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

Buddy Compiler Introduction

Buddy Compiler

Buddy Compiler is a domain-specific compiler framework.

We are building a co-design ecosystem based on MLIR and RISC-V.

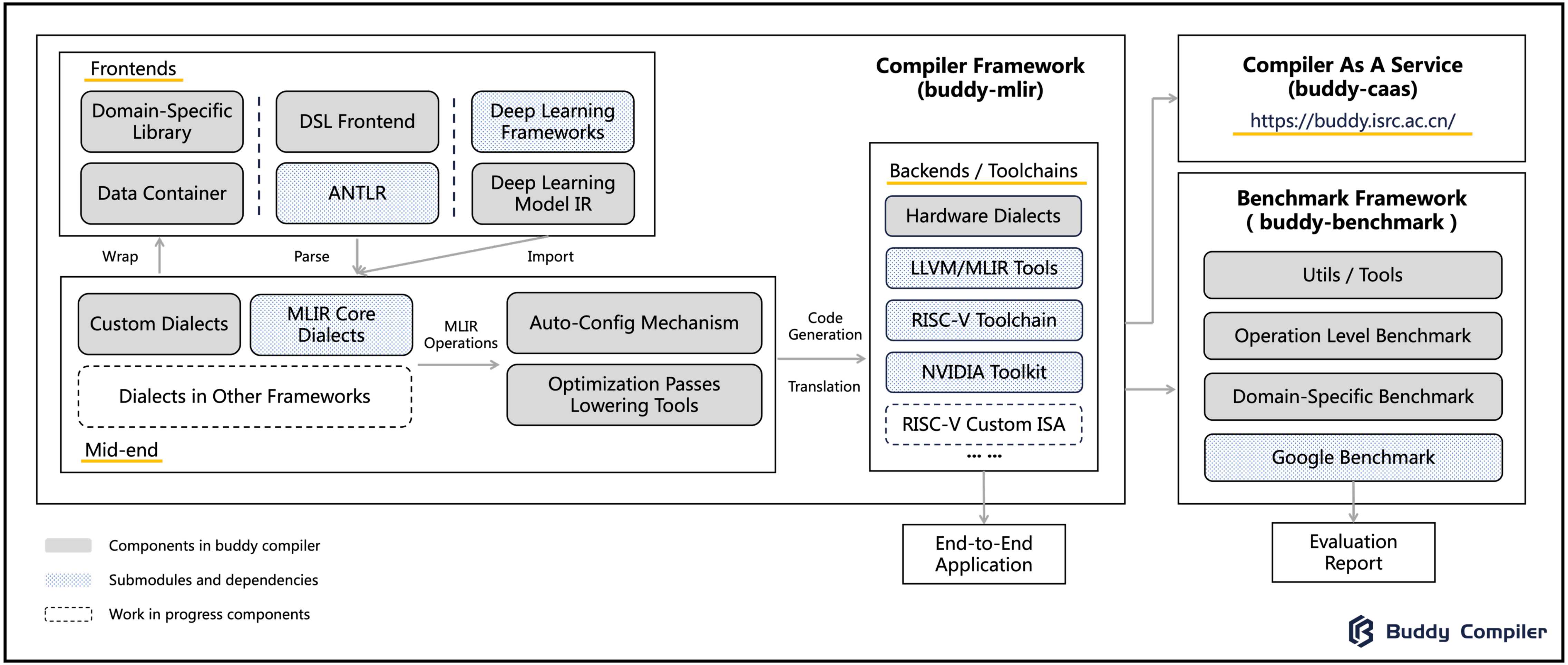
We hope to achieve deep co-design from DSL to DSA.

Deep co-design for deep learning!

