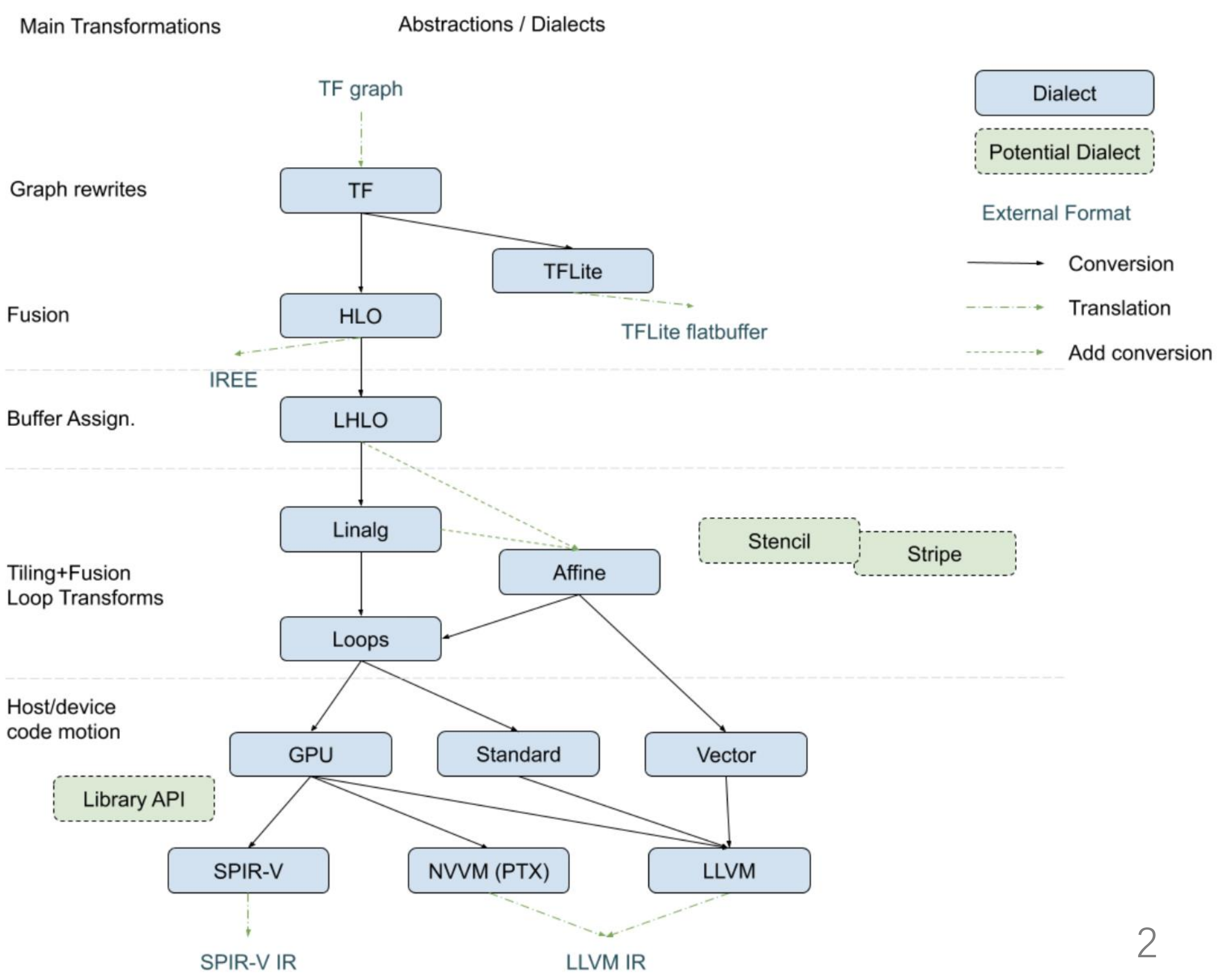
中国科学院软件研究所
Institute of Software Chinese Academy of Sciences

