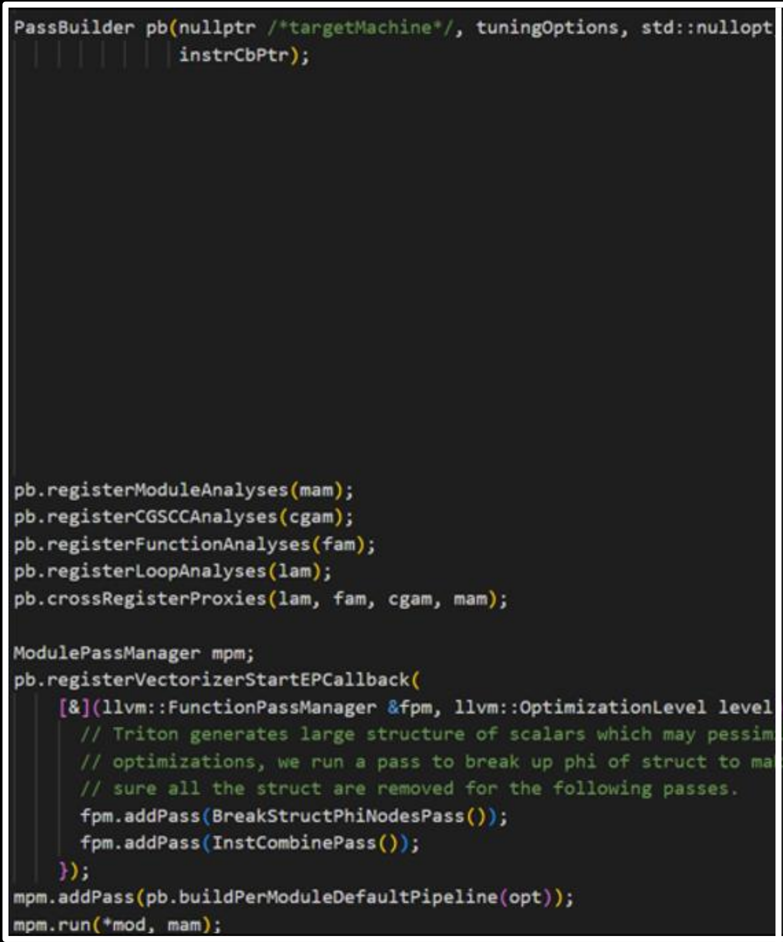
Many layers exist between the Python code and the eventually generated ISA


Triton kernel DSL (Python-like)
MLIR Lowering Pipeline
Triton IR
Triton GPU IR
LLVM IR
AMDGCN ISA (LLVM AMDGPU Backend)

https://github.com/triton-lang/triton/blob/main/python/tutorials/01-vector-add.py

