Buddy Compiler: An MLIR-Based Compilation Framework for Deep Learning Co-Design

Speaker: Hongbin Zhang | hongbin2019@iscas.ac.cn
Authors: Hongbin Zhang (ISCAS) Liutong Han (ISCAS) Prathamesh Tagore (VJTI) Sen Yang(XUPT) Zikang Liu(ICT) Yuchen Li (ISCAS)
Buddy Compiler is a domain-specific compiler framework. We are building a co-design ecosystem based on MLIR and RISC-V. We hope to achieve deep co-design from DSL to DSA. Deep co-design for deep learning!
Buddy Compiler Overview

“Buddy System” for Domain-Specific Compilers | MLIR-Based Compilation Framework for Deep Learning Co-Design

Homepage: https://buddy-compiler.github.io/
GitHub: https://github.com/buddy-compiler
Buddy Compiler Overview

Frontends: Domain-Specific Libraries, DSL Framework, Deep Learning Frameworks Integration

Online Service: Ecosystem Entry (Demonstrate, Share, and Debug)

Backends:
- Hardware-Specific MLIR Dialect
- LLVM Backend
- Toolchains
- Emulators

Mid-end:
- Domain-Specific MLIR Dialect
- Optimization and Lowering
- Auto-Configure Mechanism for Multiple Backends

Compiler Framework (buddy-mlir)
- Hardware Dialects
- LLVM/MLIR Tools
- RISC-V Toolchain
- NVIDIA Toolkit
- RISC-V Custom ISA

End-to-End Application

Compiler As A Service (buddy-caas)

Benchmark Framework (buddy-benchmark)
- Utils / Tools
- Operation Level Benchmark
- Domain-Specific Benchmark
- Google Benchmark

Evaluation Report

Homepage: https://buddy-compiler.github.io/
GitHub: https://github.com/buddy-compiler

Benchmark Framework:
- Benchmark Cases for Multiple Levels
- Evaluation and Visualization Tools
MLIR and RISC-V are a perfect match for co-design! Because they are both modular and extensible. The unified ecosystem can unlock more co-design opportunities.
Buddy Compiler Deep Learning Co-Design

Multimodal Representations

Buddy Compiler Domain-Specific Dialects

Deep Learning Model Representations

MLIR TOSA / Linalg Dialect

Vector Representation
- Vector Dialect
- RVV Dialect
- LLVM VP Intrinsic

MLIR Core Dialects
- MemRef Dialect
- Affine Dialect
- SCF Dialect
  ... ...

Gemmini Dialects
- Gemmini Operation
- Gemmini Intrinsic Operation
- Custom LLVM Extension

LLVM | RISC-V GNU Toolchain | Emulators

SIMD Processor
( RISC-V P Extension )

Vector Processor
( RISC-V V Extension )

GPGPU
( e.g. Ventus )

DSA
(e.g. Gemmini)

Preprocessing + Deep Learning Workload
- Preprocessing Operation Optimization
- Unified Data Structure to Avoid Copy Overhead
- Potential Operation Fusion Opportunity

Compiler Passes + Hardware Architecture
- Design Representations for Hardware Features
- Configure Passes by Hardware Information
- Potential Auto-Tuning / DSE Opportunity
The key to co-design is **unified abstraction and presentation**. MLIR can unify **domain-specific applications and languages** together.
Image Processing


Audio Processing [2]

C++ Libraries

```
template<typename T, size_t N>
void fir(MemRef<float, N>* input, MemRef<T, N>* filter, MemRef<float, N>* output) {
    ...
    detail::mlir_ciface_dap_fir(input, filter, output);
}
```

Domain-Specific Operations

```
func.func @dap_fir(%in : memref<?xf32> ,
                  %filter : memref<?xf32>,
                  %out : memref<?xf32>) -> () {
    dap.fir %in, %filter, %out :
    memref<?xf32>, memref<?xf32>, memref<?xf32>
    return
}
```