Buddy Compiler Overview

Buddy Compiler

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



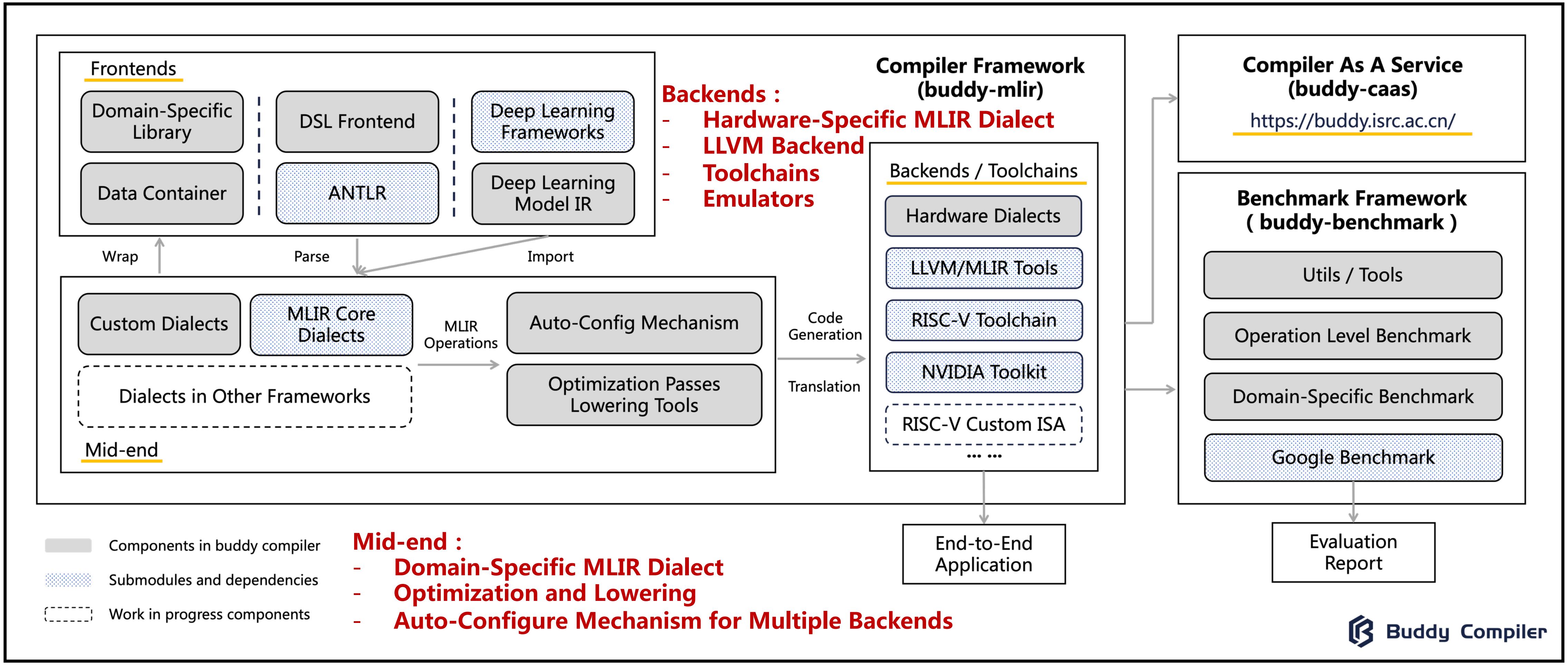
Homepage: https://buddy-compiler.github.io/
GitHub: https://github.com/buddy-compiler

Buddy Compiler Overview

Buddy Compiler

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**

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

Buddy Compiler

OpenCV | Buddy Compiler Libraries or DSL | TensorFlow | PyTorch | OpenXLA

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

Domain-Specific Application and Language Support

“

The key to co-design is unified abstraction and presentation.

MLIR can unify domain-specific applications and languages together.

”

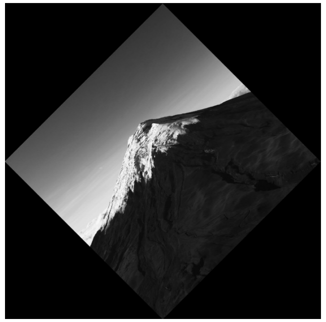
Domain-Specific Application Support

Buddy Compiler

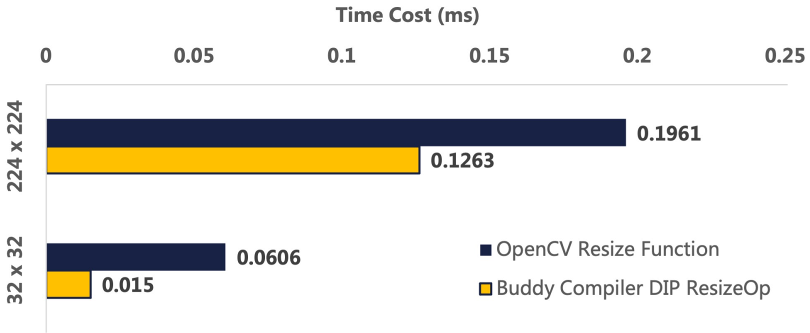
Image Processing

Original Image [1]

