Comco: A MLIR-Based Intermediate Representation for CUDA Kernel Fusion

Matthew Nappo
University of Rochester
CSC 290
mnappo@u.rochester.edu

Abstract

Kernel fusion is an optimization technique for GPU kernels aimed at reducing the overhead of host-to-device and device-to-host data transfers. These expensive data transfers often prove to be a significant bottleneck in many GPU and machine learning workloads. By fusing kernels, more work can be done on the device before data is copied out, mitigating this transfer overhead. Traditionally, kernel fusion involves manually identifying frequently sequenced kernels and combining them into a single fused kernel. The Comco compiler[1], however, seeks to automate this process by generating custom fused kernels directly from user source code. The Comco IR is a custom dialect within the Multi-Level Intermediate Representation [11] framework that captures common machine learning operations such as matrix multiplication and AllReduce and automatically applies kernel fusion when applicable.

1. Introduction

The size of machine learning (ML) models has been steadily increasing, leading to higher computational demands during training and inference. To address this, the development of domain-specific languages (DSLs) tailored for machine learning has gained much popularity. These DSLs are designed to streamline the development of machine learning routines used for model preprocessing, training, optimization, and inference. Languages like PyTorch [8, 12] and JAX [2] have gained wide adoption due to their ability to express complex kernels in a high-level syntax while also efficiently compiling to native GPU code.

Designing an effective DSL for ML requires a careful balance between expressibility and performance. Comco addresses one aspect of this tradeoff by providing an intermediate representation based on MLIR. The Comco IR enables kernel fusion, although it is not limited to this particular optimization.

[1] Source code is available at https://github.com/mattnappo/comco.

Currently, Comco IR exposes a relatively small set of operations to the user. However, the underlying kernel fusion engine is generalizable to future operations and extensions to the IR.

This work makes the following contributions:

1. Introduces the Comco IR, a MLIR dialect capable of representing common machine learning routines. Comco provides operations for matrix multiplication and AllReduce across a GPU.
2. Provides a set of transformations on user-supplied Comco IR code to fuse operations into a single CUDA kernel.
3. Implements the Comco compiler to incrementally lower Comco IR code into an optimizable form before using MLIR's final GPU lowering [5].

2. Background

2.1. MLIR

Multi-Level Intermediate Representation (MLIR) [11] is an extensible intermediate representation technology within the LLVM [10] ecosystem designed to represent arbitrary dataflow graphs and enable a wide range of optimizations. One of the key features of MLIR is its support for dialects, which are custom sets of operations, types, and transformations that extend the core MLIR language. Dialects allow users to define domain-specific abstractions, and they come with associated optimizations and lowering passes that transform the custom operations and types into operations in the core MLIR dialects. This allows for integration with LLVM-based backends, where the resulting IR can be lowered to LLVM IR for further optimization and code generation via mature compilers like Clang.

Another notable feature of MLIR is its pattern rewrite system [7], which provides a powerful mechanism for DAG-to-DAG transformation. This rewrite system is the primary feature used to implement Comco's kernel fusion engine.

2.2. Kernel Fusion

Kernel fusion [13] is a common optimization technique that combines two or more independent compute kernels into a single fused kernel that performs the same overall computation in a more efficient manner. The primary benefit of kernel fusion is the elimination of redundant data transfers between the host and the device memory. When two kernels are executed sequentially, the output of the first kernel is typically written to device or host memory and then read by the second kernel, requiring costly unnecessary data copies. By fusing these kernels, this intermediate data transfer is completely avoided, significantly improving performance.

Fusion is particularly beneficial in scenarios where there is a clear data dependency between kernels, such as when the output of the first kernel serves as the input for the second. In this case, kernel fusion allows both computations to occur entirely on the device, eliminating the need to copy data out of and back into the device memory, which is frequently a major bottleneck.

However, kernel fusion in the general case requires careful analysis. For example, dependencies of the second kernel may not be statically available, or even determinable, at the time the first kernel is launched. This can lead to incorrect fusion if not handled properly. Additionally, certain fusion opportunities may require reshaping in-memory representations of input data. It is also not always trivial to determine exactly which operations should be included in the fused kernel. CocoNet [9], for example, determines that fusion is not always more performant.

2.3. Related Work

CocoNet is a DSL and machine learning compiler that overlaps computation with communication, performs kernel fusion and operator reordering, and distributes workloads across multi-GPU clusters. CocoNet [9] provides hand-altered NCCL [6] and cuBLAS [1] routines that interweave communication with computation. CocoNet's compiler realizes when these kernels are applicable, and emits a fused CUDA kernel with the optimizations.

However, CocoNet's code generator is rudimentary, relying entirely on string concatenation and lacking any formal intermediate representation. This limitation inspired the development of Comco: to achieve similar optimizations as CocoNet, but with the advantage of using MLIR for more efficient code generation and optimization.

3. The Comco IR

The Comco IR is presently quite primitive, supporting a limited set of essential operations. These include kernel definition, matrix multiplication, AllReduce across a grid, tensor allocation, and tensor load/stores. Comco also provides wrappers around the core MLIR operations `linalg::MatmulOp` and `gpu::AllReduce`. However, these wrappers lower directly to their corresponding operations in the core MLIR dialects. Both variants are valid representations within the Comco IR.