智能软件研究中心
Intelligent Software Research Center

MLIR 向量支持的部分概述

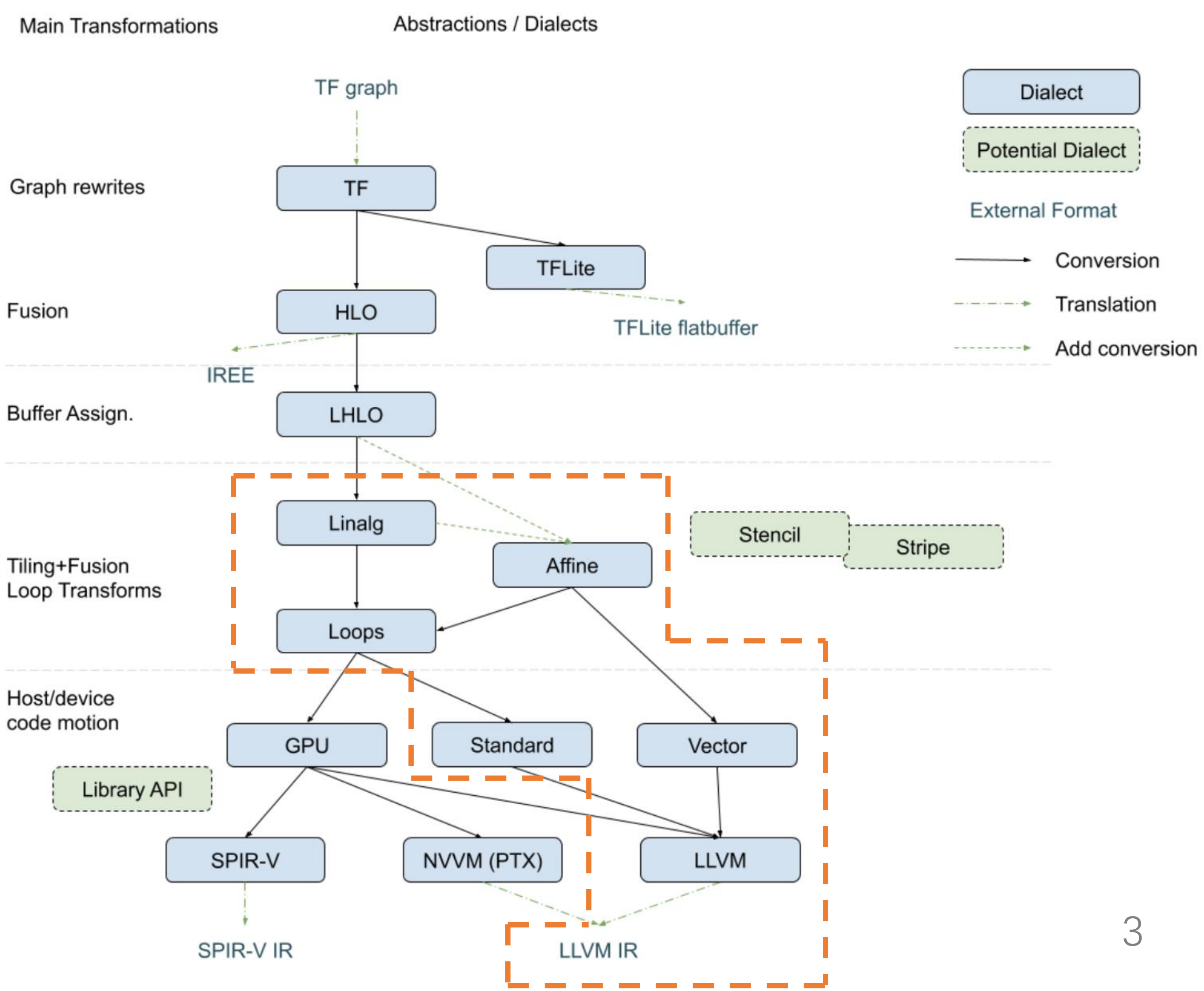
PLCT实验室
张洪滨

MLIR 层次结构



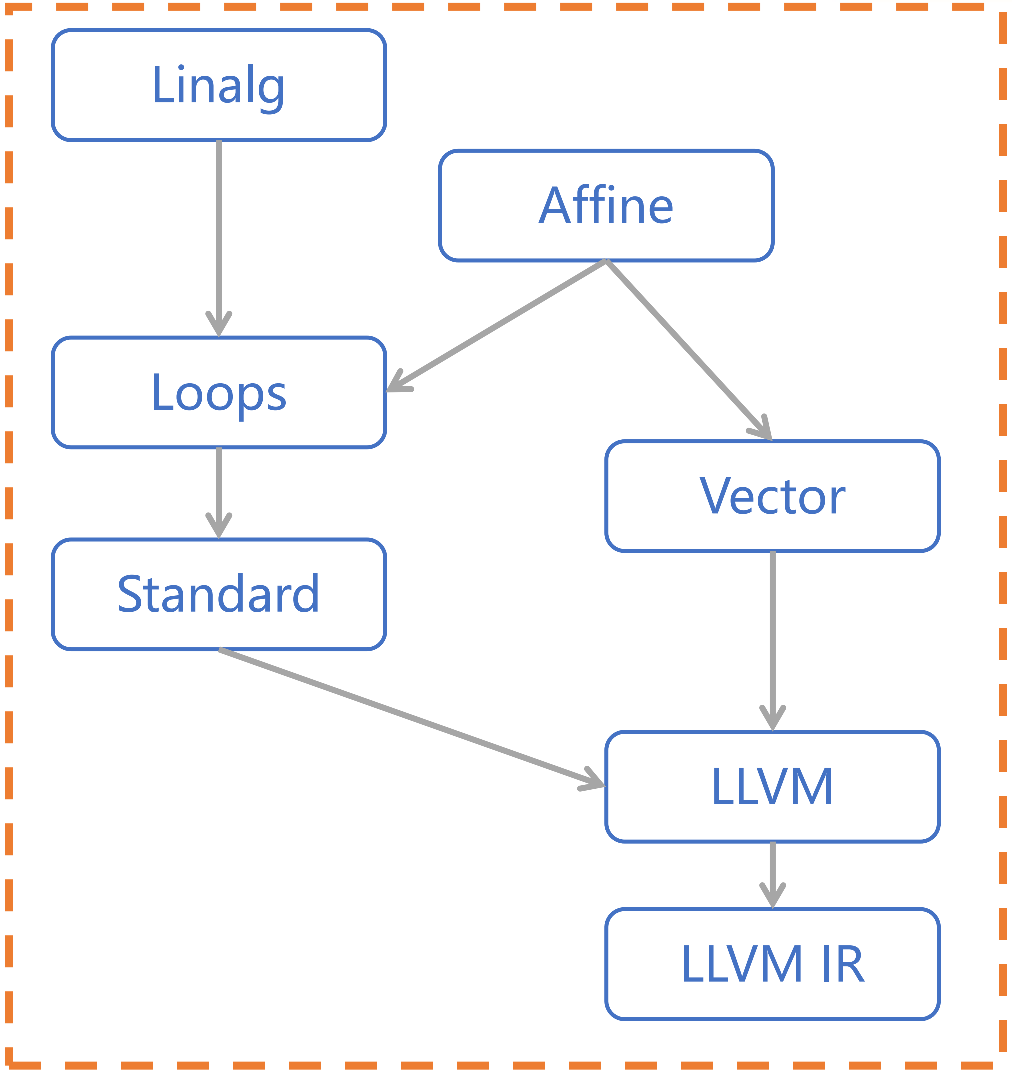
图片来自：https://mlir.llvm.org/docs/Dialects/Vector/