Correlation

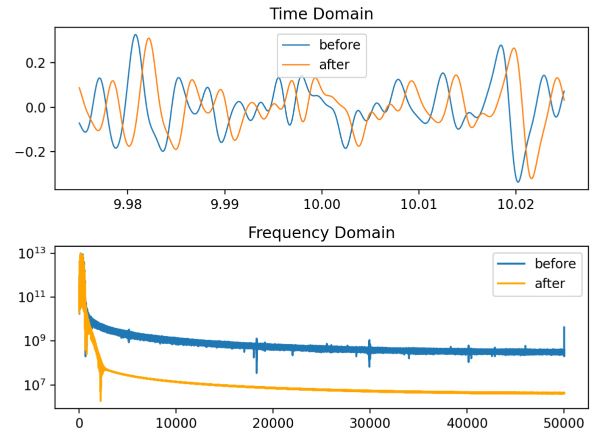


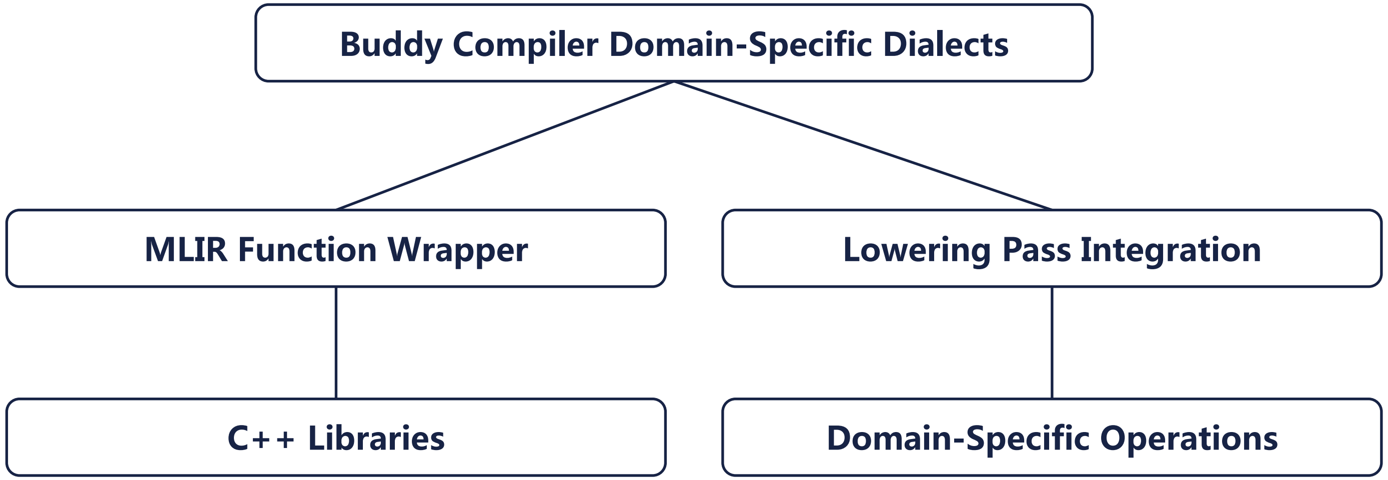
Rotation



Resize Performance Evaluation

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
[2] The origin audio is from NASA 's recording of sound on Mars - https://www.nasa.gov/connect/sounds/index.html

Domain-Specific Language Support

Buddy Compiler

```
def main() {
  var a<2, 2> = [1, 2, 3, 4, 5, 6];
  var b<2, 2> = [[1, 2], [3, 4]];
  print(a + b);
}
```

DSL Source Code

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

```
2.000000 4.000000
6.000000 8.000000
```

```
expression :
  ... ...
  | expression Add expression
  ... ...
  ;
```

ANTLR Syntax Definition (g4)

```
def AddOp : Toy_Op<"add",
    [Pure, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  ... ...

  let arguments = (ins F64Tensor:$lhs, F64Tensor:$rhs);
  let results = (outs F64Tensor);
  ... ...
}
```

MLIR Operation Definition

```
virtual std::any visitExpression(ToyParser::ExpressionContext *ctx) override {
  mlir::Value value;
  ... ...
  else if (ctx->Add() || ctx->Mul()) {
    mlir::Value lhs = std::any_cast<mlir::Value>(visit(ctx->expression(0)));
    mlir::Value rhs = std::any_cast<mlir::Value>(visit(ctx->expression(1)));
    mlir::Location loaction =
        loc(ctx->start->getLine(), ctx->start->getCharPositionInLine());
    if (ctx->Add())
      value = builder.create<mlir::toy::AddOp>(loaction, lhs, rhs);
    ... ...
  }
}
```

ANTLR Visitor

```
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], [3.0, 4.0]]> : tensor<2x2xf64>
    %3 = toy.reshape(%2 : tensor<2x2xf64>) to tensor<2x2xf64>
    %4 = toy.add %1, %3 : (tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<*xf64>
    toy.print %4 : tensor<*xf64>
    toy.return
  }
}
```

Generated MLIR

Optimization for Multiple Backends

Buddy Compiler

“

IRs are unified, and optimization should not be fragmented.