Comco also offers legal operations for commonly used neural network functions, such as Softmax, ReLU, Dropout, and Update. However, the framework does not yet perform automatic fusion for these operators, though this is the first area to explore in future work.

```
module {
   comco.func @comco_func() {
      %A = tensor.empty() : tensor<32
          x32xf32>
      %B = tensor.empty() : tensor<32
          x32xf32>
      %C = tensor.empty() : tensor<32
          x32xf32>

      %0 = arith.constant 3.14 : f32

      %out = linalg.matmul
         ins(%A, %B: tensor<32x32xf32>,
             tensor<32x32xf32>)
         outs(%C: tensor<32x32xf32>) ->
             tensor<32x32xf32>

      %s = gpu.all_reduce add %0 uniform {}
           : (f32) -> (f32)

      comco.return
   }
}
```

Figure 1. An example program in Comco IR which allocates tensors, performs a matrix multiplication, and an AllReduce.

4. Comco DAG-to-DAG Transformations

This section details the main transformations used to implement the Comco IR and kernel fusion engine within the Comco compiler.

4.1. Lowering Comco Functions

The `comco.func` and `comco.return` operations define a compute kernel which will ultimately be translated to the GPU. All user Comco code must be supplied within this region. These operators are first lowered directly to their corresponding core dialect operations: `func::FuncOp` and `func::ReturnOp`. The code in a `comco.func` will act as the main control thread running on the host. It is responsible for managing CUDA stream 0 and coordinating the launches of kernels. Additionally, this

pass ensures that the `gpu.container_module` attribute is present on the parent module, a necessary precondition for subsequent GPU lowerings.

4.2. Memory Management

While `tensors` provide a high-level abstraction useful during various analyses, they do not correspond to physical memory locations and are not suitable for lower-level stages of Comco IR. Thus, a memory management lowering is necessary early on in the pipeline. By using a combination of MLIR's One-Shot Bufferization [3] pass and custom GPU-specific `memref` annotations, Comco is able to efficiently map from tensors to `memrefs`. Memory layout is still an active area of MLIR development. These APIs and lowerings are highly unstable and often depend on the specific device architecture. The GPU dialect in MLIR provides abstractions over the three GPU address spaces: private, workgroup, and global. Comco generally places tensors in `workgroup` memory when possible.

4.3. Kernel Inlining & Op Movement

After `comco.func` lowering, the Comco compiler begins its analysis of the compute-intensive operators within the routine. This transformation searches for `MatMul` and `AllReduce` operations and places them inside a `gpu::GPULaunchOp`. The block and grid dimensions of the kernel launch are naively determined based on the dimensions of the input tensors: every rank-2 tensor element will get its own thread. This is not scalable and will need to be improved in future versions of Comco.

After the inlining phase, these kernels will be outlined using `mlir::GpuKernelOutliningPass` provided by MLIR which places them into their own `gpu::GPUFuncOp`. This lowering is more of an organizational transformation, but it does require that `memrefs` from the parent scope are explicitly passed into the `gpu::LaunchFuncOp`. This rewrite pattern also handles this transformation.

The transformations up to this point form a control group since fusion has not occured; each operation is scheduled in its own kernel. IR code at this level of transformation will be lowered to the LLVM dialect of MLIR, then to LLVM IR, then to native GPU code.

4.4. Kernel Fusion

This transformation analyzes the input-output dependencies between the matrix multiplication and AllReduce operations in the original Comco IR. If the output of one operation serves as the input to the next, or if the operations are independent, they are fused into a single kernel. The rewrite pattern driver scans the IR for such relationships, and when a match is found, it merges the operations into a shared kernel while ensuring that the original program order is preserved.

After fusion, the compiler re-analyzes the IR of the host routine to ensure that the correct fused kernel is launched. This may require updating kernel arguments as well as adjusting block and grid dimensions to properly invoke the fused kernel's new structure.

```
func.func @main() {
   %A = tensor.empty() : tensor<32x32xf32>
   %B = tensor.empty() : tensor<32x32xf32>
   %C = tensor.empty() : tensor<32x32xf32>
   %1 = index.constant 1
   gpu.launch
       blocks (%gx, %gy, %gz)
       in (%ggx = %1, %ggy = %1, %ggz = %1)
      threads (%tx, %ty, %tz)
      in (%ttx = %1, %tty = %1, %ttz = %1)
          {
      %out = linalg.matmul
         ins(%A, %B: tensor<32x32xf32>,
             tensor<32x32xf32>)
         outs(%C: tensor<32x32xf32>) ->
             tensor<32x32xf32>
      gpu.terminator
   }
   gpu.launch
       blocks (%gx, %gy, %gz)
       in (%ggx = %1, %ggy = %1, %ggz = %1)
      threads (%tx, %ty, %tz)
      in (%ttx = %1, %tty = %1, %ttz = %1)
          {
      %c = arith.constant 3.14 : f32
      %s = gpu.all_reduce add %c uniform {}
           : (f32) -> (f32)
      gpu.terminator
   }
   func.return
}
```

Figure 2. An example of the kernel inlining transformation applied to the Comco IR code in Figure 1.