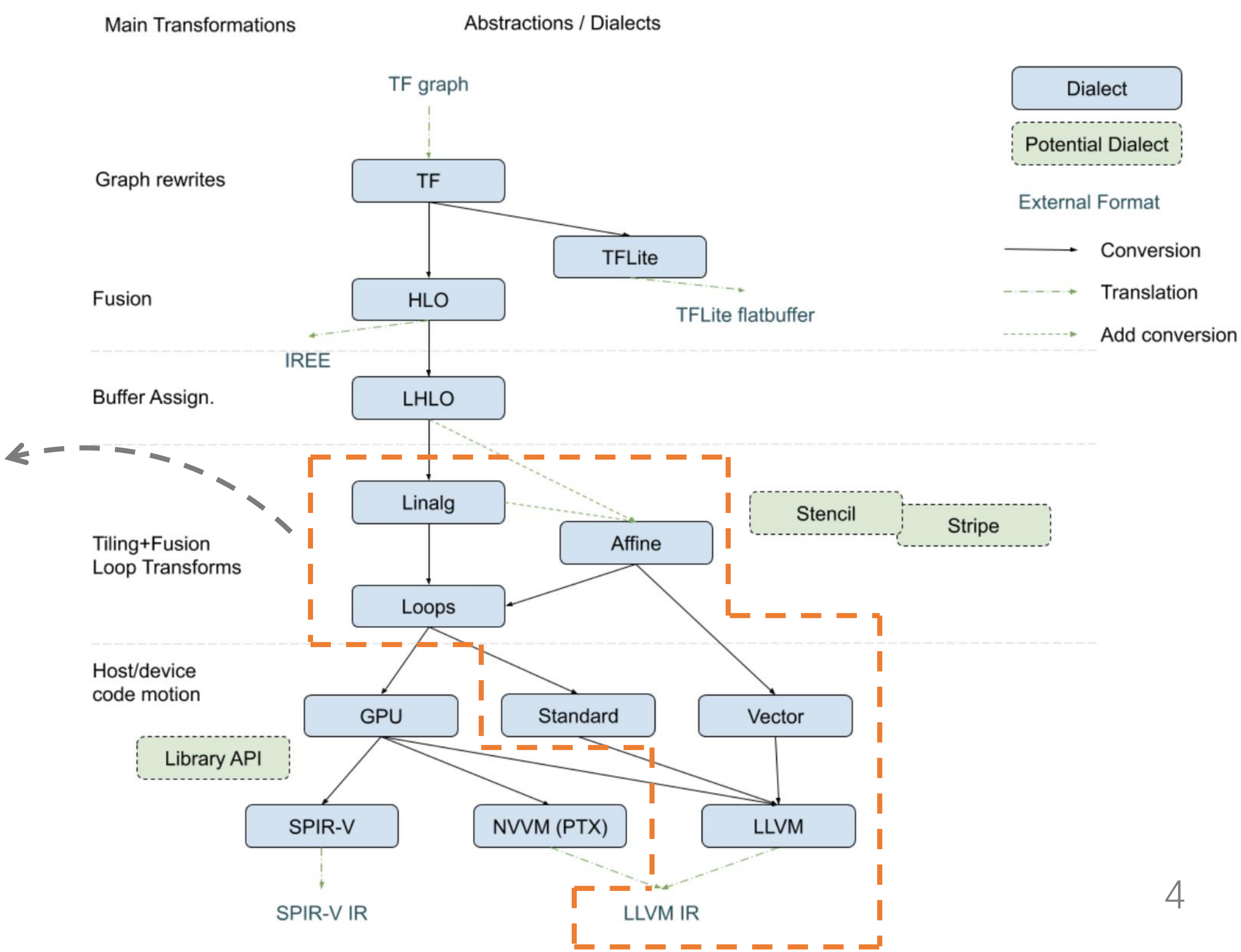
MLIR 层次结构



图片来自：https://mlir.llvm.org/docs/Dialects/Vector/

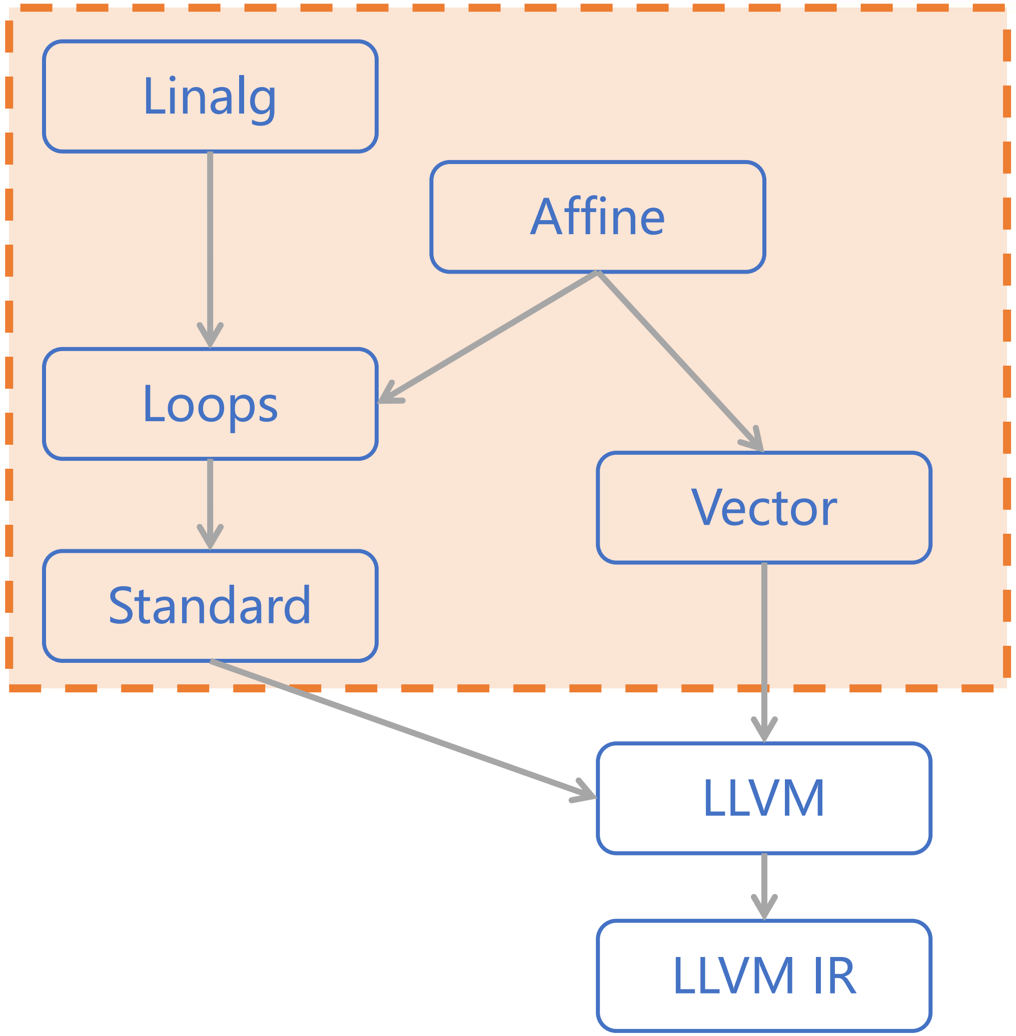