No one wants to port an algorithm to every platform!

”

Optimization for Multiple Backends

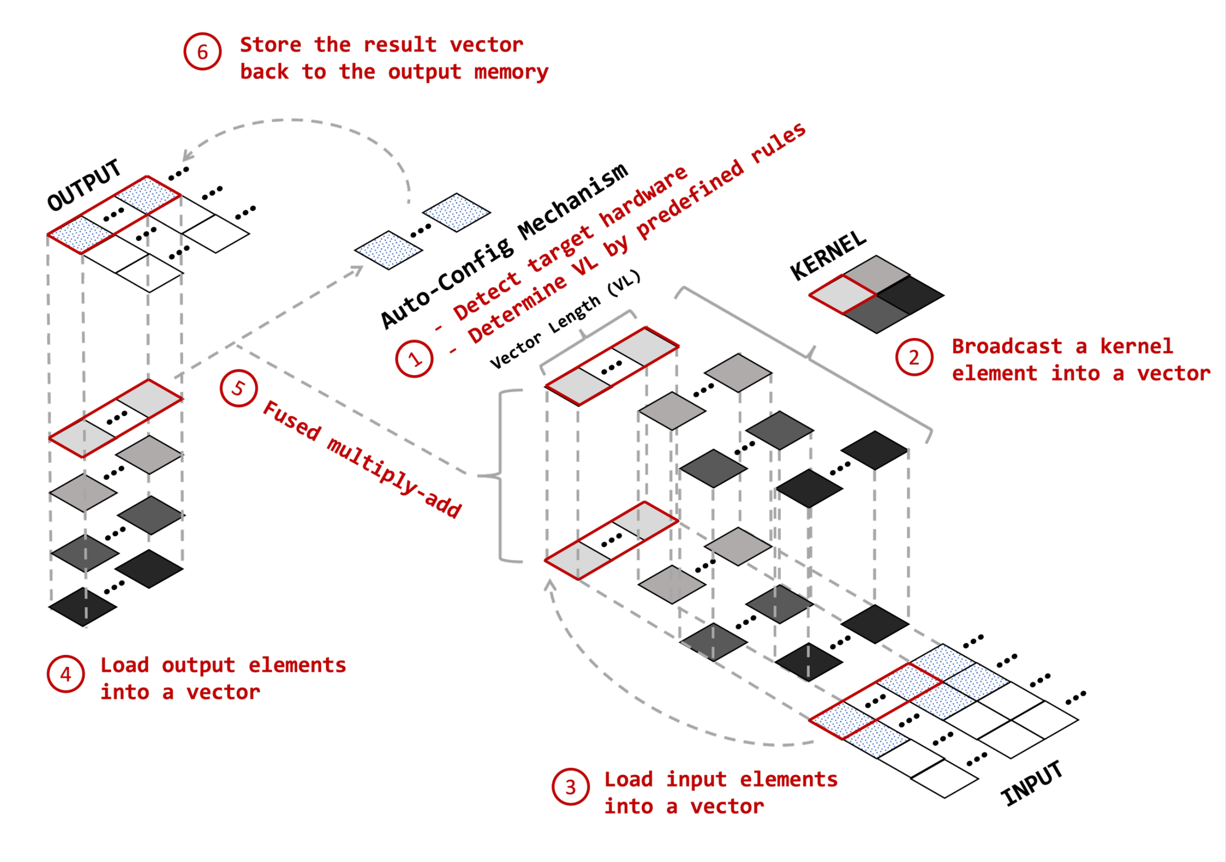
Buddy Compiler

1. Combination of Multiple Optimization Strategy

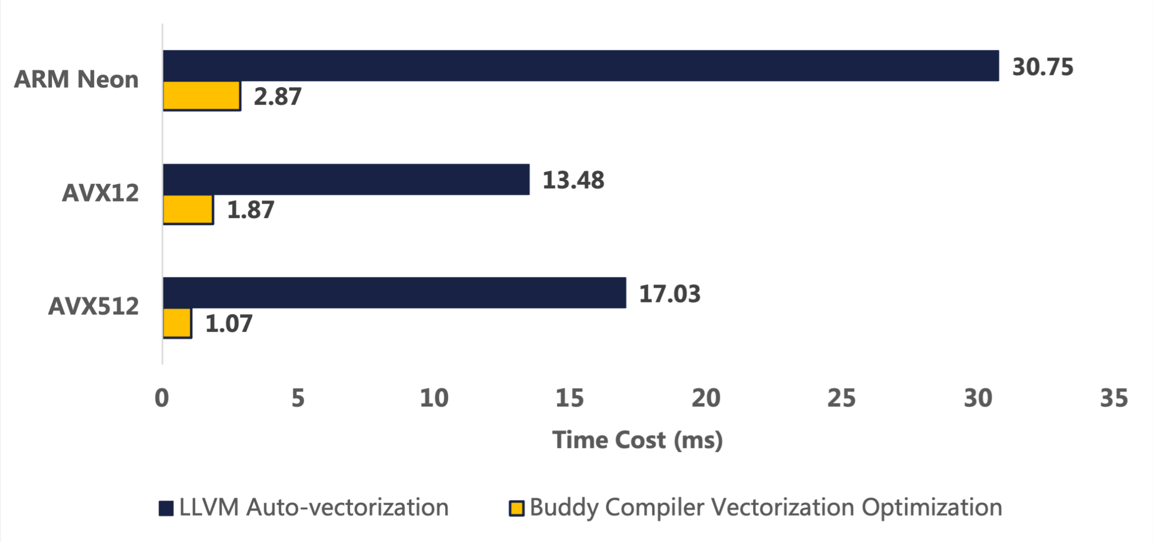
- High-Level Optimization Algorithm
- Compilation Optimization

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 x1024 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,

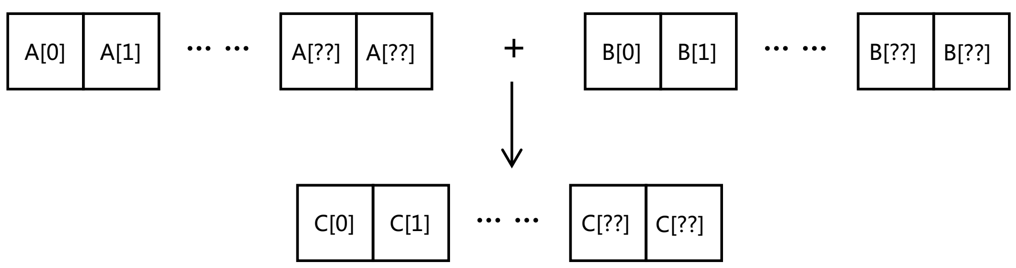
let's add a new IR abstraction.

"

RISC-V High-Performance Hardware Support - RVV

Buddy Compiler

RVV Tail Processing



Get the application vector length (d) at runtime

Mask-Based Approach

```
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
```

Strip-Mining Approach

```
AVL = d
While(AVL > 0):
  do:
    vl = setvl AVL , LMUL , SEW
    vector load vl
    vector add vl
    vector store vl
    AVL = AVL - vl
End
```

Set Dynamic VL ← ~~vl = setvl AVL , LMUL , SEW~~

Ops Accept Dynamic VL { ~~vector load vl~~, ~~vector add vl~~, ~~vector store vl~~