5. Compute Operator Lowering

Now Comco must lower the `linalg::MatmulOp` and `gpu::GPUAllReduce` operations. There are many approaches to this transformation. One approach is to lower these operations to LLVM function calls that invoke dynamically-linked shared libraries such as NCCL [6] and cuBLAS [1]. This was the original approach taken by Comco via the EmitC [4] dialect of MLIR. One limitation of this method was ensuring ABI compatibility between the LLVM-generated code and the expectations of the external libraries. As a result, this approach was abandoned, though

```
module attributes {gpu.container_module} {
  gpu.module @comco_func_kernel {
    gpu.func @comco_func_kernel(%arg0:
        memref<32x32xf32>, %arg1: memref<32
        x32xf32>, %arg2: memref<32x32xf32>)
        kernel attributes {known_block_size =
         array<i32: 4, 1, 1>, known_grid_size
         = array<i32: 4, 1, 1>} {
      // excluded: GPU launch dimension
          extraction
      linalg.matmul ins(%arg0, %arg1 : memref
          <32x32xf32>, memref<32x32xf32>)
          outs(%arg2 : memref<32x32xf32>)
      %cst = arith.constant 3.140000e+00 :
          f32
      %0 = gpu.all_reduce add %cst uniform {
      } : (f32) -> f32
      gpu.return
    }
  }
  func.func @main() {
    %alloc = memref.alloc() {alignment = 64
        : i64, memory_space = #gpu.
        address_space<global>} : memref<32
        x32xf32>
    %alloc_0 = memref.alloc() {alignment =
        64 : i64, memory_space = #gpu.
        address_space<global>} : memref<32
        x32xf32>
    %alloc_1 = memref.alloc() {alignment =
        64 : i64, memory_space = #gpu.
        address_space<global>} : memref<32
        x32xf32>
    %c1 = arith.constant 1 : index
    %c4 = arith.constant 4 : index
    gpu.launch_func @comco_func_kernel::
        @comco_func_kernel blocks in (%c4, %
        c1, %c1) threads in (%c4, %c1, %c1)
        args(%alloc : memref<32x32xf32>, %
        alloc_0 : memref<32x32xf32>, %alloc_1
         : memref<32x32xf32>)
    %c1_2 = arith.constant 1 : index
    %c4_3 = arith.constant 4 : index
    return
  }
}
```

Figure 3. An example of the kernel fusion transformation applied to the Comco IR code in Figure 3.

it is still viable at least in principle.

Instead, Comco lowers matrix multiplications to a naive kernel in the `affine` MLIR dialect, and lowers `AllReduce` operations using MLIR's `GpuAllReduceRewriter`. While this is not the most performant approach, it does provide correct semantics.

6. Final Code Generation

At this stage of the Comco lowering pipeline, all high-level operations have been rewritten. The IR is in a suitable state to undergo the `GPULowerToNVVM` pipeline provided by MLIR [5]. This serializes all device code into the `cubin` format and translates all MLIR to LLVM IR. The resultant LLVM IR can be compiled by a tool such as Clang, or can be JIT-compiled by linking the CUDA runtime libraries with MLIR's `mlir-cpu-runner` utility.

7. Future Work & Limitations

Directions of future work include:

- Supporting a wider range of operations to increase the expressibility of Comco.
- Generating more sophisticated kernels for the compute and network-intensive operations.
- Providing optimizations that automatically overlap computation with communication.
- Implementing an autotuner which generates multiple kernels with varying block/grid dimensions, fusion orders, and operation lowerings.
- Supporting multi-GPU collective communications to enable Comco to run efficiently within a distributed system.

8. Conclusion

This work introduces Comco, a MLIR-based intermediate representation and compiler that automatically fuses matrix multiplication and `AllReduce`-intensive kernels.

References

[1] cuBLAS: CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/. 2, 3

[2] JAX: High performance array computing. https://jax.readthedocs.io/en/latest/index.html. 1

[3] MLIR Bufferization. https://mlir.llvm.org/docs/Bufferization/. 3

[4] MLIR 'emitc' Dialect. https://mlir.llvm.org/docs/Dialects/EmitC/. 3

[5] MLIR 'gpu' Dialect. https://mlir.llvm.org/docs/Dialects/GPU/. 1, 4

[6] NCCL: NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl. 2, 3

[7] Pattern Rewriting: Generic DAG-to-DAG Rewriting. https://mlir.llvm.org/docs/PatternRewriter/. 1

[8] Jason Ansel, Edward Yang, Horace He, and et al. Gimelshein. Pytorch 2: Faster machine learning through

dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery. 1

[9] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Coconet: Co-optimizing computation and communication for distributed machine learning. *CoRR*, abs/2105.05720, 2021. 2

[10] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. pages 75– 86, 04 2004. 1

[11] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020. 1

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, and et al. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. 1

[13] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, 2010. 2