Adding Instrumentation Infrastructure to MLIR Pass Pipeline

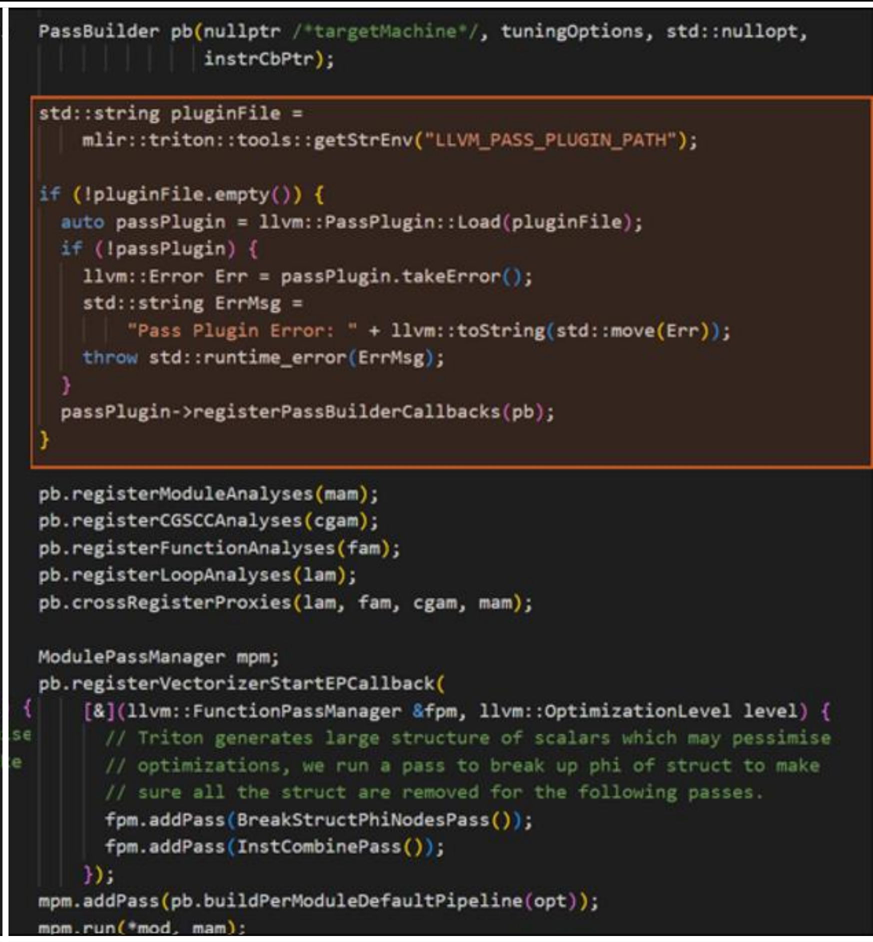
- Instrumentation can be done on either the MLIR or LLVM level
  - Instrumenting at the LLVM level seems to balance our current needs of lower-level control and upper-level source information
- Instrumentation is done using a combination of LLVM Plugins, analysis specific passes, and associated Clang generated CUDA/HIP kernels
- The instrumentation passes are added and registered in the “optimize module” phase of the MLIR flow
  - This is a common section in MLIR frameworks where the upper level MLIR interfaces with the LLVM pass manager



```
PassBuilder pb(nullptr /*targetMachine*/, tuningOptions, std::nullopt
               instrCbPtr);

pb.registerModuleAnalyses(mam);
pb.registerCGSCCAnalyses(cgam);
pb.registerFunctionAnalyses(fam);
pb.registerLoopAnalyses(lam);
pb.crossRegisterProxies(lam, fam, cgam, mam);

ModulePassManager mpm;
pb.registerVectorizerStartEPCallback(
    [&](llvm::FunctionPassManager &fpm, llvm::OptimizationLevel level
      // Triton generates large structure of scalars which may pessim se
      // optimizations, we run a pass to break up phi of struct to ma e
      // sure all the struct are removed for the following passes.
      fpm.addPass(BreakStructPhiNodesPass());
      fpm.addPass(InstCombinePass());
    });
mpm.addPass(pb.buildPerModuleDefaultPipeline(opt));
mpm.run(*mod, mam);
```



```
PassBuilder pb(nullptr /*targetMachine*/, tuningOptions, std::nullopt,
               instrCbPtr);

std::string pluginFile =
    mlir::triton::tools::getStrEnv("LLVM_PASS_PLUGIN_PATH");

if (!pluginFile.empty()) {
  auto passPlugin = llvm::PassPlugin::Load(pluginFile);
  if (!passPlugin) {
    llvm::Error Err = passPlugin.takeError();
    std::string ErrMsg =
        "Pass Plugin Error: " + llvm::toString(std::move(Err));
    throw std::runtime_error(ErrMsg);
  }
  passPlugin->registerPassBuilderCallbacks(pb);
}

pb.registerModuleAnalyses(mam);
pb.registerCGSCCAnalyses(cgam);
pb.registerFunctionAnalyses(fam);
pb.registerLoopAnalyses(lam);
pb.crossRegisterProxies(lam, fam, cgam, mam);

ModulePassManager mpm;
pb.registerVectorizerStartEPCallback(
    [&](llvm::FunctionPassManager &fpm, llvm::OptimizationLevel level) {
      // Triton generates large structure of scalars which may pessimise
      // optimizations, we run a pass to break up phi of struct to make
      // sure all the struct are removed for the following passes.
      fpm.addPass(BreakStructPhiNodesPass());
      fpm.addPass(InstCombinePass());
    });
mpm.addPass(pb.buildPerModuleDefaultPipeline(opt));
mpm.run(*mod, mam);
```