MLIR Limitation

Information Required at Compile Time :

- Dynamic VL Configuration
  - AVL Configuration
  - LMUL Configuration
  - SEW Configuration
- Operations Dynamic VL Operand

No SETVL Operation Cannot Set Dynamic VL

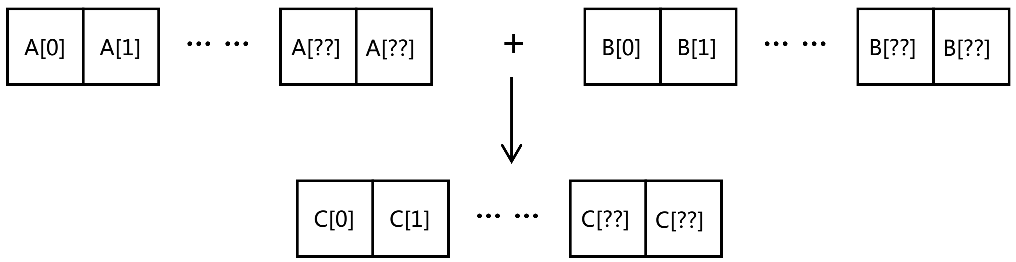
Vector operations do not accept dynamic VL parameters.

```
%0 = arith.addf %v, %v : vector<8xf32>
```

RISC-V High-Performance Hardware Support - RVV

Buddy Compiler

RVV Tail Processing



Get the application vector length (d) at runtime

Mask-Based Approach

```
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
```

Strip-Mining Approach

```
AVL = d
While(AVL > 0):
  do:
    vl = setvl AVL , LMUL , SEW      <- Set Dynamic VL   (struck through)
    vector load vl                   } Ops Accept Dynamic VL   (struck through)
    vector add vl                    }   (struck through)
    vector store vl                  }   (struck through)
    AVL = AVL - vl
End
```

