Kokkos and MLIR

Brian Kelley and Kim Liegeois, Sandia National Laboratories

Kokkos User Group Meeting 2023

December 12, 2023

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2023-14543C

Introduction

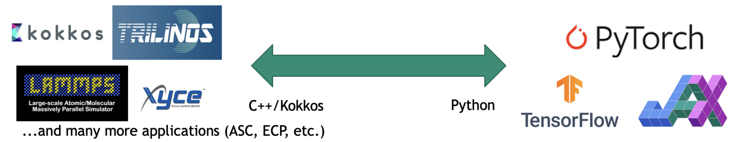
- MLIR
- Why MLIR is useful to the Kokkos ecosystem
- Previous Work
- Examples: ResNet18 and SpMV
- Ongoing Work
- Conclusion

Overview

- MLIR: Multi-Level Intermediate Representation
- Part of the LLVM project
- Like LLVM IR, an SSA language designed to be automatically analyzed and optimized by a compiler
- Unlike LLVM, includes high-level operations, e.g. matrix multiplication
- Operations organized into families called **dialects**

- Some built-in dialects, higher to lower level:
  - `linalg`: high-level tensor, matrix and vector operations
  - `scf`: structured control flow: parallel loops, `if`, `while`
  - `memref`: allocate and access multidimensional arrays
  - `gpu`: heterogeneous memory and GPU kernels in a CUDA-like model
  - `llvm`: LLVM IR instructions (making MLIR a superset of LLVM)
- MLIR includes **passes** which analyze and transform a program
  - Lowering: convert high-level operation(s) to lower-level equivalent
  - Optimizing: e.g. loop fusion, tiling, strength reduction...
- Users can also create their own dialects and passes

Motivation



Why bring Kokkos and MLIR together?

- Interfacing between Python ML frameworks (PyTorch etc.) and Kokkos C++ HPC codes
  - Direct interfacing: increased developer effort, and introduced dependencies on Python packages
  - Instead, MLIR can be used to automatically generate C++ source code from Python ML models. Integrate this into Kokkos-based HPC codes.

Why bring Kokkos and MLIR together?

- Automation of tedious programming tasks
  - Host-device memory migration
  - Parameter search and autotuning
  - Overlapping of independent computations using execution space instances

Previous Work

- torch-mlir: external open-source project that compiles PyTorch models to MLIR
  - Includes end-to-end infrastructure to compile, load and run a model within Python
- Kokkos $\rightarrow$ MLIR emitter: 2022 exploratory project at Sandia
  - Project investigated whether MLIR was a suitable interface between Kokkos and machine learning frameworks
  - Emitter generates Kokkos C++ source code from MLIR program with mid-level dialects
  - Successfully emitted ResNet18 pre-trained model, then compiled and ran with CUDA backend
- Sparse tensor dialect
  - Effort led by Dr. Aart Bik (Google) to add first-class sparse tensor support to MLIR
  - Describe tensor formats and layouts at high level: CRS, block structure, doubly-compressed dimensions, etc.
  - Parallel code generation from high-level `linalg` operations

Examples

- ResNet18 (CNN image classifier) end-to-end example.
- Begin with pre-trained PyTorch model:

```
ResNet(
(conv1): Conv2d(3, 64, ...)
(bn1): BatchNorm2d(64, ...)
(relu): ReLU(inplace=True)
(maxpool): MaxPool2d(kernel_size=3, ...)
(layer1): Sequential(...)
...
```

- Use torch-mlir to generate high-level MLIR
- All weights arrays are included in the MLIR as constant `memrefs`
- Then use built-in passes to lower to mid-level dialects
- These dialects most closely match Kokkos's level of abstraction

- `%N`: an SSA value (the result of an operation)
- `scf.parallel`: parallel N-dimensional loop
- `memref.load`: read a value from an array
- `arith.divf`: floating point division

```
scf.parallel (%arg1, %arg2, %arg3, ...) {
  %249 = memref.load %224[%arg1, %arg2]
  %250 = arith.divf %249, %cst : f32
  memref.store %250, %225[%arg1, ...]
  scf.yield
}
```