MLIR 层次结构





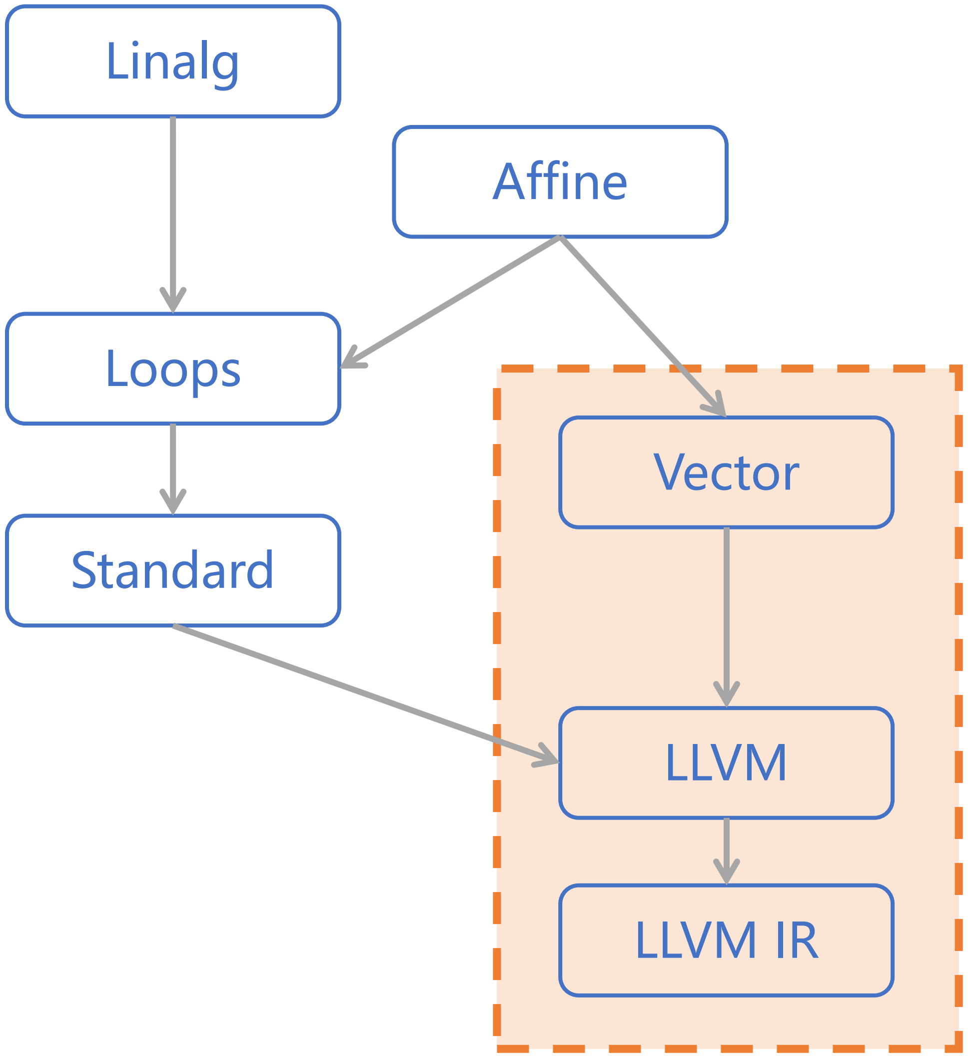
图片来自：https://mlir.llvm.org/docs/Dialects/Vector/