MLIR Limitation

Add RVV MLIR Support (balance the generality and specificity)

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

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

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

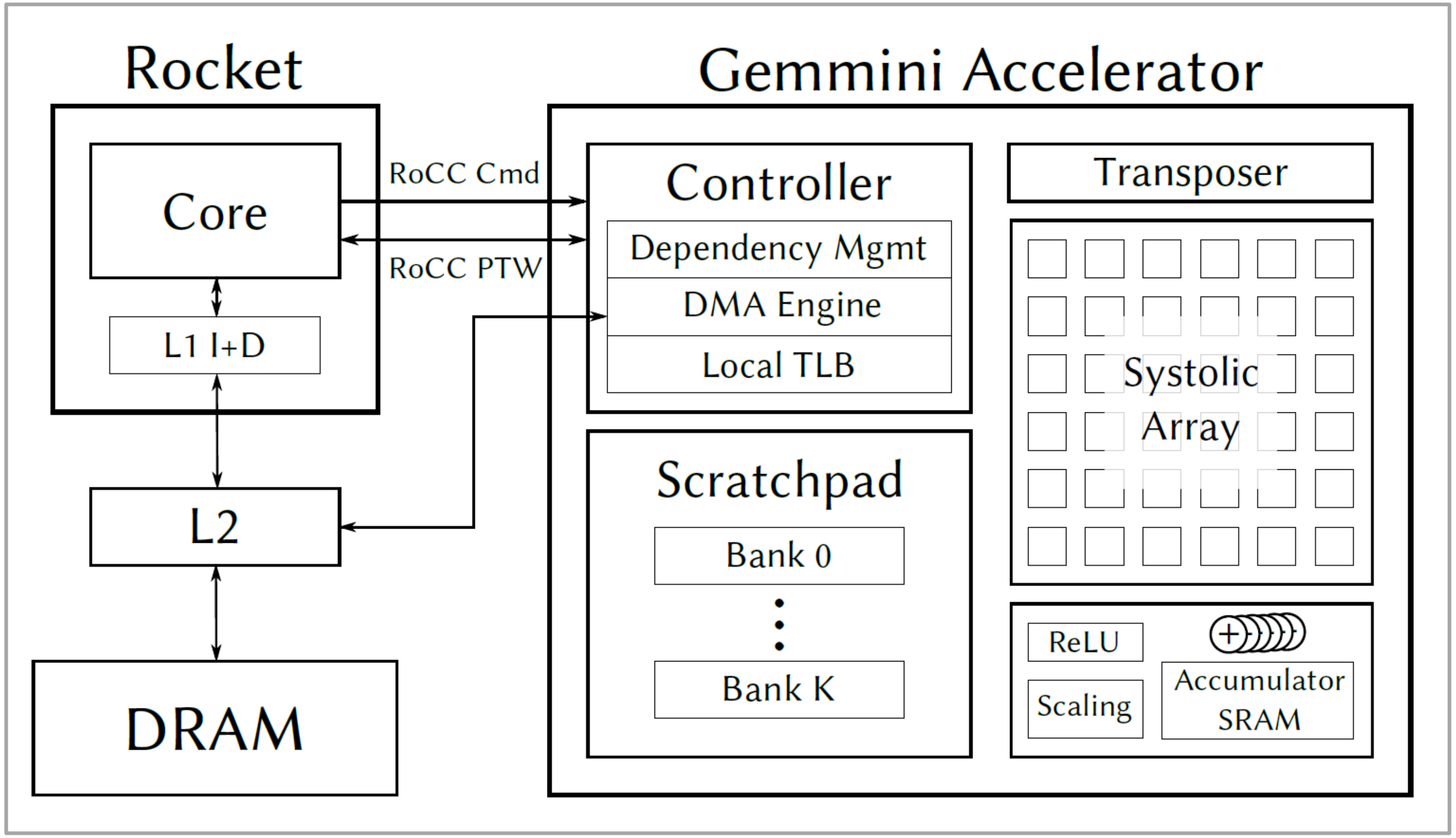
2 – Generic Vector Predication Operation

```
%vec = vector_exp.predication %mask, %vl : vector<[4]xi1>, i32 {
  %ele = vector.load %m[%c0, %c0]: memref<8x8xi32>, vector<[4]xi32>
  vector.yield %ele : vector<[4]xi32>
} : vector<[4]xi32>
```