- Use the Kokkos emitter to generate C++ code, one operation at a time

```
Kokkos::parallel_for(Kokkos::MDRangePolicy<
exec_space, Kokkos::Rank<4>>(...),
KOKKOS_LAMBDA(int64_t unit_v1255, int64_t ...)
{
  int64_t v1255 = v10 + unit_v1255 * v7;
  int64_t v1256 = v10 + unit_v1256 * v7;
  int64_t v1257 = v10 + unit_v1257 * v7;
  int64_t v1258 = v10 + unit_v1258 * v7;
  float v1259 = v249(v1255, v1256);
  float v1260 = v1259 / v2;
  ...
```

- Compile the C++ code to a shared library, with Kokkos linked in
- Load the library into Python with CTypes, and run model inference

```
# Example.py
import kokkosModule
predictions = kokkosModule.forward(image)
```

image: 

```
predictions:
[('Labrador retriever', 70.657),
 ('golden retriever', 4.988), ...]
```

- Or, use the C++ code directly from an existing Kokkos program
- Takes image as input (RGB, 224×224) and returns probability vector (1000 classes)

```
// mlir_kokkos_module.cpp
Kokkos::View<float[1][1000], Kokkos::LayoutRight>
forward(Kokkos::View<float[1][3][224][224], Kokkos::LayoutRight> v1) {
  ...
}
```

- Use PyTACO (included with MLIR) to express tensor formats and computation
- Use sparse tensor dialect to generate parallel sparse matrix times vector kernel

```
A = pt.tensor([5, 5], [pt.dense, pt.compressed], dtype=pt.float64)
b = pt.tensor([A.shape[1]], [pt.dense], dtype=pt.float64)
c = pt.tensor([A.shape[0]], [pt.dense], dtype=pt.float64)
# fill A and b...
i, j = pt.get_index_vars(2)
c[i] = A[i,j] * b[j]
```

- Apply sparse tensor lowering pipeline
- Use Kokkos emitter to generate kernel with two-level parallelism

```
void kokkos_sparse_kernel_0(
  Kokkos::View<double*, ...> v1,  // c values
  Kokkos::View<size_t*, ...> v2,  // A row offsets
  Kokkos::View<size_t*, ...> v3,  // A column entries
  Kokkos::View<double*, ...> v4,  // A values
  Kokkos::View<double*, ...> v5   // b values
  ) {
typedef Kokkos::TeamPolicy<exec_space>::member_type member_type;
int league_size = (5 - 0 + 1 - 1) / 1;
Kokkos::TeamPolicy<exec_space> policy (league_size, Kokkos::AUTO(), Kokkos::AUTO() );
Kokkos::parallel_for(policy, KOKKOS_LAMBDA(member_type member)
{
  int64_t unit_v6 = member.league_rank ();
  int64_t v6 = 0 + unit_v6 * 1;
  ...
  Kokkos::parallel_reduce(
  Kokkos::TeamVectorRange(member, (v10 - v8 + 1 - 1) / 1),
  [&](const int64_t &unit_v12, double& v13)
  {
    ...
    double v17 = v15 * v16;
    v13 += v17;
  }, v11)
  // memref.store
  v1(v6) = v11;
  ...
}
```

Ongoing and Future Work

- Tensor partitioning and distributed computations
  - We have a prototype `partition` dialect
  - Can express a 2D block distributed SpMV now
- Use partitioning infrastructure to target spatial dataflow accelerators
  - These systems have many small processors in a grid, with fast communication between neighbors
  - Want to distribute sparse tensors and computations over processors
  - Have successfully run SpMV kernel on NextSilicon Maverick using OpenMP programming model
- Automatic differentiation at the MLIR level

Conclusion

- MLIR can express ML and HPC programs at varying levels of abstraction
- Transformations can exploit high-level information that would not be available to a conventional C++ compiler
- We are developing tools with MLIR to benefit HPC use cases:
  - MLIR $\rightarrow$ Kokkos C++ emitter: convert MLIR to portable Kokkos-based source code
  - Partition dialect: support tiled and distributed sparse tensors
  - Target novel spatial dataflow accelerators
- **Thank you! Any questions?**