MLIR 层次结构



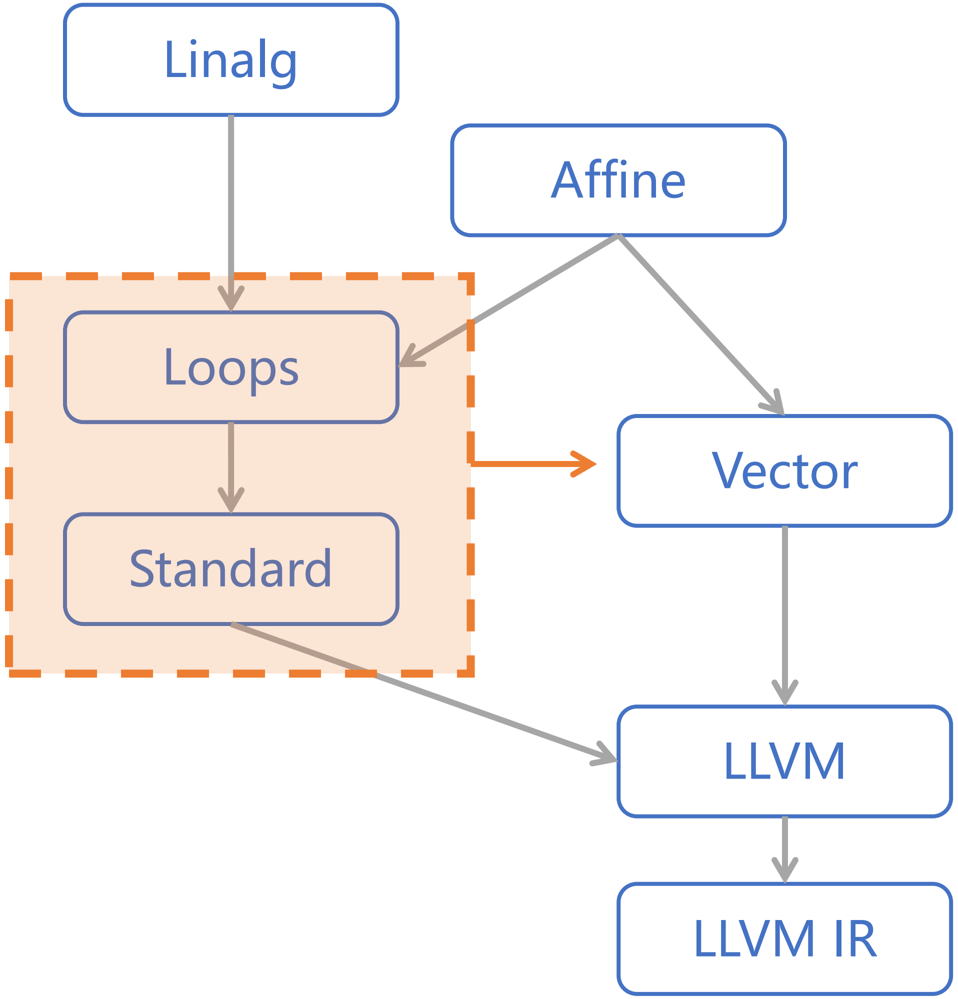
1. 高层 IR 向量化
2. MLIR 使用 Intrinsic

MLIR 层次结构



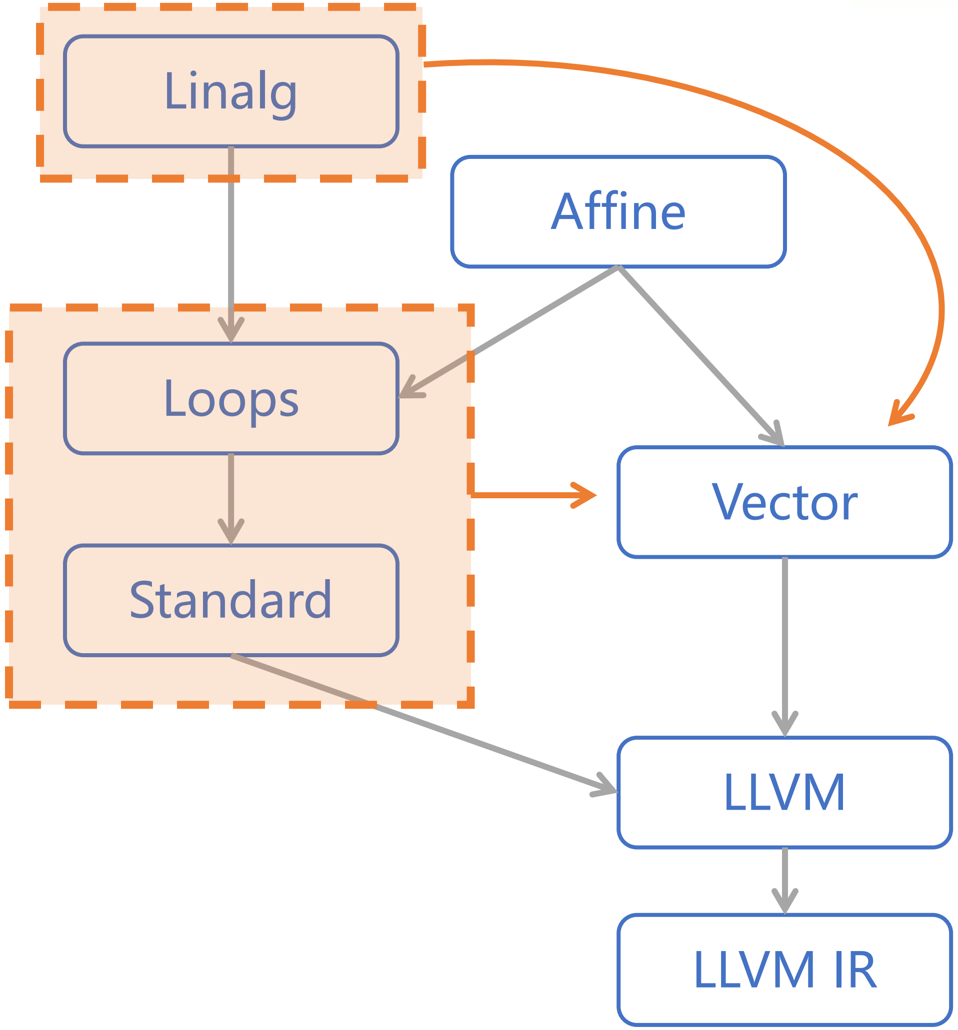
1. 高层 IR 向量化
2. MLIR 使用 Intrinsic

高层 IR 自动向量化



① 将标量的循环进行向量化

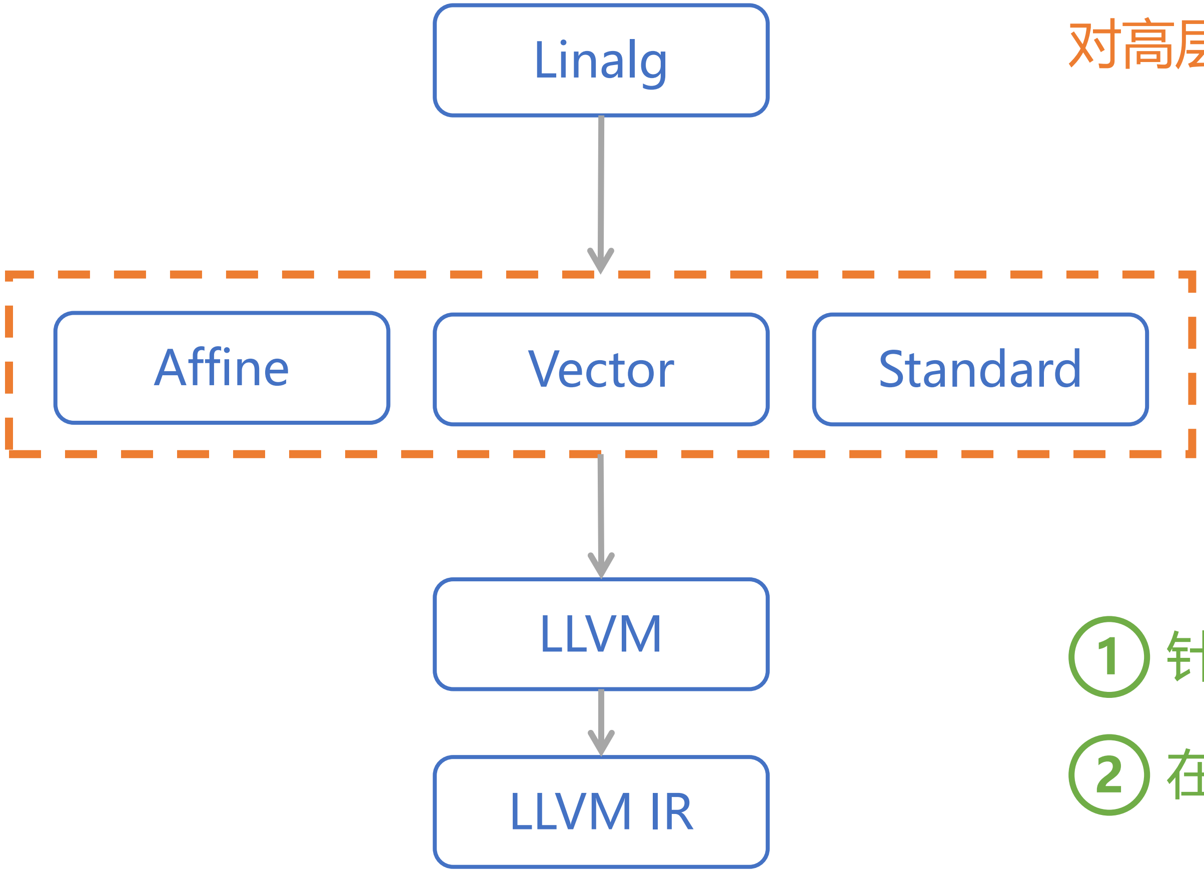
高层 IR 自动向量化



② 对高层 Ops 使用向量化 Pass 进行重写

① 将标量的循环进行向量化

高层 IR 自动向量化



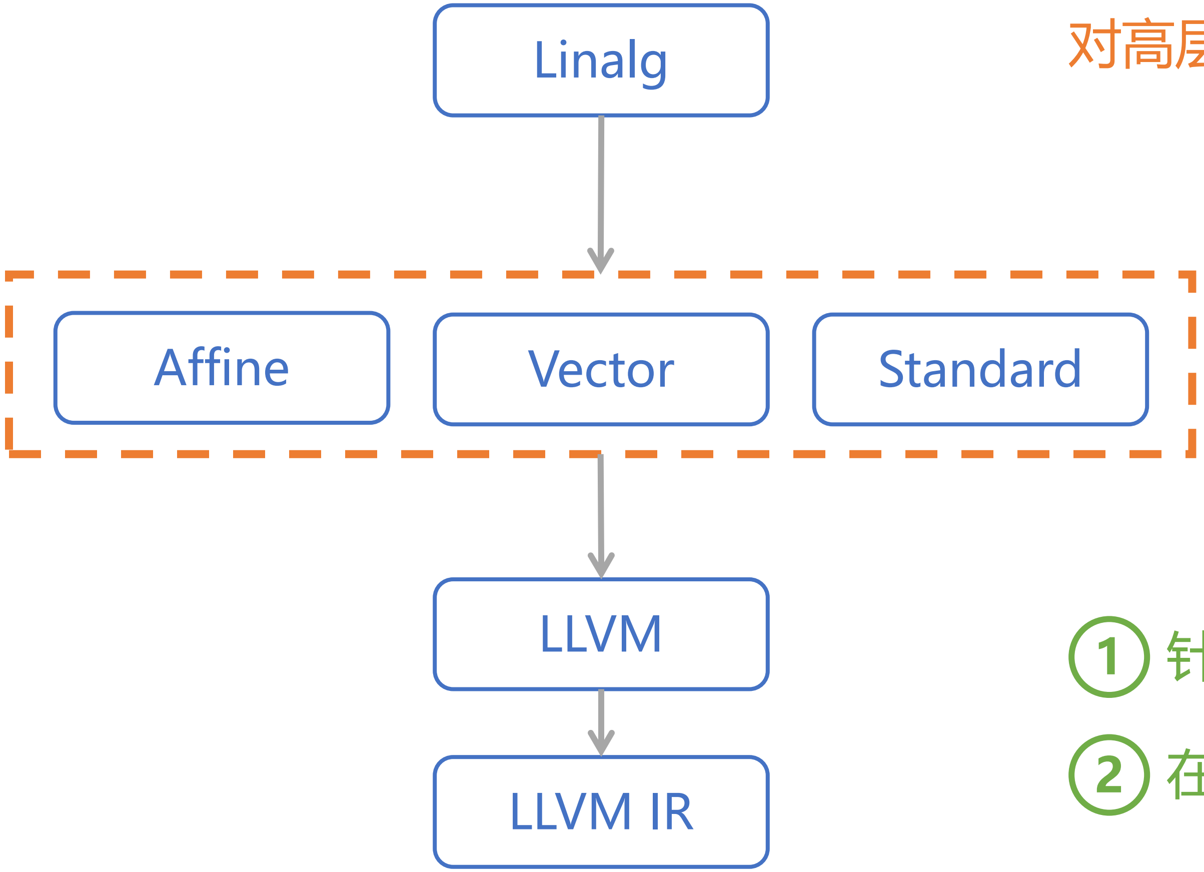
对高层 Ops 使用向量化 Pass 进行重写

使用多个 Dialects 实现向量化 Pass

1 针对性设计的向量化 Pass 效果更好

2 在高层 IR 上实现向量化算法可重用性好

高层 IR 自动向量化



对高层 Ops 使用向量化 Pass 进行重写

使用多个 Dialects 实现向量化 Pass

1. 针对性设计的向量化 Pass 效果更好
2. 在高层 IR 上实现向量化算法可重用性好

卷积向量化工具 conv-opt

buddy-mlir：https://github.com/buddy-compiler/buddy-mlir

高层 IR 自动向量化

玉兔月球车边缘检测（Sobel Kernel)

原始图片	3x3 Kernel	5x5 Kernel	7x7 Kernel	9x9 Kernel
1024 x 1024	OpenCV: 0.005236 s	OpenCV: 0.011715 s	OpenCV: 0.023568 s	OpenCV: 0.033107 s
	conv-opt: 0.005106 s	**conv-opt: 0.006815 s**	**conv-opt: 0.008309 s**	**conv-opt: 0.012519 s**

卷积向量化工具 conv-opt

buddy-mlir：https://github.com/buddy-compiler/buddy-mlir

MLIR 使用 Intrinsic

以 X86 为例

```
$ <mlir-opt> <input file> -convert-vector-to-llvm="enable-x86vector"
```

MLIR 使用 Intrinsic

以 X86 为例