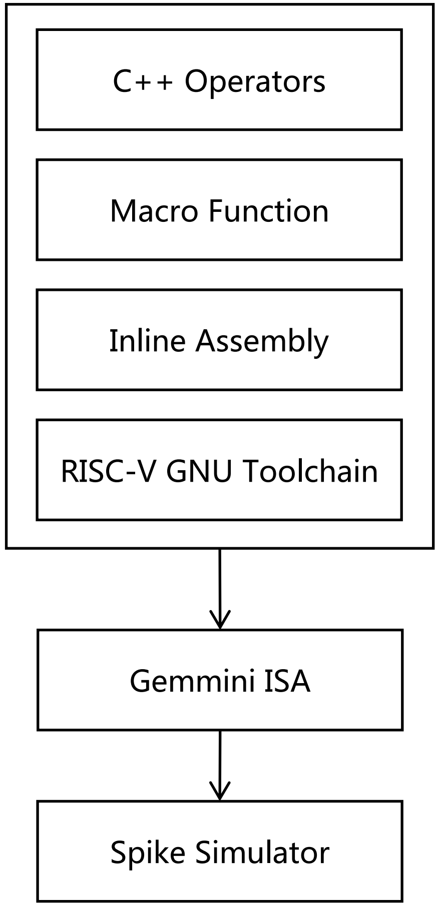
RISC-V High-Performance Hardware Support - Gemmini

Buddy Compiler

Gemmini Hardware Architecture[1]