[1] The origin image is from MediaStorm - https://www.ysjf.com/materialLibrary
def main() {
    var a<2, 2> = [1, 2, 3, 4, 5, 6];
    var b<2, 2> = [[1, 2], [3, 4]];
    print(a + b);
}

$ dsl-compiler tensor-add.toy -emit=jit

2.000000 4.000000
6.000000 8.000000

module {
    toy.func @main() {
        %0 = toy.constant dense<[1.0, 2.0, 3.0, 4.0]> : tensor<4xf64>
        %1 = toy.reshape(%0 : tensor<4xf64>) to tensor<2x2xf64>
        %2 = toy.constant dense<[1.0, 2.0]>
        %3 = toy.reshape(%2 : tensor<2xf64>) to tensor<2x2xf64>
        %4 = toy.add %1, %3 : (tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<*xf64>
        toy.print %4 : tensor<*xf64>
        toy.return
    }
}
IRs are unified, and optimization should not be fragmented.
No one wants to port an algorithm to every platform!
Optimization for Multiple Backends

1. Combination of Multiple Optimization Strategy
   - High-Level Optimization Algorithm
   - Compilation Optimization

2. Using MLIR Vector Dialect to Achieve Portable Optimization

3. Detect Target Hardware and Configure Optimization Pass

Broadcast-Based Vectorization Algorithm for Convolution Operation

MLIR Convolution Operation (Conv2D) Comparison
(Input Size: 1024 x 1024 Kernel Size: 3 x 3)
Of course, co-design should consider **hardware features**!
If you do need to expose those hardware features to the compiler, let's **add a new IR abstraction**.
RVV Tail Processing

\[
\begin{array}{cccccc}
\downarrow & & & \downarrow & \\
C[0] & C[1] & \cdots & C[?] & C[?]
\end{array}
\quad +
\begin{array}{cccccc}
B[0] & B[1] & \cdots & B[?] & B[?]
\end{array}
\Rightarrow
\begin{array}{cccccc}
C[0] & C[1] & \cdots & C[?] & C[?]
\end{array}
\]

- Get the application vector length (d) at runtime
- Mask-Based Approach
- Strip-Mining Approach

Information Required at Compile Time:
- Dynamic VL Configuration
  - AVL Configuration
  - LMUL Configuration
  - SEW Configuration
- Operations Dynamic VL Operands

Vector operations do not accept dynamic VL parameters.

\[
\%0 = \text{arith.addf } \%v, \%v : \text{vector<8xf32>}
\]

MLIR Limitation

B[0]  B[1]  \cdots  B[?]  B[?]  \\
C[0]  C[1]  \cdots  C[?]  C[?]  \\

Tail Processing

\begin{align*}
\text{Tail} &= \text{getTail}(d) \\
\text{Loop:} & \quad \text{if} \ (\text{not Tail}) \\
& \quad \text{vector load} \\
& \quad \text{vector add} \\
& \quad \text{vector store} \\
& \quad \text{else} \\
& \quad \text{calculate mask} \\
& \quad \text{masked load} \\
& \quad \text{masked add} \\
& \quad \text{masked store} \\
& \quad \text{end if} \\
& \quad \text{End loop}
\end{align*}

Set Dynamic VL

\begin{align*}
\text{AVL} &= d \\
\text{While}(\text{AVL} > 0): \\
& \quad \text{do:} \\
& \quad \quad \text{vl} = \text{setvl AVL, LMUL, SEW} \\
& \quad \quad \text{vector load vl} \\
& \quad \quad \text{vector add vl} \\
& \quad \quad \text{vector store vl} \\
& \quad \quad \text{AVL} = \text{AVL} - \text{vl} \\
& \quad \quad \text{End}
\end{align*}

Ops Accept Dynamic VL

No SETVL Operation Cannot Set Dynamic VL
**Add RVV MLIR Support** (balance the generality and specificity)

1 – RVV-Specific Dialect SetVL Operation: Set dynamic vector length

```mlir
%vl = rvv.setvl %avl, %sew, %lmul : index
```