```
$ <mlir-opt> <input file> -convert-vector-to-llvm=“enable-x86vector”
```

```
// llvm-project/mlir/include/mlir/Conversion/Passes.td

def ConvertVectorToLLVM : Pass<"convert-vector-to-llvm", "ModuleOp"> {
  ... ...
  let options = [
    ... ...
    Option<"enableX86Vector", "enable-x86vector",
           "bool", /*default=*/"false",
           "Enables the use of X86Vector dialect while lowering the vector dialect” >
}
```

MLIR 使用 Intrinsic

以 X86 为例

```
$ <mlir-opt> <input file> -convert-vector-to-llvm="enable-x86vector"
```

```
// llvm-project/mlir/include/mlir/Conversion/Passes.td

def ConvertVectorToLLVM : Pass<"convert-vector-to-llvm", "ModuleOp"> {
  ... ...
  let options = [
    ... ...
    Option<"enableX86Vector", "enable-x86vector",
           "bool", /*default=*/"false",
           "Enables the use of X86Vector dialect while lowering the vector dialect” >
}
```

```
// llvm-project/mlir/lib/Conversion/VectorToLLVM/ConvertVectorToLLVMPass.cpp

if (enableX86Vector) {
  configureX86VectorLegalizeForExportTarget(target);
  populateX86VectorLegalizeForLLVMExportPatterns(converter, patterns);
}
```