Implementation Details

- Instrumentation pass
  - Clones the kernel and adds an extra kernel argument for host-device communication
  - Loads pre-compiled bitcode file of instrumentation function
  - Merges the instrumentation module into the MLIR generated module and resolves symbol conflicts
  - Injects the instrumentation function at pre-defined areas of interest
- Runtime component of instrumentation module intercepts the original kernel and swaps out the cloned/instrumented one
- The instrumentation manages Host-Device communication
  - GPU instrumentation is written to a device side global memory buffer
  - If a GPU wave tries to write to the device side data buffer but finds that there is insufficient space
    - Signals the host code to empty the buffer
    - Once the host signals the wave that the buffer has been emptied, the wave then proceeds

```
define amdgpu_kernel void @add_kernel(ptr addrspace(1) nocapture
readonly %0, ptr addrspace(1) nocapture readonly %1, ptr
addrspace(1) nocapture writeonly %2, i32 %3){
  %5 = tail call i32 @llvm.amdgcn.workgroup.id.x(), !dbg !11
  %6 = shl i32 %5, 10, !dbg !12
  %7 = tail call i32 @llvm.amdgcn.workitem.id.x(), !dbg !13
  %8 = shl i32 %7, 2, !dbg !13
  %9 = and i32 %8, 1020, !dbg !13
  %10 = or disjoint i32 %9, %6, !dbg !14
  %11 = icmp slt i32 %10, %3, !dbg !15
  br i1 %11, label %.critedge, label %.critedge2, !dbg !16
```

```
define amdgpu_kernel void @add_kernelPv(ptr addrspace(1) nocapture
readonly %0, ptr addrspace(1) nocapture readonly %1, ptr
addrspace(1) nocapture writeonly %2, i32 %3, ptr addrspace(1) %4){
  %5 = tail call i32 @llvm.amdgcn.workgroup.id.x(), !dbg !66
  %6 = shl i32 %5, 10, !dbg !67
  %7 = tail call i32 @llvm.amdgcn.workitem.id.x(), !dbg !68
  %8 = shl i32 %7, 2, !dbg !68
  %9 = and i32 %8, 1020, !dbg !68
  %10 = or disjoint i32 %9, %6, !dbg !69
  %11 = icmp slt i32 %10, %3, !dbg !70
  br i1 %11, label %.critedge, label %.critedge2, !dbg !71
```