- **AVL** = Application Vector Length
- **SEW** = Selected Element Width
- **LMUL** = Vector Register Group Multiplier

2 – Generic Vector Predication Operation

```mlir
%vec = vector_exp.predication %mask, %vl : vector<4x1i32>, i32 { %ele = vector.load %m[%c0, %c0]: memref<8x8x1i32>, vector<4x1i32> vector.yield %ele : vector<4x1i32> } : vector<4x1i32>
```

**caret notes:**

- RVV Tail Processing
- Mask-Based Approach
- Strip-Mining Approach
- Get the application vector length (d) at runtime
- Tail = getTail(d)
- Loop:
  - if (not Tail)
  - vector load
  - vector add
  - vector store
  - else
  - calculate mask
  - masked load
  - masked add
  - masked store
  - end if
- End loop

**Set Dynamic VL**

- AVL = d
- While(AVL > 0):
  - do:
    - vl = setvl AVL, LMUL, SEW
    - vector load vl
    - vector add vl
    - vector store vl
    - AVL = AVL - vl
  - end if
- End loop
RISC-V High-Performance Hardware Support - Gemmini

Gemmini Hardware Architecture\(^1\)

![Gemmini Hardware Architecture Diagram]

Gemmini Software Stack

- C++ Operators
- Macro Function
- Inline Assembly
- RISC-V GNU Toolchain
- Gemmini ISA
- Spike Simulator

\(^1\) The origin image is from Gemmini GitHub repository - https://github.com/ucb-bar/gemmini
RISC-V High-Performance Hardware Support - Gemmini

Deep Learning Models

```
linalg.matmul
linalg.conv_2d
... ...
gemmini.tile_matmul
gemmini.tile_conv
... ...
```

Linalg Dialect

```
int_riscv_loop_ws_config_bounds
int_riscv_loop_ws_config_addrs_ab
int_riscv_loop_ws_config_addrs_dc
int_riscv_loop_ws_config_strides_ab
int_riscv_loop_ws_config_strides_dc
... ...
```

Gemmini Dialect

```
gemmini.tile_matmul
gemmini.tile_conv
... ...
```

Gemmini Intrinsic

```
gemmini.tile_matmul
gemmini.tile_conv
... ...
```

Gemmini ISA

```
Gemmini MLIR Intrinsic
Gemmini LLVM Intrinsic
RISC-V GNU Toolchain
```

Buddy Compiler Gemmini Support

```
MLIR Operations
Gemmini MLIR Dialect
Gemmini LLVM Intrinsic
RISC-V GNU Toolchain
```

Gemmini Software Stack

```
C++ Operators
Macro Function
Inline Assembly
RISC-V GNU Toolchain
```

Spike Simulator

```
Gemmini ISA
```

Buddy Compiler
### Multimodal Representations

**Buddy Compiler Domain-Specific Dialects**

- Vector Representation
  - Vector Dialect
  - RVV Dialect
  - LLVM VP Intrinsic

- **MLIR Core Dialects**
  - MemRef Dialect
  - Affine Dialect
  - SCF Dialect

- **Gemmini Dialects**
  - Gemmini Operation
  - Gemmini Intrinsic Operation
  - Custom LLVM Extension

### Deep Learning Model Representations

**MLIR TOSA / Linalg Dialect**

### LLVM | RISC-V GNU Toolchain | Emulators

- SIMD Processor (RISC-V P Extension)
- **Vector Processor** (RISC-V V Extension)
- GPGPU (e.g. Ventus)
- DSA (e.g. Gemmini)

### Preprocessing + Deep Learning Workload

- Preprocessing Operation Optimization
- Unified Data Structure to Avoid Copy Overhead
- Potential Operation Fusion Opportunity

### Compiler Passes + Hardware Architecture

- Design Representations for Hardware Features
- Configure Passes by Hardware Information
- Potential Auto-Tuning / DSE Opportunity
Thanks

hongbin2019@iscas.ac.cn