配置合法/非法 Operation

执行 Operation 变换

MLIR 使用 Intrinsic

以 X86 为例

x86vector

x86vector Dialect 定义

X86 Intrinsic

X86 Intinsic 定义

MLIR 使用 Intrinsic

以 X86 为例

x86vector

x86vector Dialect 定义

Vector Ops

抽象层次高，语义更清晰

Intrinsic Ops

抽象层次低，语义贴近Intrinsic

X86 Intrinsic

X86 Intinsic 定义

MLIR 使用 Intrinsic

以 X86 为例

Vector Ops

抽象层次高，语义更清晰

x86vector

使用 mlir-opt 工具，开启 enable-x86vector 选项

x86vector Dialect 定义

Intrinsic Ops

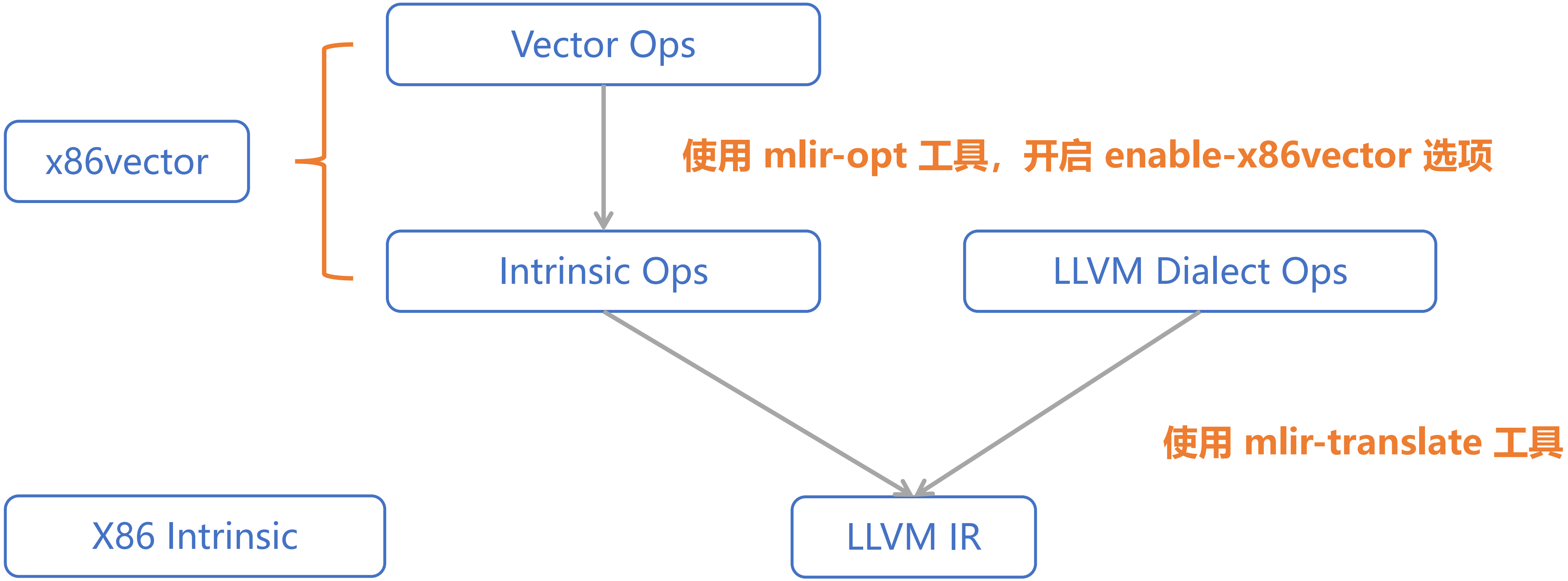
抽象层次低，语义贴近Intrinsic

X86 Intrinsic

X86 Intinsic 定义

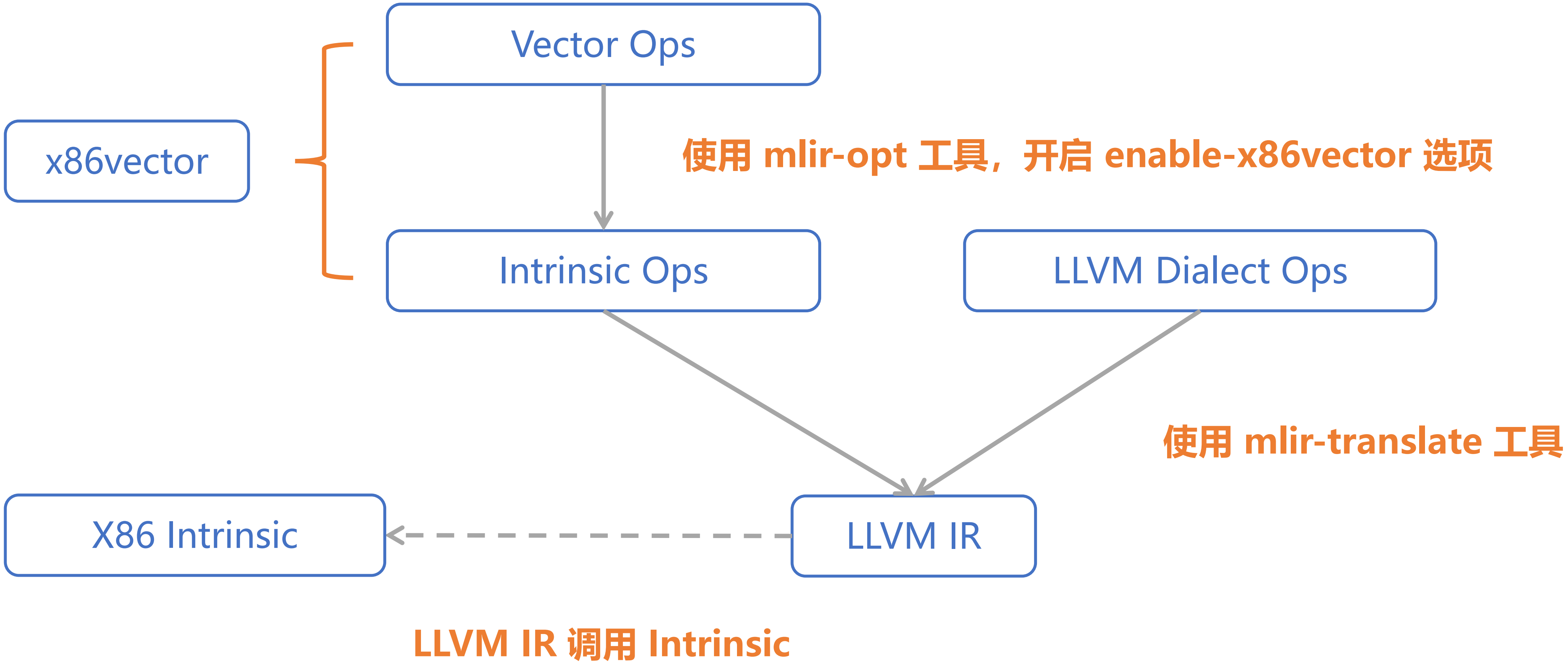
MLIR 使用 Intrinsic

以 X86 为例



MLIR 使用 Intrinsic

以 X86 为例



MLIR 使用 Intrinsic

中国科学院软件研究所
Institute of Software Chinese Academy of Sciences
智能软件研究中心
Intelligent Software Research Center

以 X86 Dot 为例

```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```

使用 mlir-opt 工具，开启 enable-x86vector 选项

```
func @main() {
  %cst = constant dense<[... ...]> : vector<8xf32>
  %cst_0 = constant dense<[... ...]> : vector<8xf32>
  %0 = llvm.mlir.constant(-1 : i8) : i8
  %1 = "x86vector.avx.intr.dp.ps.256"(%cst, %cst_0, %0) : (vector<8xf32>, vector<8xf32>, i8)
                                                          -> vector<8xf32>
  ... ...
}
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

```
func @main() {
  %cst = constant dense<[... ...]> : vector<8xf32>
  %cst_0 = constant dense<[... ...]> : vector<8xf32>
  %0 = llvm.mlir.constant(-1 : i8) : i8
  %1 = "x86vector.avx.intr.dp.ps.256"(%cst, %cst_0, %0) : (vector<8xf32>, vector<8xf32>, i8)
                                                          -> vector<8xf32>
  ... ...
}
```

Operation 和 Intrinsic 参数和返回值为一对一映射关系

```
// llvm-project/llvm/include/llvm/IR/IntrinsicsX86.td
// Vector dot product
let TargetPrefix = "x86" in { // All intrinsics start with "llvm.x86.".
def int_x86_avx_dp_ps_256 : GCCBuiltin<"__builtin_ia32_dpps256">,
        Intrinsic<[llvm_v8f32_ty], [llvm_v8f32_ty, llvm_v8f32_ty, llvm_i8_ty], ... ...>;
}
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

```
func @main() {
  %cst = constant dense<[... ...]> : vector<8xf32>
  %cst_0 = constant dense<[... ...]> : vector<8xf32>
  %0 = llvm.mlir.constant(-1 : i8) : i8
  %1 = "x86vector.avx.intr.dp.ps.256"(%cst, %cst_0, %0) : (vector<8xf32>, vector<8xf32>, i8)
                                                          -> vector<8xf32>
  ... ...
}
```

使用 mlir-translate 工具