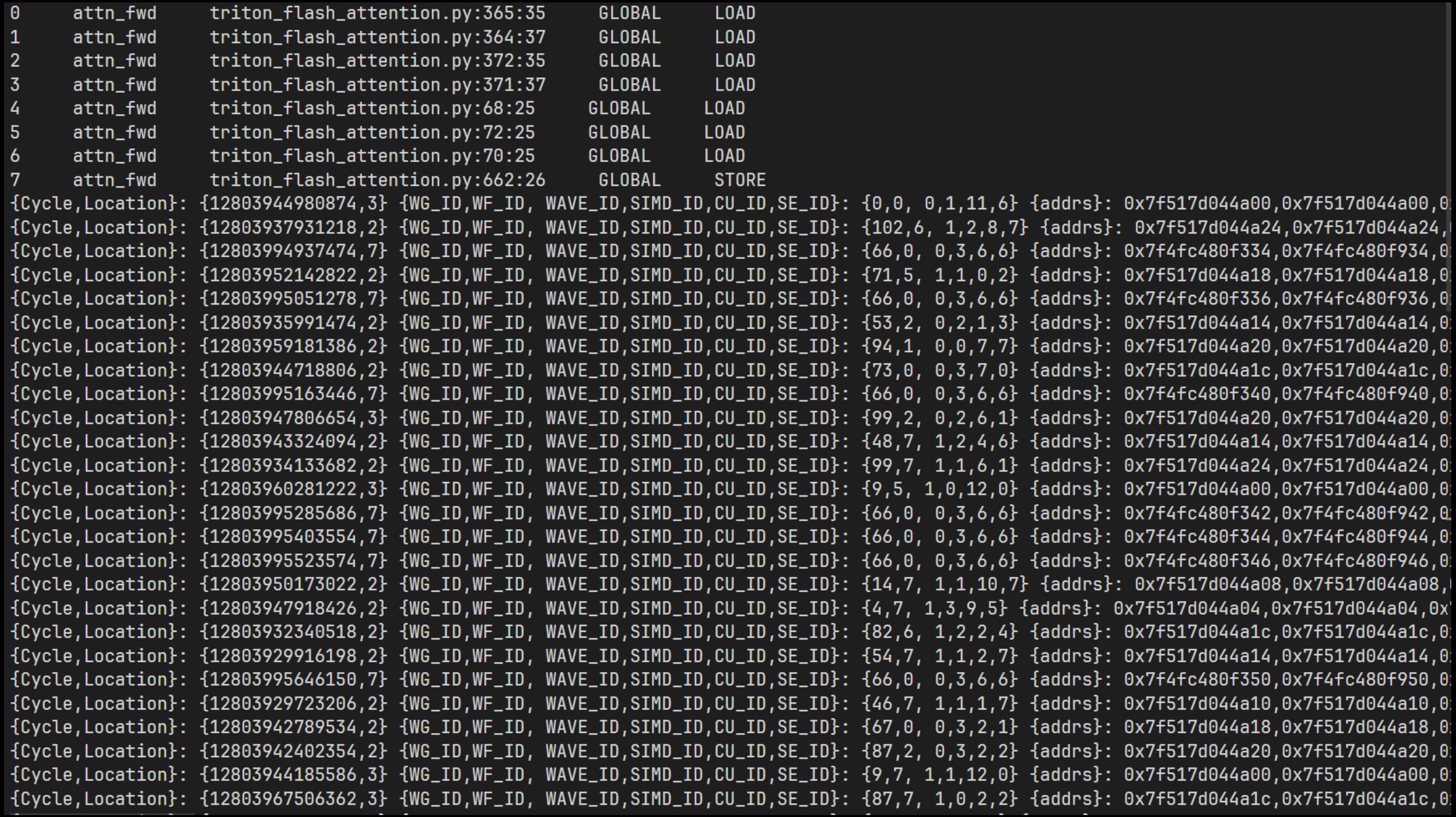
Triton Memory Trace Example

- Memory traces track and store the virtual memory access pattern of a kernel
- Memory traces instrumentation pass
  - Adds a compiler pass at every global load/store that
    - Inserts a GPU function that calculates the per wave virtual addresses
  - One address for each thread per wave
    - AMD GPUs have 64 threads per wave
  - Address meta data is also added to the trace
    - Source location
    - MLIR level objects (e.g., source tensor
    - Timestamps
    - Wave, XCD, SIMD, CU, etc. IDs

```
__attribute__((used)) __device__ void
memoryTrace(void *addressPtr, uint32_t LocationId, void *hostBufferPtr) {
 if(isSharedMemPtr(addressPtr))
   return;
  uint64_t address = reinterpret_cast<uint64_t>(addressPtr);
  // Mask of the active threads in the wave
  int activeThreadMask = __builtin_amdgcn_read_exec();
  // Find first active thread in the wave by finding the position of the least
  // significant bit set to 1 in the activeThreadMask
  const int firstActiveLane = __ffsll(activeThreadMask) - 1;
  uint64_t addrArray[WaveFrontSize];
  for (int i = 0; i < WaveFrontSize; i++) {
    addrArray[i] = __shfl(address, i, WaveFrontSize);
  }
  uint32_t Lane =
      __builtin_amdgcn_mbcnt_hi(~0u, __builtin_amdgcn_mbcnt_lo(~0u, 0u));

  if (Lane == firstActiveLane) {
    uint64_t MemTraceData = reinterpret_cast<MemTraceData_t>(hostBufferPtr);
    unsigned int hw_id = 0;
    uint64_t Time = 0;
    unsigned int xcc_id = 0;
    Time = __builtin_amdgcn_s_memrealtime();
    asm volatile("s_getreg_b32 %0, hwreg(HW_REG_HW_ID)" : "=s"(hw_id));
    MemTraceData.Cycle = Time;
    MemTraceData.LocationId = LocationId;
    MemTraceData.WaveId = hw_id & 0xf;
    MemTraceData.SIMD = (hw_id & 0x30) >> 4;
    MemTraceData.CU = (hw_id & 0xf00) >> 8;
    MemTraceData.SE = (hw_id & 0xe000) >> 13;
    MemTraceData.XCD = xcc_id;
    for (int i = 0; i < WaveFrontSize; i++)
        MemTraceData.addrArray[i] = addrArray[i];
    }
}
```