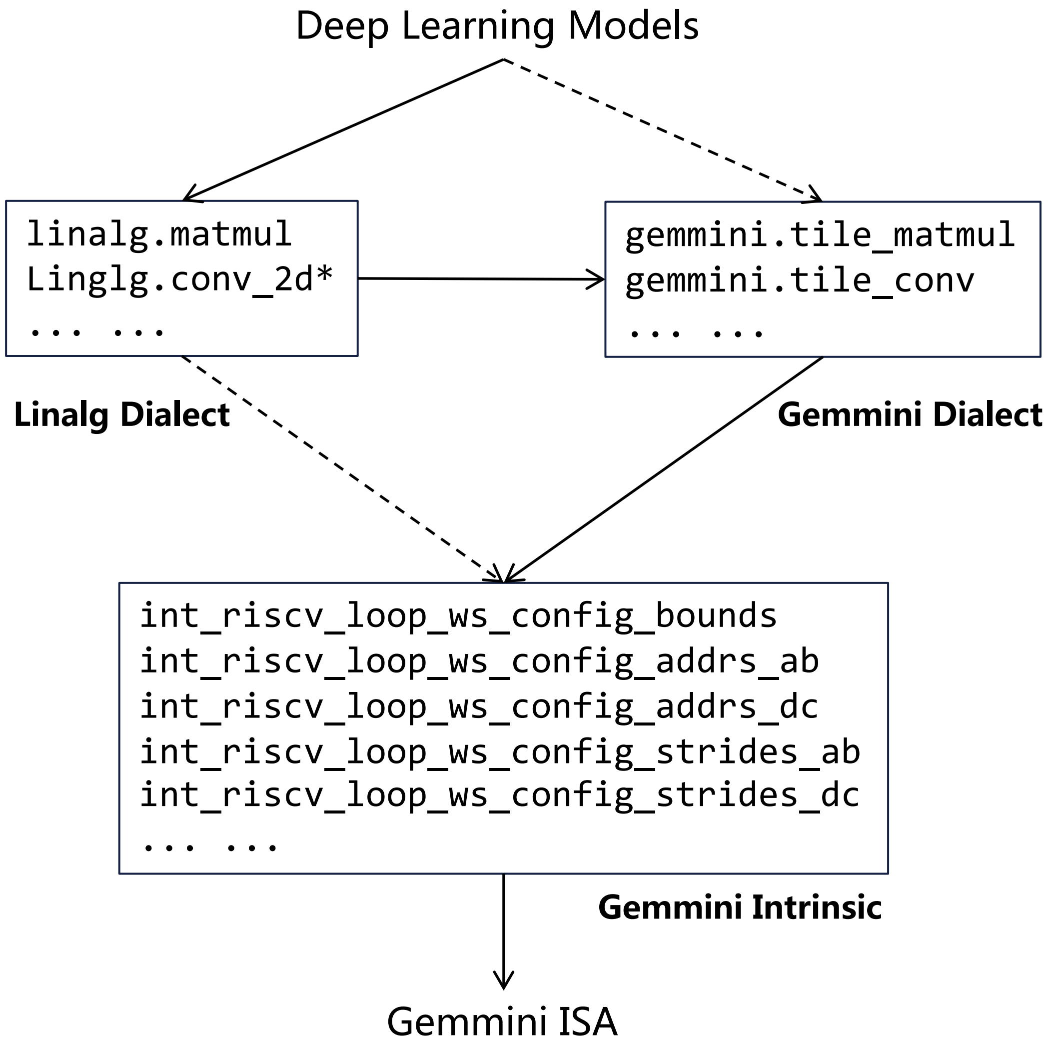


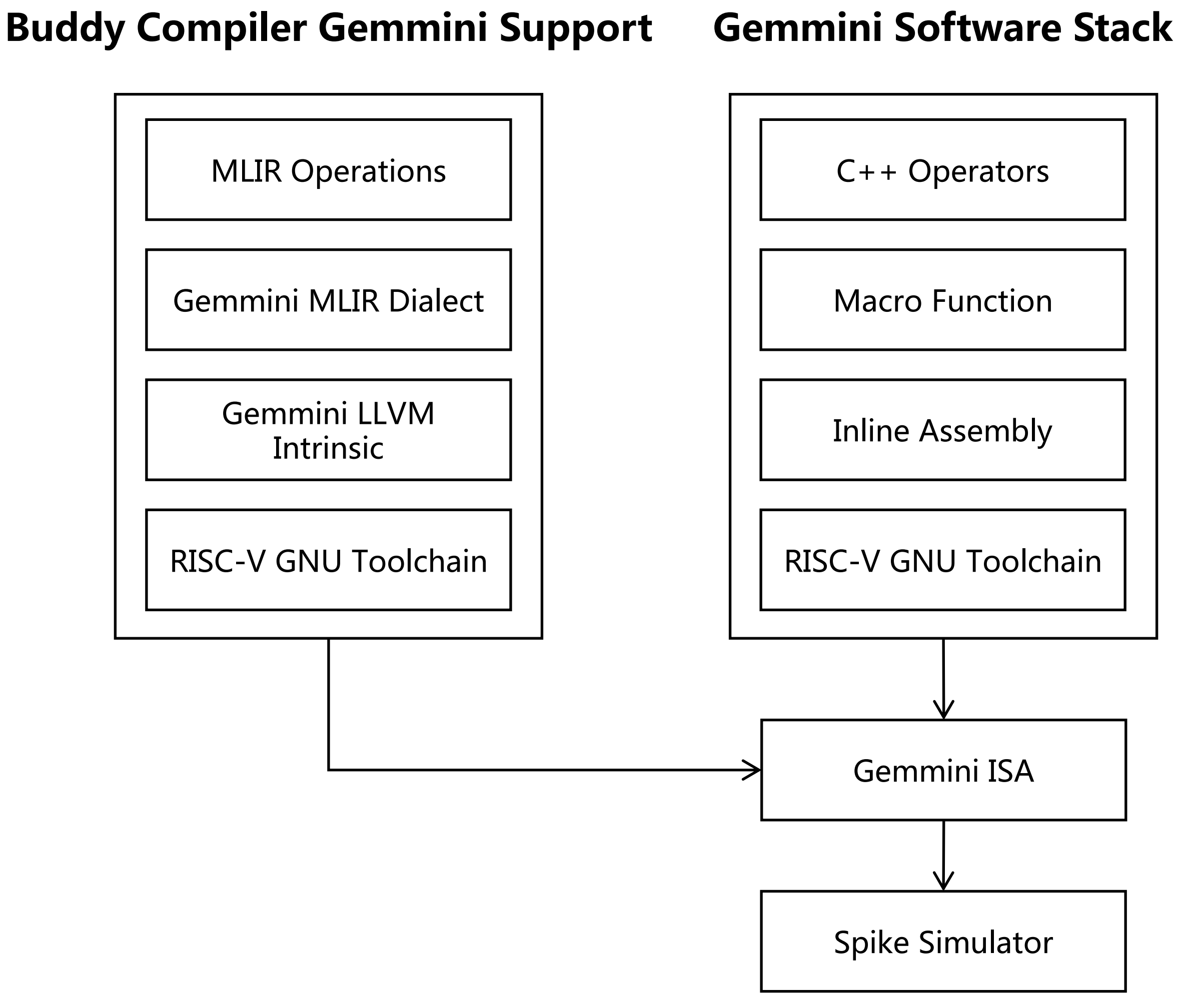
Gemmini Software Stack



[1] The origin image is from Gemmini GitHub repository - https://github.com/ucb-bar/gemmini

RISC-V High-Performance Hardware Support - Gemmini





What's Next ...

Buddy Compiler

OpenCV | Buddy Compiler Libraries or DSL

Multimodal Representations

Buddy Compiler Domain-Specific Dialects

TensorFlow | PyTorch | OpenXLA

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

Buddy Compiler

Thanks

hongbin2019@iscas.ac.cn