```
define void @main() {
  %1 = call <8 x float> @llvm.x86.avx.dp.ps.256(<8 x float> <... ...>, <8 x float> <... ...>, i8 -1)
  ... ...
}
```

MLIR 使用 Intrinsic

中国科学院软件研究所
Institute of Software Chinese Academy of Sciences
智能软件研究中心
Intelligent Software Research Center

以 X86 Dot 为例

```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```

使用 mlir-cpu-runner 工具

```
( 110, 110, 110, 110, 382, 382, 382, 382 )
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```

```
( 1, 2, 3, 4, 5, 6, 7, 8 )
( 9, 10, 11, 12, 13, 14, 15, 16 )
```

使用 mlir-cpu-runner 工具

```
( 110, 110, 110, 110, 382, 382, 382, 382 )
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```

使用 mlir-cpu-runner 工具

```
( 1, 2, 3, 4, 5, 6, 7, 8 )
( 9, 10, 11, 12, 13, 14, 15, 16 )
```

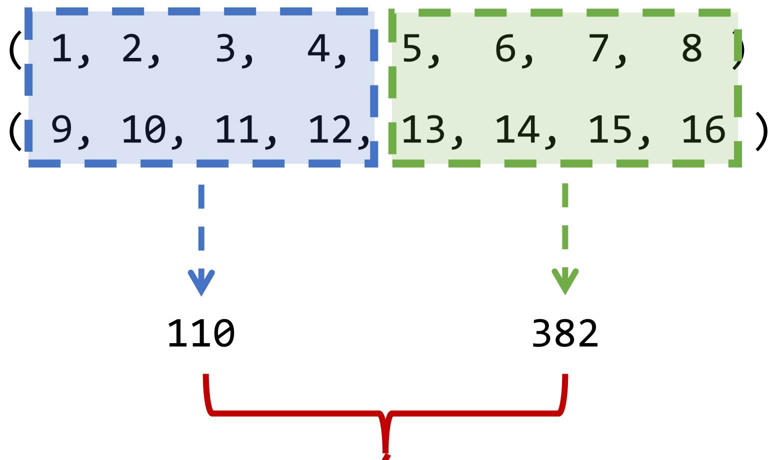
110 382

```
( 110, 110, 110, 110, 382, 382, 382, 382 )
```

MLIR 使用 Intrinsic

以 X86 Dot 为例

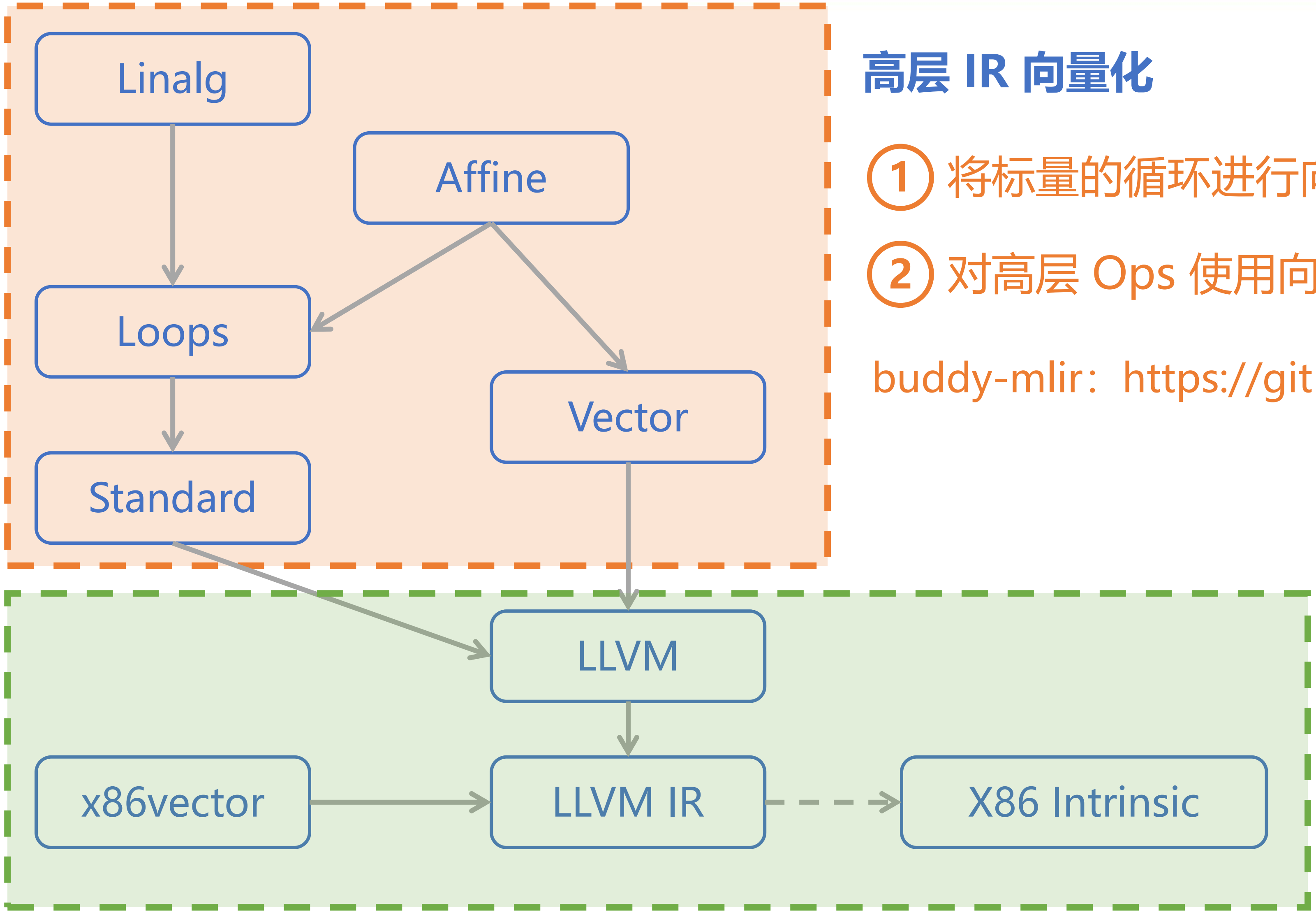
```
func @main() {
  %v1 = constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : vector<8xf32>
  %v2 = constant dense<[9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]> : vector<8xf32>
  %v3 = x86vector.avx.intr.dot %v1, %v2 : vector<8xf32>

  vector.print %v3 : vector<8xf32>

  return
}
```



使用 mlir-cpu-runner 工具

```
( 110, 110, 110, 110, 382, 382, 382, 382 )
```

总结



高层 IR 向量化

① 将标量的循环进行向量化

② 对高层 Ops 使用向量化 Pass 进行重写

buddy-mlir：https://github.com/buddy-compiler/buddy-mlir

MLIR 使用 Intrinsic

① 定义硬件向量 Dialect

② 调用硬件向量 Intrinsic