Memory Trace Example Output



```
0     attn_fwd     triton_flash_attention.py:365:35     GLOBAL     LOAD
1     attn_fwd     triton_flash_attention.py:364:37     GLOBAL     LOAD
2     attn_fwd     triton_flash_attention.py:372:35     GLOBAL     LOAD
3     attn_fwd     triton_flash_attention.py:371:37     GLOBAL     LOAD
4     attn_fwd     triton_flash_attention.py:68:25     GLOBAL     LOAD
5     attn_fwd     triton_flash_attention.py:72:25     GLOBAL     LOAD
6     attn_fwd     triton_flash_attention.py:70:25     GLOBAL     LOAD
7     attn_fwd     triton_flash_attention.py:662:26     GLOBAL     STORE
{Cycle,Location}: {12803944980874,3} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {0,0, 0,1,11,6} {addrs}: 0x7f517d044a00,0x7f517d044a00,0
{Cycle,Location}: {12803937931218,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {102,6, 1,2,8,7} {addrs}: 0x7f517d044a24,0x7f517d044a24,
{Cycle,Location}: {12803994937474,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f334,0x7f4fc480f934,0
{Cycle,Location}: {12803952142822,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {71,5, 1,1,0,2} {addrs}: 0x7f517d044a18,0x7f517d044a18,0
{Cycle,Location}: {12803995051278,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f336,0x7f4fc480f936,0
{Cycle,Location}: {12803935991474,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {53,2, 0,2,1,3} {addrs}: 0x7f517d044a14,0x7f517d044a14,0
{Cycle,Location}: {12803959181386,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {94,1, 0,0,7,7} {addrs}: 0x7f517d044a20,0x7f517d044a20,0
{Cycle,Location}: {12803944718806,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {73,0, 0,3,7,0} {addrs}: 0x7f517d044a1c,0x7f517d044a1c,0
{Cycle,Location}: {12803995163446,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f340,0x7f4fc480f940,0
{Cycle,Location}: {12803947806654,3} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {99,2, 0,2,6,1} {addrs}: 0x7f517d044a20,0x7f517d044a20,0
{Cycle,Location}: {12803943324094,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {48,7, 1,2,4,6} {addrs}: 0x7f517d044a14,0x7f517d044a14,0
{Cycle,Location}: {12803934133682,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {99,7, 1,1,6,1} {addrs}: 0x7f517d044a24,0x7f517d044a24,0
{Cycle,Location}: {12803960281222,3} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {9,5, 1,0,12,0} {addrs}: 0x7f517d044a00,0x7f517d044a00,0
{Cycle,Location}: {12803995285686,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f342,0x7f4fc480f942,0
{Cycle,Location}: {12803995403554,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f344,0x7f4fc480f944,0
{Cycle,Location}: {12803995523574,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f346,0x7f4fc480f946,0
{Cycle,Location}: {12803950173022,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {14,7, 1,1,10,7} {addrs}: 0x7f517d044a08,0x7f517d044a08,
{Cycle,Location}: {12803947918426,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {4,7, 1,3,9,5} {addrs}: 0x7f517d044a04,0x7f517d044a04,0x
{Cycle,Location}: {12803932340518,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {82,6, 1,2,2,4} {addrs}: 0x7f517d044a1c,0x7f517d044a1c,0
{Cycle,Location}: {12803929916198,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {54,7, 1,1,2,7} {addrs}: 0x7f517d044a14,0x7f517d044a14,0
{Cycle,Location}: {12803995646150,7} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {66,0, 0,3,6,6} {addrs}: 0x7f4fc480f350,0x7f4fc480f950,0
{Cycle,Location}: {12803929723206,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {46,7, 1,1,1,7} {addrs}: 0x7f517d044a10,0x7f517d044a10,0
{Cycle,Location}: {12803942789534,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {67,0, 0,3,2,1} {addrs}: 0x7f517d044a18,0x7f517d044a18,0
{Cycle,Location}: {12803942402354,2} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {87,2, 0,3,2,2} {addrs}: 0x7f517d044a20,0x7f517d044a20,0
{Cycle,Location}: {12803944185586,3} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {9,7, 1,1,12,0} {addrs}: 0x7f517d044a00,0x7f517d044a00,0
{Cycle,Location}: {12803967506362,3} {WG_ID,WF_ID, WAVE_ID,SIMD_ID,CU_ID,SE_ID}: {87,7, 1,0,2,2} {addrs}: 0x7f517d044a1c,0x7f517d044a1c,0
```

vLLM Triton AMD GPU backend, on MI300X, offline_inference.py example:
https://github.com/vllm-project/vllm/blob/main/examples/offline_inference.py

Current Status and Future Development Roadmap

- Implemented Instrumentation Passes
  - Intra-kernel timestamps for region-based timing
  - Memory access pattern traces
  - Memory coalescing and shared memory bank conflicts
- Interested in hearing from users to influence future development road map
- The infrastructure to use the instrumentation passes
  - Triton
    - https://github.com/triton-lang/triton/pull/3953 (May 2024)
  - IREE
    - https://github.com/iree-org/iree/pull/18347 (August 2024)
- Instrumentation passes are open-source and extendible/modifiable by users
  - https://github.com/CRobeck/instrument-amdgpu-kernels

Copyright and disclaimer